A transition system semantics for the control-driven coordination language Manifold

F. Arbab, J.W. de Bakker, M.M. Bonsangue [1]
J.J.M.M. Rutten, A. Scutellà

CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

G. Zavattaro

Bologna University, Department of Computer Science
Mura Anteo Zamboni 7, 40127 Bologna, Italy

Abstract

Coordination languages are a new class of parallel programming languages which manage the interactions among concurrent programs. Basically, coordination is achieved either by manipulating data values shared among all active processes or by dynamically evolving the interconnections among the processes as a consequence of observations of their state changes. The latter, also called control-driven coordination, is supported by Manifold. We present the formal semantics of a kernel of Manifold, based on a two-level transition system model: the first level is used to specify the ideal behavior of each single component in a Manifold system, whereas the second level captures their interactions. Although we apply our two-level model in this paper to define the semantics of a control-oriented coordination language, this approach is useful for the formal studies of other coordination models and languages as well.

Keywords and Phrases: parallel computing, coordination languages, operational semantics, transition system semantics, models of communication.

1 Introduction

Coordination models and languages represent a new approach to design and development of concurrent systems. Their purpose is to separate computing concerns from those activities that deal with the structure of an application

[1] Corresponding author. E-mail: `Marcello.Bonsangue@cwi.nl`

(communication mechanisms, protocols, system configurations, etc.) called 'coordination activities' [18,28]. The interest in coordination has intensified in the last few years, as evidenced by the increasing number of conferences, tracks, and papers devoted to this topic, and by the recent upsurge of research activity in the theoretical computer science community in this field. Furthermore, the field of coordination has a considerable overlap of interest with the work on software architectures and configuration languages, especially for systems with more complex and dynamically evolving architectures[34,22,31].

In spite of their generic label, most coordination languages are actually not languages, rather, they are only collections of primitive operations meant to augment conventional computation languages. There have been relatively few attempts to study coordination as a stand-alone programming paradigm, with its own self-contained programming language; Gamma [30] and Interaction Abstract Machines [2] are some such examples, and **Manifold**, the subject of study in this paper, is another.

Coordination models and languages can be classified as either *data-oriented* or *control-oriented* [8]. For instance, Linda [29] uses a data-oriented coordination model, whereas **Manifold** is a control-oriented coordination language. Coordination models and languages can also be classified as either *endogenous* or *exogenous* [6]. For instance, Linda is based on an endogenous model, whereas Strand [27] and **Manifold** are exogenous coordination languages [2]. Endogenous models and languages provide primitives that must be incorporated *within* a computation for its coordination. In applications that use such models, primitives that affect the coordination of each module are inside the module itself. In contrast, exogenous models and languages provide primitives that support coordination of entities from *without*. In applications that use exogenous models primitives that affect the coordination of each module are outside the module itself.

A focal point of the activity on the theoretical aspects of coordination is, of course, formal semantics. There are several attempts to define formal semantics for coordination languages based on shared data-spaces and generative communication, e.g., Linda [16,19–21,14,15], Gamma [30,17], μ-Log [33], and SPLICE [11]. On the other hand, formal treatments of the semantics of control-oriented coordination languages are very scarce. We know only of two preliminary studies both on the formal semantics of an earlier version of **Manifold** [39].

In this paper, we present the formal operational semantics for the core of the

[2] Logic-programming-based coordination models, such as Strand, essentially impose the implicit, restrictive control structure of logic programming on the components they coordinate. Other than being exogenous, there is no similarity between these models and **Manifold**.

coordination language **Manifold**, using transition systems. The study of the formal semantics of **Manifold** is interesting for at least three reasons: (1) as a pure, self-contained coordination language (as opposed to language extensions) that contains no computation primitives, **Manifold** is a unique programming language with interesting properties; (2) the view of coordination embodied in **Manifold** is actually more general than the **Manifold** language itself; and (3) the inherent characteristics of **Manifold** suggest a certain methodology for defining its formal semantics that, in our view, is applicable to a much wider spectrum of coordination models and languages as well. An exercise in the formal semantics of **Manifold** is a good way to grasp the essential concepts related to all these areas.

Everything in **Manifold** is a process: computing entities, (meta-)coordinator entities, and communication links. Our approach consists of defining an operational model for **Manifold**, based on a two-level transition system. The first level consists of a number of transition systems, each of which defines the behavior of a single process, embedded in an ideal environment and hence independent of the rest of the processes. The second level consists of a single transition system that defines the interactions among the first-level transition systems.

Multi-level transition systems were first used to define the formal semantics of coordination languages in [39]. Although both these attempts, as well as our current paper, use multi-level transition systems specifically to define the semantics of **Manifold**-like models and languages, this formalism is not specific to **Manifold** nor to control-oriented coordination. Indeed, multi-level transition systems are much more general and seem to be suitable for formalizing data-oriented coordination models and languages as well, as illustrated more recently in [17].

We use a set of first level transition systems to specify processes as autonomous entities that can compute and/or interact with their environment. Thus, every step of the computation in such a process may depend not only on the internal state of the process, but also on some input it may obtain from its environment. Such processes are open systems in a sense analogous to Wegner's notion of Interaction Machines [42]. Typically, each such transition system is unbounded and nondeterministic, reflecting the fact that the process it represents is an interactive system; i.e., its unpredictable behavior depends on the input it obtains from an external environment that it does not control. The details of the internal activity of every process (e.g., its computations) are described by its respective first-level transition system. Most such detail is irrelevant for, and hence unobservable by, the second-level transition system. The second-level transition system, thus, abstracts away the semantics of the first level processes, and is concerned only with their (mutually engaging) externally observable behavior. Dually, by (conceptually) embedding each single process in an ideal environment, its interactions with the rest of the processes become irrelevant for the definition of individual first-level transition systems.

The activity of an entire MANIFOLD application is modeled by the second-level transition system. Here, a configuration corresponds to a set of active processes (i.e., coordinators, atomic processes, and/or streams) each of which is associated with a list of pending messages that have already been broadcast but not yet received. Each second-level transition is defined in terms of a number of first-level transitions reflecting the actions of some interacting processes. The second-level transitions are based only on a partial view of the whole system, reflecting the true time and space decoupling of processes in a MANIFOLD application.

The contributions of this paper can be summarized as follows. First, the presented operational semantics is a rigorous definition of the intended meaning of MANIFOLD programs. Given that MANIFOLD is a real-life programming language for coordination, this can be considered a major achievement. It turned out to be a far from trivial task to give a mathematically precise and transparent description of the semantics of MANIFOLD, essentially because what it does is different than the semantics of traditional computation languages. Notably, the event broadcasting mechanism, on the one hand, and the management of dynamically changing process configurations, including the creation and removal of processes as well as stream connection and reconnection, on the other, each constitutes a powerful non-trivial machinery that is more generally applicable in its own right. The fact that we have succeeded in modeling all these aspects and their interplay in a uniform and transparent way, simultaneously justifies the various choices that have been made in the design of the language.

Second, we feel that some of the operational techniques that have been used in the construction of our model, are of interest in themselves, beyond the specifics of the language MANIFOLD:

- Transition systems, which are structures commonly used in operational semantics, have been used in a uniform and universal way. Every one of the three different types of processes that exist in MANIFOLD (coordinator-, stream-, and atomic processes), is modeled as a transition system (in the first level). Each such transition system describes the potential steps that its corresponding process can take, assuming that it is embedded in an environment that is optimally cooperative. The interplay of all processes put to work together is, again, described by one 'big' transition system in the second level, comprising the parallel composition of all individual first-level transition systems. This second-level transition system thus constrains and describes all the actual steps of an entire MANIFOLD program.
- Computation is performed by atomic processes only, whereas the activity of

all other processes is regarded as coordination, not computation. A clear and consistent separation of computation and coordination has been achieved by modeling atomic processes as abstract transition systems with predefined transitions; in contrast, the transitions for coordinator and stream processes are defined in all detail in the paper. The transition relation of a coordinator process is, more specifically, determined by the MANIFOLD program text it is executing.

- The embedding of atomic processes in our model is an elegant formal expression of how arbitrary black-box processes can be coopted as participants in a coordinated cooperative application, without their own knowledge.
- In our model, the autonomous status of streams (modeled as transition systems), includes an explicit and precise semantic description of their various connection modes. This constitutes a key ingredient in the modeling of asynchronous communication in general.
- The event broadcasting mechanism has been modeled using some state-of-the-art semantic tools, notably Mazurkiewicz traces [35]. In fact, the present treatment of events has already given rise to similar applications for other coordination languages, such as SPLICE [12].

The rest of this paper is organized as follows. In Section 2, we give an informal overview of the MANIFOLD language. In Section 3, we give an abstract syntax for writing coordinator processes, and characterize a class of transition systems that specify the behavior of atomic processes (the only computational entities in MANIFOLD). In Section 4, we define labeled transition systems for coordinator processes, for atomic processes, and for streams. These transition systems specify the behavior of their respective processes in isolation, without any interaction with their environment. In Section 5, the interaction between different processes and streams is modeled by a new transition system, combining different transitions of the previous systems. Thus, we obtain a formal description of the behavior of an entire MANIFOLD system. Section 6 is the conclusion of the paper, and the basic notations for partial functions as used in the paper appear in the appendix.

2 An informal overview of MANIFOLD

This section is a brief informal overview of the MANIFOLD coordination language. MANIFOLD is a control-oriented coordination language for managing complex, dynamically changing interconnections among sets of independent, concurrent, cooperating processes [5,3,4,9].

Two major concepts in MANIFOLD are separation of concerns and anonymous communication. Separation of concerns means that computation concerns are isolated from the communication and cooperation concerns. Anonymous communication means that the parties engaged in communication with each other need not know each other. Furthermore all communication is asynchronous. In MANIFOLD communication is either through broadcast of events or through point-to-point channel connections which, generally, are established between two communicating processes by a third party coordinator process.

A MANIFOLD application consists of a finite set of *process type* definitions. Because a process type is analogous to the notion of class in object oriented languages, in this paper we use the term "class" instead, whenever it helps clarity. There are two kinds of classes: computation classes and coordinator classes. A class can have formal parameters which will be replaced by their corresponding actual parameters when an instance of that class is created at run time. Instances of coordinator classes are called coordinator processes, while instances of computation classes are called atomic processes. The execution starts with the creation of an instance of the designated initial coordinator class. For simplicity, we assume in this paper that the initial coordinator class is parameterless.

Coordinator class definitions are written in the block-structured MANIFOLD language. Since their instances are involved only in coordination, there is no need for the constructs and the entities that are common in conventional programming languages such as values, expressions, and sequential composition. The only entities known to MANIFOLD are processes (including streams, which are asynchronous channels), ports, and events. The main control structure is an event-driven guarded statement evaluation mechanism. Primitive operations allow for creation, activation, and killing of processes; broadcast of events; and dynamic (re)connection of the ports of some of these processes via streams. These operations take place in a coordinator process instance as a consequence of the repeated evaluation of conditions on the events it receives from its environment. If several conditions can be successfully evaluated within a block, one is selected non-deterministically[3], and its associated actions are executed. When all actions are executed, the coordinator process goes back to evaluating the conditions again. This iteration continues, until the execution of a specific action terminates the coordinator process.

Computation class definitions may be written in different programming languages and some of them may not know anything about MANIFOLD. Instances of computation classes, also called atomic processes, typically are not aware of the fact that they are cooperating with other processes within the MANIFOLD system. An atomic process is not responsible for the communication that is

[3] This non-determinism is actually subject to a priority scheme on the matching event occurrences that can potentially open the alternative guards. This scheme ensures that the handling of events with higher priority take precedence over lower-priority events. For simplicity, we do not model this priority scheme in our formal semantics presented in this paper.

necessary for it to obtain the proper input it requires to perform its computation, nor is it responsible for the communication that is necessary to deliver the results it produces to their proper recipients. They can compute and produce and consume values through their ports, and broadcast and receive events.

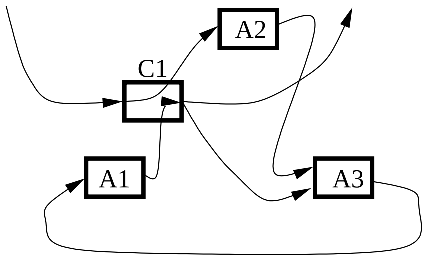

Fig. 1. Ports and streams connecting a coordinator (C1) and atomic processes (A*i*)

Both coordinator and atomic processes have named *ports* of connection. Atomic processes use them to exchange values with streams in their environment. Coordinator processes do not produce or consume values themselves; their ports are used as support for communication between two or more streams. Because both coordinator and atomic processes have ports, through which values are (or at least, seem to be) produced and consumed, they are externally indistinguishable from one another (see Figure 1).

Because atomic and coordinator processes are absolutely indistinguishable from the point of view of other processes, coordinator processes can, recursively, manage the communication of other coordinator processes just as if they were computation processes. This means that any coordinator can also be used as a higher-level or meta-coordinator, to build a sophisticated hierarchy of coordination protocols. Such higher-level coordinators are not possible in most other coordination languages and models.

A stream is a communication channel with an unbounded buffer for transporting values between the ports of atomic or coordinator processes. A stream represents a reliable and directed flow of information from its *source* to its *sink*. Once a stream is established between a (port of a) producer process and a (port of a) consumer process, it operates autonomously and transfers the values from its source to its sink. The (process at the) sink of a stream requiring a value is suspended only if no values are available in the stream. The suspended sink resumes as soon as the next value becomes available for its consumption. An attempt by the source of a stream to place a value into the stream is never suspended because of the unbounded buffer capacity of the stream. Streams can be created and broken by coordinator processes. Furthermore, they can be reconnected, meaning that one of their ends can be first disconnected and then reconnected to another port. Because a stream may contain some pending values in its buffer, it may or may not be desirable for a stream to immediately disconnect itself from its source or its sink as soon as its connection at its opposite end is broken. Therefore, it is meaningful for a stream to remain connected at one of its ends, after it is disconnected from the other. Two types of connection can be identified between a port and a stream: break-type and keep-type. A *break-type* connection between a port and a stream breaks automatically when the connection at the other end of the stream breaks. A *keep-type* connection, on the other hand, persists even after the connection at the other end of the stream breaks. The connection between a stream and a port it is connected to is severed when (1) either the stream or the process to which the port belongs dies, or (2) a coordinator process breaks up its break-type connections.

The combination of break-type and keep-type connections at the two ends of a stream lead to four different stream types designated as `BB`, `BK`, `KB`, and `KK`. The letters '`B`' and '`K`' in a stream type name respectively designate *break* and *keep* connections of a streams of that type, where the rightmost letter refers to the stream's connection with its sink, and the leftmost letter to the one with its source:

> `BB`: A stream of this type is disconnected from either its producer or consumer automatically, as soon as it is disconnected from the other.
> `BK`: A stream of this type is disconnected from its producer automatically, as soon as it is disconnected from its consumer, but disconnection from its producer does not disconnect the stream from its consumer.
> `KB`: A stream of this type is disconnected from its consumer automatically, as soon as it is disconnected from its producer, but disconnection from its consumer does not disconnect the stream from its producer.
> `KK`: A stream of this type is not disconnected from either of its processes automatically, if it is disconnected from the other.

Independent of the communication mechanism offered by the streams, there is a broadcasting mechanism for information exchange in **Manifold**. Both atomic and coordinator processes may broadcast events in their environment, each broadcast yielding an *event occurrence* for which the broadcasting process becomes its *event source*. Once an event is broadcast by a process, the latter continues with its processing, while the resulting event occurrence propagates through the environment independently. Broadcast event occurrences are eventually received in the *event memory* of every observing coordinator or atomic process. The observed event occurrences in the event memory of a process can be examined and reacted on by the observer process at its own leisure.

Programming in **Manifold** is a game of dynamically creating (coordinator and/or computation) process instances and dynamically (re)connecting the ports of some of these processes via streams, in reaction to observed event occurrences. **Manifold** encourages a discipline for the design of concurrent software that results in two separate sets of modules: pure coordination, and pure computation. This separation disentangles the semantics of computation modules from the semantics of the coordination protocols. The coordination modules construct and maintain a dynamic data-flow graph where each node is a process. These modules do no perform any computation, but only make the prescribed changes to the connections among various processes in the application, which changes only the topology of the graph. The computation modules, on the other hand, cannot possibly change the topology of this graph, making both sets of modules easier to verify and more reusable. The concept of reusable pure coordination modules in **Manifold** is demonstrated, e.g., by using (the object code of) the same **Manifold** coordinator program that was developed for a parallel/distributed bucket sort algorithm, to perform function evaluation and numerical optimization using domain decomposition[7,23].

The **Manifold** system runs on multiple platforms and consists of a compiler, a run-time system library, a number of utility programs, and libraries of builtin and predefined processes of general interest. Presently, it runs on IBM RS6000 AIX, IBM SP1/2, Solaris, Linux, Cray YMP, and SGI IRIX. A **Manifold** application consists of a (potentially very large) number of processes running on a network of heterogeneous hosts, some of which may be parallel systems. Processes in the same application may be written in different programming languages and some of them may not know anything about **Manifold**, nor the fact that they are cooperating with other processes through **Manifold** in a concurrent application. A number of these processes may run as independent operating-system-level processes, and some will run together as light-weight processes (preemptively scheduled threads) inside an operating-system-level process. None of this detail is relevant at the level of the **Manifold** source code, and the programmer need not know anything about the eventual configuration of his or her application in order to write a **Manifold** program.

Manifold has been used in a number of real applications, e.g., implementing parallel and distributed versions of a semi-coarsened multi-grid Euler solver algorithm[24,25]; modeling cooperative Information Systems[37,38]; coordination of Loosely-Coupled Genetic Algorithms on parallel and distributed platforms[41]; coordination of multiple solvers in a concurrent constraint programming system[10]; and coordination of a distributed propositional theorem checker [26].

Differences between **Manifold** *and its kernel*

Some of the features of **Manifold** have been left out of the kernel language that is the subject of our formal study in this paper. This makes the kernel simpler, but we do not believe this simplification has compromised the interesting semantic issues of the language.

A number of linguistic features of **Manifold** are common in most other modern programming languages as well. They include scope rules, separate compilation, parameterized sub-programs, import/export of entities among modules, etc. We ignore these features in our kernel because there is nothing very unusual about such features and their semantics are well understood for many programming languages. Furthermore, **Manifold** supports some syntactic sugar for common control structures and expressions. In reality, only the front end of the **Manifold** compiler knows about such conventions and internally translates them into their corresponding normal syntax. We have left out these "extensions" because, effectively, they "do not exist" even in the real **Manifold** language either. A few coordinator types are predefined in **Manifold**. They provide special services, such as guarding ports, obtaining single values from ports, producing reference values, and dereferencing them. In our kernel, such special purpose coordinators are nothing special.

Finally, there are a number of more significant features that have been left out of the kernel for simplicity. Although these features do have an impact on the practical usefulness of the language, their semantic significance is too small to justify the added volume of their inclusion in this paper. Among them are:

- Scope rules and statements that control the broadcast of events. They support static and dynamic means for "programming" which event sources are observable to which processes.
- Event handling declarations. Event occurrences can be saved, ignored, and assigned higher or lower priorities by each process for its own purposes.
- Death of streams. A stream automatically dies when it detects certain conditions that signify it is no longer needed.
- Flow of values. Port managers (see Section 5) are also aware of the flow of values through their ports, in order to allow the evaluation of flow related port conditions.
- Opening and closing individual ports.
- Automatic breakup of streams at the end of each block.

3 Abstract syntax of Manifold

We assume the existence of two finite disjoint sets of types designating the classes for coordinator and atomic processes: `CType`, ranged over by `C`, and `AType` ranged over by `A`, respectively. Both coordinator and atomic processes have *ports* through which values are exchanged with or among streams. We assume the existence of a finite set of ports, `Ports`, ranged over by `i`, `o`.

Each of the four stream types, mentioned earlier, will get a fixed interpretation.

We define the set $\texttt{SType}$ of stream types, with its typical element $\texttt{S}$, as

$$\texttt{SType} = \{\texttt{BB}, \texttt{BK}, \texttt{KB}, \texttt{KK}\},$$

and assume $\texttt{SType}$ is disjoint from both $\texttt{AType}$ and $\texttt{CType}$.

In this section, we define the abstract syntax for coordinator classes. No explicit syntax will be defined for computation classes: they may be written in any programming language, about which we do not wish to make any assumptions. Instead, as explained in the introduction, we define computation classes in terms of abstract transition systems that model their externally observable behavior.

Coordinator classes

A coordinator process refers to other processes, events, streams, ports, and process types by means of elements of the following countable and disjoint sets:

- *PrcNm* of *process names*, ranged over by p, q;
- *StrNm* of *stream names*, ranged over by s;
- *EvtNm* of *event names*, ranged over by e, f;
- *PrcTypNm* of *process type names*;
- *StrTypNm* of *stream type names*;
- *PrtNm* of *port names*.

There are special event names $\texttt{begin}$, $\texttt{die}$, and $\texttt{stop}$ denoting, respectively, the request for starting a process, the request for terminating a process, and the actual termination of a process. Also, we assume that $\texttt{0}$ and *self* are special process names denoting the system and the name used by each process to denote itself. Furthermore, for each process type $\texttt{P} \in \texttt{AType} \cup \texttt{CType}$ there is a countable set of process names $\mathit{PrcNm}(\texttt{P})$ that excludes $\texttt{0}$ and *self*, such that $\mathit{PrcNm}(\texttt{P}_1)$ is disjoint from $\mathit{PrcNm}(\texttt{P}_2)$ if $\texttt{P}_1 \neq \texttt{P}_2$. If $p \in \mathit{PrcNm}(\texttt{P})$ then we say that the process named p is of type $\texttt{P}$. For simplicity, we define the function $\mathit{ClassOf} \colon \mathit{PrcNm} \rightharpoonup \texttt{AType} \cup \texttt{CType}$ such that $\mathit{ClassOf}(p) = \texttt{P}$ if and only if $p \in \mathit{PrcNm}(\texttt{P})$.

We denote by *Const* the set of *constants* obtained as the union of $\texttt{CType}$, $\texttt{AType}$, $\texttt{SType}$, $\texttt{Ports}$ and $\{\texttt{0}, \texttt{begin}, \texttt{die}, \texttt{stop}\}$. Also, we denote by *Nm* the set of *names* obtained as the union of *PrcNm*, *StrNm*, *EvtNm*, *PrcTypNm*, *StrTypNm*, *PrtNm*, $\texttt{CType}$, $\texttt{AType}$, $\texttt{SType}$, and $\mathit{PrcNm} \times \texttt{Ports}$. An element of $\mathit{PrcNm} \times \texttt{Ports}$ is a pair, say $p.\texttt{i}$, denoting the port $\texttt{i}$ of a process named p. Note that constants in $\texttt{Ports}$ are not included in *Nm* because a port must always be paired with the name of its owner process. The set of names is ranged over by n, m, x. We use the convention of writing in italic those names that can be replaced by other names in a substitution (defined below).

In order to specify a coordinator class we need the following grammar:

$$
\begin{aligned}
G &::= ep?S \mid (G + G)\,, \\
ep &::= (\mathtt{0}, \mathtt{begin}) \mid (p, e) \mid (*p, e) \mid (p, *e) \mid (*p, *e) \\
S &::= \mathbf{end} \mid \mathbf{halt} \mid \mu.S \mid \ \{\!|\ G\ |\!\}.S \\
\mu &::= \mathbf{evn}(e) \mid \mathbf{raise}(e) \mid \mathbf{post}(e) \mid \mathbf{prc}(p{:}\pi, \vec{n}) \mid \mathbf{start}(p) \mid \mathbf{finish}(p) \mid \\
&\qquad \mathbf{str}(s{:}\rho, \alpha, \beta) \mid \mathbf{break}(s) \mid \mathbf{src}(s, \alpha) \mid \mathbf{snk}(s, \beta)
\end{aligned}
$$

In this grammar, e is an event name in *EvtNm* but not in *Const*, p is a process name in *PrcNm* other than `0`; s is a stream name in *StrNm*; α, β are either port names in *PrtNm* or pairs in *PrcNm* $\times$ `Ports` with the left component other than `0`; $\vec{n} \in Nm$ is a (possibly empty) list of names; π is either a process type in `CType` $\cup$ `AType` or a process type name in *PrcTypNm*; and ρ is either a stream type in `SType` or a stream type name in *StrTypNm*. We use lists where order matters (e.g., matching of parameters) but we often treat them as sets, abusing the formal notation accordingly. We use the operator $\otimes$ to construct lists of names; e.g., $w{\otimes}x{\otimes}y{\otimes}z$ is a list of the four elements w, x, y, and z.

In the above grammar, we say that G is a *command*, or guarded statement, ep is an *event pattern*, S is a *statement* and μ an action. We let *EvnPat* be the set of all event patterns.

A *coordinator class* is a pair $\langle G, P\rangle$, where G is a command, and $P \subseteq$ `Ports` is a set declaring the ports of the class. The execution of an instance of the class $\langle G, P\rangle$ loops over the execution of the command G until the statement **halt** is executed.

The command $ep?S$ matches the event pattern ep with stored event occurrences (to be defined below). The $*$ prefixed names can match with the process and/or event names of any event occurrence, yielding bound names in the scope of S. They act as formal parameter of the statement S. For example, in the command $(p, *e)?S$ the event pattern $(p, *e)$ can match with the event occurrence (p, e') and yield a version of the statement S where the actual event name e' is substituted for the formal event name e. The $+$ operator is a *guarded sum*, where all alternatives are statements guarded by their event patterns. The $+$ operator selects one of its alternatives non-deterministically [4].

Next, we informally describe the meaning of the statements.

[4] See footnote 3.

- **end** is the terminated statement.
- **halt** is the statement that causes the termination of the process.
- $\mathbf{evn}(e).S$ is the declaration of a new event. It creates a fresh event name, say e', and reduces to a version of the statement S where e' is substituted for e.
- $\mathbf{raise}(e).S$ broadcasts the event name e to all processes and becomes S.
- $\mathbf{post}(e).S$ sends the event name e only to the process executing it and becomes S.
- $\mathbf{prc}(p{:}\pi, \vec{n}).S$ is executed if π is a coordinator or atomic process type, say $\mathtt{P}$, and the actual parameters $\vec{n}$ are consistent with the formal parameters of the declaration of $\mathtt{P}$. It creates a new instance of a process of type $\mathtt{P}$ and assigns to it a fresh process name in $\mathit{PrcNm}(\mathtt{P})$, say q. The list of names $\vec{n}$ is passed to the new process q as its actual parameters. This statement then reduces to a version of S where the name q is substituted for p.
- $\mathbf{start}(p).S$ sends a message for starting the execution of the process p and becomes S.
- $\mathbf{finish}(p).S$ sends a message for finishing the execution of the process p and becomes S.
- $\mathbf{str}(s{:}\rho, \alpha, \beta).S$ is executed if ρ is a stream type, say $\mathtt{S}$, and α and β are pairs in $\mathit{PrcNm} \times \mathtt{Ports}$, say $p.\mathtt{i}$ and $q.\mathtt{o}$, respectively. It creates a new stream of type $\mathtt{S}$ connecting the port $\mathtt{i}$ of process p to the port $\mathtt{o}$ of process q, and assigns to it a fresh stream name, say t. This statement then reduces to a version of S where the name t is substituted for s.
- $\mathbf{break}(s).S$ severs the break-type connections of the stream s, if any, and becomes S.
- $\mathbf{src}(s, \alpha).S$ is executed if α is a pair in $\mathit{PrcNm} \times \mathtt{Ports}$, say $p.\mathtt{i}$. It reconnects the source of a stream s to the port $\mathtt{i}$ of process p and becomes S.
- $\mathbf{snk}(s, \beta).S$ is executed if β is a pair in $\mathit{PrcNm} \times \mathtt{Ports}$, say $q.\mathtt{o}$. It reconnects the sink of the stream s to the port $\mathtt{o}$ of process q and becomes S.
- $\{\!|\ G\ |\!\}.S$ executes the command G and when G terminates it becomes the statement S.

The set $bn(G) \subseteq \mathit{Nm}$ of *bound names* in a command G is defined inductively as follows:

$$
\begin{aligned}
bn((0, \mathtt{begin})?S) &= \{\mathit{self}\} \cup bn(S) \\
bn((p, e)?S) &= \{\mathit{self}\} \cup bn(S) \\
bn((*p, *e)?S) &= \{\mathit{self}, p, e\} \cup bn(S) \\
bn((p, *e)?S) &= \{\mathit{self}, e\} \cup bn(S) \\
bn((*p, e)?S) &= \{\mathit{self}, p\} \cup bn(S) \\
bn(G_1 + G_2) &= (bn(G_1) \setminus \mathit{fn}(G_2)) \cup (bn(G_2) \setminus \mathit{fn}(G_1))\,,
\end{aligned}
$$

where $fn(G)$ is the set of *free names* of G defined as those names occurring in G but not in $bn(G) \cup Const$, and, for each statement S, the set $bn(S)$ of bound names of S is defined inductively by

$$\begin{aligned}
bn(\mathbf{end}) &= \emptyset \\
bn(\mathbf{halt}) &= \emptyset \\
bn(\mathbf{evn}(e).S) &= \{e\} \cup bn(S) \\
bn(\mathbf{prc}(p{:}\pi, \vec{n}).S) &= \{p, \mathtt{self}\} \cup bn(S) \\
bn(\mathbf{str}(s{:}\rho, \alpha, \beta).S) &= \{s, \mathtt{self}\} \cup bn(S) \\
bn(\{\!|\ G\ |\!\}.S) &= (bn(G) \setminus fn(S)) \cup bn(S)\,,
\end{aligned}$$

and $bn(\mu.S) = bn(S)$ for all other actions μ. As before, the set of free names $fn(S)$ of a statement S is defined by those names occurring in S but not in $bn(S) \cup Const$. For example, in the statement

$$\mathbf{str}(s{:}\mathtt{BK}, p.\mathtt{i}, x).\mathbf{end}$$

the names $s \in StrNm$ and $\mathtt{BK} \in \mathtt{SType}$ are bound, while $p \in PrcNm$ and $x \in PrtNm$ are free. Note that $\mathtt{i} \in \mathtt{Ports}$ is not a name in Nm, and therefore is neither free nor bound. Free names are formal parameters to be replaced by actual ones at the moment a process is created.

As usual, free names can be replaced by other names. *Substitution* $G[n/m]$ of a name m for a name n on a command G is defined inductiveley as follows:

- $(ep?S)[n/m]$ is the command $(ep[n/m])?(S[n/m])$, where $ep[n/m]$ is the event pattern obtained by substituting the name m for the name n in ep if $n \in fn(ep?S)$, and $S[n/m]$ is the statement obtained by substituting m for n in S (to be defined below);
- $(G_1 + G_2)[n/m] = (G_1[n/m]) + (G_2[n/m])$.

Substitution $S[n/m]$ of a name m for a name n on a statement S is define similarly:

- $\mathbf{end}[n/m] = \mathbf{end}$;
- $\mathbf{halt}[n/m] = \mathbf{halt}$;
- $(\mu.S)[n/m]$ is the statement $(\mu[n/m]).(S[n/m])$, where $\mu[n/m]$ is the action obtained by substituting the name m for the name n in μ if $n \in fn(\mu.S)$;
- $(\{\!|\ G\ |\!\}.S)[n/m] = \{\!|\ G[n/m]\ |\!\}.(S[n/m])$.

We require the following consistency between names in a substitution:

- a process name $p \in PrcNm$ can be replaced only by a process name $q \in$

PrcNm;

- a stream name $s \in \mathit{StrNm}$ can be replaced only by a stream name $t \in \mathit{StrNm}$;
- an event name $e \in \mathit{EvtNm}$ can be replaced only by an event name $f \in \mathit{EvtNm}$;
- a process type name $c \in \mathit{PrcTypNm}$ can be replaced by either a process type name $d \in \mathit{PrcTypNm}$, a coordinator type $\mathtt{P} \in \mathtt{CType}$, or an atomic type $\mathtt{Q} \in \mathtt{AType}$ only;
- a stream type name $k \in \mathit{StrTypNm}$ can be replaced by either a stream type name $j \in \mathit{StrTypNm}$ or a stream type $y \in \mathtt{SType}$ only;
- a port name $a \in \mathit{PrtNm}$ can be replaced by either a port name $b \in \mathit{PrtNm}$ or a pair $p.x \in \mathit{PrcNm} \times \mathtt{Ports}$ only.

For $\vec{n} = n_1 \cdots n_l$ and $\vec{m} = m_1 \cdots m_l$ we denote by $S[\vec{n}/\vec{m}]$ the statement obtained by simultaneously substituting in S all occurrences of every n_i with its corresponding m_i.

For every coordinator type $\mathtt{C} \in \mathtt{CType}$ there is a unique declaration

$$\mathtt{C}(\vec{x}) = \langle G_{\mathtt{C}}, P_{\mathtt{C}} \rangle$$

where $\vec{x}$ is a list of distinct names in Nm, $G_{\mathtt{C}}$ is a command and $P_{\mathtt{C}}$ is a subset of $\mathtt{Ports}$. We say that $\mathtt{i}$ *is a port of* $\mathtt{C}$ if $\mathtt{i} \in P_{\mathtt{C}}$. The names in $\vec{x}$ are the formal parameters that will be replaced by actual ones at the moment a new instance of the class $\langle G_{\mathtt{C}}, P_{\mathtt{C}} \rangle$ is created. All free names in the command $G_{\mathtt{C}}$ are in $\vec{x}$. Note that since $\mathtt{Ports}$ and Nm are disjoint, $P_{\mathtt{C}}$ and $\vec{x}$ are disjoint.

Computation classes

Computation classes may be written in different programming languages and are declared using atomic types. Instances of computation classes use values to carry out their computation. Therefore, we assume the existence of an abstract set $\mathtt{Val}$ of *values*, ranged over by v. Values are produced and consumed by atomic processes and are transported through streams.

For every atomic type $\mathtt{A} \in \mathtt{AType}$ and list $\vec{n}$ of a given fixed length of distinct names in Nm there is a unique declaration

$$\mathtt{A}(\vec{n}) = \langle \Sigma_{\mathtt{A},\vec{n}}, \mathit{Act}_{\mathtt{A},\vec{n}}, \longrightarrow_{\mathtt{A},\vec{n}}, \sigma_{\mathtt{A},\vec{n}}, P_{\mathtt{A},\vec{n}} \rangle \, .$$

As for coordinator classes, computation classes also have formal parameters. The list of names $\vec{n}$ represents the actual parameter, and right hand side of the above declaration is the computation class obtained by substituting actual parameters for the formal ones (which in our abstract setting need not be mentioned). A *computation class* consists of a transition system with an abstract set of states ($\sigma \in$) $\Sigma_{\mathtt{A},\vec{n}}$, a set of observable actions ($a \in$) $Act_{\mathtt{A},\vec{n}}$, and a (nondeterministic) labelled transition relation $\longrightarrow_{\mathtt{A},\vec{n}} \subseteq \Sigma_{\mathtt{A},\vec{n}} \times Act_{\mathtt{A},\vec{n}} \times \Sigma_{\mathtt{A},\vec{n}}$ that describes the behavior of the instances of the class. We assume that the sets of internal states of transition systems of different types and for different actual parameters are disjoint. Intuitively, $\sigma \xrightarrow{a}_{\mathtt{A},\vec{n}} \sigma'$ denotes a transition step in which the action $a \in Act_{\mathtt{A},\vec{n}}$ is executed, causing the state to change from σ to σ'. An instance of this class starts its activity in the initial state $\sigma_{\mathtt{A},\vec{n}}$. The last component $P_{\mathtt{A},\vec{n}} \subseteq \mathtt{Ports}$ contains the ports of the class. We require that $P_{\mathtt{A},\vec{n}} = P_{\mathtt{A},\vec{m}}$ also for $\vec{n} \neq \vec{m}$ and therefore say that $\mathtt{i}$ *is a port of* $\mathtt{A}$ if $\mathtt{i} \in P_{\mathtt{A},\vec{n}}$, for some list $\vec{n}$ of event names in $EvtNm$.

The set $Act_{\mathtt{A},\vec{n}}$ of observable actions of a computation class of type $\mathtt{A}$ with actual parameter $\vec{n}$ is defined as follows:

$$\begin{aligned} Act_{\mathtt{A},\vec{n}} = \; & \{\tau, \mathit{halt}\} \\ & \cup \{\mathit{raise}(e) \mid e \in \vec{n} \cap \mathit{EvtNm}\} \\ & \cup \{\mathit{receive}(e) \mid e \in \vec{n} \cap \mathit{EvtNm}\} \\ & \cup \{\mathit{get}(\mathtt{i}, v) \mid \mathtt{i} \in P_{\mathtt{A},\vec{n}}, v \in \mathtt{Val}\} \\ & \cup \{\mathit{put}(\mathtt{o}, v) \mid \mathtt{o} \in P_{\mathtt{A},\vec{n}}, v \in \mathtt{Val}\} . \end{aligned}$$

The action τ represents an internal action, about which no assumptions are made. It may, for instance, be the update of a local store (which is externally invisible). The action *halt* indicates that the process has finished its activity. An atomic process may broadcast an event name e by executing an action $\mathit{raise}(e)$, as long as e belongs to the parameters of the declaration. By performing the action $\mathit{receive}(e)$, an atomic process can at any stage receive an event name e broadcast by other processes if e is one of the names passed as parameter to $\mathtt{A}$. This permanent 'event-enabledness' is formally expressed by the following condition, which the transition system must satisfy:

$$\forall \sigma \in \Sigma_{\mathtt{A},\vec{n}} \, \forall e \in \vec{n} \cap \mathit{EvtNm} \, \exists \sigma' \in \Sigma_{\mathtt{A},\vec{n}}, \; \sigma \xrightarrow{\mathit{receive}(e)}_{\mathtt{A},\vec{n}} \sigma' .$$

Atomic processes may read or write values through their ports, by performing the actions $\mathit{get}(\mathtt{i}, v)$ and $\mathit{put}(\mathtt{o}, v)$. If at some moment an atomic process is willing to read a value through one of its ports, then it should be willing to accept *any* value. This results in a second condition on the transition relation:

$$\forall \sigma \xrightarrow{\mathit{get}(\mathtt{i},v)}_{\mathtt{A},\vec{n}} \sigma' \, \forall v' \in \mathit{Val} \, \exists \sigma'' \in \Sigma_{\mathtt{A},\vec{n}}, \; \sigma \xrightarrow{\mathit{get}(\mathtt{i},v')}_{\mathtt{A},\vec{n}} \sigma'' .$$

Note that the computation of a process specified by a computational class may continue also after the action *halt* has been executed. However, within the **Manifold** system, a process is executed until the action *halt* is observed.

Finally, for a fixed atomic type $\mathtt{A} \in \mathtt{AType}$ we require coherence for the computation class associated to $\mathtt{A}$ with respect to different lists of formal parameters: for all lists $\vec{n}$ and $\vec{m}$ of equal length of names in Nm we require that there is a bijection $f \colon \Sigma_{\mathtt{A},\vec{n}} \to \Sigma_{\mathtt{A},\vec{m}}$ such that

- it preserves the initial state, i.e. $f(\sigma_{\mathtt{A},\vec{n}}) = \sigma_{\mathtt{A},\vec{m}}$, and
- it preserves the transition relation, i.e. $\sigma \xrightarrow{a}_{\mathtt{A},\vec{n}} \sigma'$ if and only if $f(\sigma) \xrightarrow{a'}_{\mathtt{A},\vec{n}} f(\sigma')$, where a' is the action a in which every occurrence of a name in $\vec{n}$ is replaced by its corresponding name in $\vec{m}$.

Examples

The examples in this section illustrate some simple **Manifold** programs written in our abstract syntax introduced above. As a first example, we use a client-server application. Clients ask for services by raising events. Reacting to such an event, the server creates and "hooks up" a specific process that will provide the desired service to its requesting client. For instance, we can imagine requests to use electronic mail or File Transfer Protocol facilities.

Our application consists of the following process types (classes):

$$\mathtt{CType} = \{\mathtt{Main}, \mathtt{Srv}\}$$
$$\mathtt{AType} = \{\mathtt{Elm}, \mathtt{Ftp}, \mathtt{Clt}\}$$

The coordination type `Main` creates and activates the clients and the server processes as instances of the atomic type `Clt` and the coordinator type `Srv`, respectively. The computation types `Elm` and `Ftp` represent, respectively, electronic mail and File Transfer Protocol facilities. Declarations for the coordinator classes are as follows:

$$\mathtt{Main}(\varepsilon) = \langle G_{\mathtt{Main}}, \emptyset \rangle$$
$$\mathtt{Srv}(\mathit{mail\_req} {\otimes} \mathit{ftp\_req}) = \langle G_{\mathtt{Srv}}, \{\mathtt{in}, \mathtt{out}\} \rangle$$

The definition of the body for the coordinator types `Main` and `Srv` is given below. The computation types `Elm` and `Ftp` have associated declarations

$$\mathtt{Elm}(\varepsilon) = \langle \Sigma_{\mathtt{Elm}}, Act_{\mathtt{Elm}}, \to, \sigma_0, \{\mathtt{in}, \mathtt{out}\} \rangle$$
$$\mathtt{Ftp}(\varepsilon) = \langle \Sigma_{\mathtt{Ftp}}, Act_{\mathtt{Ftp}}, \to, \sigma_0, \{\mathtt{in}, \mathtt{out}\} \rangle$$

and the declaration for the computation type `Clt` is

$$\mathtt{Clt}(ev_1 {\otimes} ev_2) = \langle \Sigma_{\mathtt{Clt}}, \to, \sigma_0, \{\mathtt{in}, \mathtt{out}\} \rangle.$$

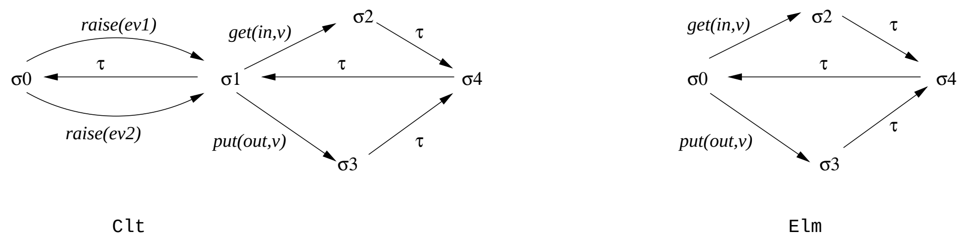

Fig. 2. Transition relations for `Clt` and `Elm`

The transition relations that define the behavior of the computation classes `Clt` and `Elm` can be summarized as in Figure 2. We omit the transitions $\sigma_i \xrightarrow{receive(ev_j)} \sigma_i$, for $0 \leq i \leq 4$ and $j \in \{1,2\}$, for the class `Clt` in this figure because events ev_1 and ev_2 are meant to be raised only and have no effect when received. The transition relations for the class `Ftp` are the same as for `Elm`. The internal transitions of these processes are uninteresting for their coordination and are thus simply summarized as transitions labeled τ.

The body for the coordinator class type `Main` creates and activates one instance of `Srv` and three instances of `Clt`.

$$\begin{aligned} G_{\texttt{Main}} \equiv (\texttt{0}, \texttt{begin})?\ & \mathbf{evn}(\mathit{mail}).\mathbf{evn}(\mathit{ftp}). \\ & \mathbf{prc}(\mathit{Server}{:}\texttt{Srv}, \mathit{mail}{\otimes}\mathit{ftp}).\mathbf{prc}(C1{:}\texttt{Clt}, \mathit{mail}{\otimes}\mathit{ftp}). \\ & \mathbf{prc}(C2{:}\texttt{Clt}, \mathit{mail}{\otimes}\mathit{ftp}).\mathbf{prc}(C3{:}\texttt{Clt}, \mathit{mail}{\otimes}\mathit{ftp}). \\ & \mathbf{start}(\mathit{Server}).\mathbf{start}(C1).\mathbf{start}(C2).\mathbf{start}(C3).\mathbf{halt} \end{aligned}$$

The body for the coordinator class `Srv` is given below. It reacts to a request (for a service) from a client by setting up two connections (streams): one from the input port of the client to the output port of the process providing the requested service, and vice versa. The process name p in $(*p, \mathit{mail})$ and $(*p, \mathit{ftp})$ will be replaced by a specific client's name at run time through matches with the occurrences of events *mail* and *ftp* raised by instances of `Clt`.

$$\begin{aligned} G_{\texttt{Srv}} \equiv (*p, \mathit{mail\_req})?\ & \mathbf{prc}(q{:}\texttt{Elm}, \varepsilon).\mathbf{start}(q). \\ & \mathbf{str}(s1{:}\texttt{BK}, p.\texttt{out}, q.\texttt{in}).\mathbf{str}(s2{:}\texttt{BK}, q.\texttt{out}, p.\texttt{in}). \\ & \mathbf{end} \\ + & \\ (*p, \mathit{ftp\_req})?\ & \mathbf{prc}(r{:}\texttt{Ftp}, \varepsilon).\mathbf{start}(r). \\ & \mathbf{str}(s3{:}\texttt{BK}, p.\texttt{out}, r.\texttt{in}).\mathbf{str}(s4{:}\texttt{BK}, r.\texttt{out}, p.\texttt{in}). \\ & \mathbf{end} \end{aligned}$$

Note that the process *Server* never ends.

As our second example, we consider a simplified version of the real MANIFOLD program called *ProtocolX* in reference [7]. This program was originally developed to coordinate a dynamic data-flow network of atomic processes to perform a distributed bucket-sort. The separation of concerns principle of MANIFOLD implies that the actual computation (i.e., sorting) coordinated by this program is a completely irrelevant detail; this program is simply an expression of a recursive coordination scheme suitable, e.g., for a class of divide-and-concur type applications. For instance, this same program is used to coordinate a distributed numerical optimization through dynamic domain decomposition application. This reusable pure coordination module is more properly described in reference [7]. The purpose of our presentation here is to demonstrate the expressive power of our kernel language by showing its closeness to the real MANIFOLD language.

$\mathtt{Sorter}(\mathit{AtomicSorter} \otimes \mathit{Merger}) = \langle G_{\mathtt{Sorter}}, \{\mathtt{in}, \mathtt{out}\} \rangle$

$$
\begin{array}{l}
G_{\mathtt{Sorter}} \equiv \\
\quad (\mathtt{0}, \mathtt{begin})? \\
\quad\quad \mathbf{evn}(\mathit{filled}).\mathbf{evn}(\mathit{finished}).\mathbf{evn}(e). \\
\quad\quad \mathbf{prc}(g1{:}\mathtt{PortGuard}, \mathtt{self.in} \otimes (\mathit{a\_everdisconnected} \otimes \mathit{empty}) \otimes \mathit{finished}). \\
\quad\quad \mathbf{start}(g1). \\
\quad\quad \mathbf{prc}(\mathit{atomicsorter}{:}\mathit{AtomicSorter}, \mathit{filled}).\mathbf{start}(\mathit{atomicsorter}).\mathbf{post}(e).\{| \\
\quad\quad\quad (\mathtt{self}, e)? \\
\quad\quad\quad\quad \mathbf{str}(s1{:}\mathtt{KB}, \mathtt{self.in}, \mathit{atomicsorter}.\mathtt{in}).\mathbf{end} \\
\quad\quad |\}.\{| \\
\quad\quad\quad (g1, \mathit{finished})? \\
\quad\quad\quad\quad \mathbf{post}(e).\{| \\
\quad\quad\quad\quad\quad (\mathtt{self}, e)?\mathbf{str}(s5{:}\mathtt{BK}, \mathit{atomicsorter}.\mathtt{out}, \mathtt{self.out}).\mathbf{end} \\
\quad\quad\quad\quad |\}.\mathbf{break}(s1).\mathbf{end} \\
\quad\quad\quad + \\
\quad\quad\quad (\mathit{atomicsorter}, \mathit{filled})? \\
\quad\quad\quad\quad \mathbf{prc}(\mathit{newsorter}{:}\mathtt{Sorter}, \mathit{AtomicSorter} \otimes \mathit{Merger}).\mathbf{start}(\mathit{newsorter}). \\
\quad\quad\quad\quad \mathbf{prc}(\mathit{merger}{:}\mathit{Merger}, \varepsilon).\mathbf{start}(\mathit{merger}).\mathbf{post}(e).\{| \\
\quad\quad\quad\quad\quad (\mathtt{self}, e)? \\
\quad\quad\quad\quad\quad\quad \mathbf{snk}(s1, \mathit{newsorter}.\mathtt{in}). \\
\quad\quad\quad\quad\quad\quad \mathbf{str}(s2{:}\mathtt{BK}, \mathit{newsorter}.\mathtt{out}, \mathit{merger}.\mathtt{in}). \\
\quad\quad\quad\quad\quad\quad \mathbf{str}(s3{:}\mathtt{BK}, \mathit{atomicsorter}.\mathtt{out}, \mathit{merger}.\mathtt{in}). \\
\quad\quad\quad\quad\quad\quad \mathbf{str}(s4{:}\mathtt{BK}, \mathit{merger}.\mathtt{out}, \mathtt{self.out}).\mathbf{end} \\
\quad\quad\quad\quad |\}.\{| \\
\quad\quad\quad\quad\quad (g1, \mathit{finished})?\ \mathbf{break}(s1).\mathbf{end} \\
\quad\quad\quad\quad |\}.\mathbf{end} \\
\quad\quad |\}.\mathbf{end}
\end{array}
$$

Although, as mentioned above, the program knows nothing about sorting, in the following presentation, for didactical reasons, we assume that its purpose is to sort an unspecified number of values that arrive through its input port. The `Sorter` coordinator has two ports, `in` and `out`, and takes two classes as its arguments: *AtomicSorter* and *Merger*. The body of `Sorter` is $G_{\texttt{Sorter}}$. An instance of `Sorter`, thus, first declares some events and then creates *g1* as an instance of another coordinator class called `PortGuard`, passing it three parameters: a port, `self.in`, a list, ($a\_everdisconnected \otimes empty$), and an event, *finished*. `PortGuard` represents a predefined class in the real MANIFOLD language.

Note that for our purposes, there is nothing special about `PortGuard`: it is just a coordination class that happens to be predefined. An instance of `PortGuard` watches the port it is passed as its first argument and when the sequence of *port-conditions* in its second argument is satisfied for that port, it raises the event that is its third argument. [5] The elements of a port-conditions list are members of a fixed set of predefined predicates on the status of ports in MANIFOLD. Each such predicate is satisfied when a certain fact about the (history, state, or incident of) (dis)connection of a port and its (incoming or outgoing) streams becomes true (12 connection predicates). Three other predicates in this set deal with the (un)availability and the flow of values through ports. [6] For instance, the condition *a_everdisconnected* is true for a port if an incoming stream has ever been disconnected from that port [7]. The *empty* condition is true if the port has a connection with at least one outgoing stream that contains no value. See reference [7] for more details.

A `Sorter` instance then creates an instance of the *AtomicSorter* class, which it calls *atomicsorter*, passing it the event *filled* as its parameter. The process *atomicsorter* is expected to wait until it can read $n > 0$ values through its input port `in`, raise the *filled* event when the n values have been read (each process dynamically decides for itself what its number n is), sort the values, and write them out in their sorted order through its output port `out`. The `Sorter` instance then connects its own `in` port to the `in` port of *atomicsorter* and enters a block.

At this point, one of two things can happen: either *atomicsorter* raises *filled* or $g1$ raises *finished*. The event *filled* means that *atomicsorter* has read in the number of values it is willing to sort, and the rest of the input values now

[5] Having a list as an argument is a slight deviation from our strict simplified syntax in this paper; but allowing ourselves a bit of syntactic freedom for the exceptional case of a predefined class is not a big indulgence!

[6] Although not mentioned explicitly, the port manager construct described in Section 5 in this paper already contains enough information to allow `PortGuard` successfully evaluate 8 of its 12 connection predicates. The rest of the conditions require straight-forward extensions which are too distracting to mention in this paper.

[7] In the terminology of Section 5, if the cardinality of the set I of the port has ever dropped to 0.

must be diverted to a new instance of `Sorter`. The event *finished* means that all input values have passed through and have been consumed. In reaction to *finished*, the `out` port of *atomicsorter* is connected to the `out` port of the `Sorter` instance, $s1$ is disconnected (to make *atomicsorter* realize the end of its input), and the `Sorter` instance suspends. In reaction to *filled*, *newsorter* and *merger* are created as new instances of `Sorter` and *Merger* classes, respectively. The rest of the input to the `Sorter` instance (part of which may be contained in the buffer of the stream $s1$) and the output of *atomicsorter* are fed into *merger*, and the `out` port of *merger* is connected to the `out` port of the `Sorter` instance. The `Sorter` instance then waits for *finished*, in reaction to which it disconnects $s1$ (to make *newsorter* realize the end of its input) and suspends.

4 Semantics of MANIFOLD: first level

In this section we define labeled transition systems for instances of coordinator classes, instances of computation classes, and for streams. We specify processes as autonomous systems that can compute and interact with their environment. Therefore, each such transition system is typically non-deterministic and unbounded, reflecting an unpredictable behavior which depends on the input a process obtains as the result of an interaction with its environment.

In Section 5, we give a transition system for an entire **MANIFOLD** system, wherein the various first-level transition systems defined in the present section are to be embedded and 'put to work'. Thus, some of the definitions in the present section can be fully understood only in the context of the definitions of Section 5, and the reader may have to postpone grasping the details of the full picture until the end of that section.

4.1 Coordinator processes

The transition system we present for a coordinator process is driven by the syntactic structure of the class of which it is an instance. First, we introduce the semantic domains and functions we use to define the semantics of coordinator processes.

Several streams can be connected to have the same port as their common source. When a value becomes available through such a port, it is replicated and a copy is inserted in each stream that has this port as its source. The replication of values can create some inconsistencies with respect to the expected result. Suppose p is an atomic process that continuously produces values through its port x, and let q be a coordinator process that creates within the same block, two streams s_1 and s_2, both connected to the port x of p as their common source. Note that streams can consume values from their sources as soon as they are created (see Section 4.3) and that two streams cannot both be created instantaneously by the same coordinator process. Thus, it is possible for one of the two streams to consume and carry some values before the second one is in place. This contradicts the intended semantics of MANIFOLD and to rectify it we use a port locking mechanism analogous to the one used in its actual implementation.

If a port is locked, no value can pass through it. In Section 5, we define this concept more precisely by associating with each process a port manager. Informally, each time a new stream is connected to a source port within a block of a coordinator process, the port manager of the process that owns the port locks that port. When a coordinator process finishes the execution of a block construct, all ports used as sources of the streams created or reconnected in that block are unlocked. To this end, each coordinator process stores within a block the number of streams connected in that block to each source port. This information is modeled by a partial function in

$$\mathit{Unlocks} = (\mathit{PrcNm} \times \mathtt{Ports}) \rightharpoonup \mathbb{N}\,.$$

For $U \in \mathit{Unlocks}$, if $U(\alpha) > 0$ then the port α must be unlocked $U(\alpha)$ times at the end of the block. Note that $U(\alpha) = 0$ does not necessarily mean that the port α is completely unlocked; other stream connections to this same source port constructed by other coordinator processes that may not have yet finished the execution of their respective blocks may still be keeping the port locked.

In a coordinator process the unlock information is local within each block. Since blocks can be nested, we use a stack to maintain the unlock information. Each time a new block is entered an empty unlock information is allocated onto the top of the stack, and when a coordinator process exits from a block, its unlock information is popped from the stack.

In Section 2, we saw that in addition to the possibility of putting and reading values through their ports, processes may communicate with each other asynchronously through an event mechanism. An event name e may be broadcast by a coordinator or an atomic process p to its environment, yielding an *event occurrence* (p, e) for which the broadcasting process p becomes its *event source*. The set of event occurrences is therefore defined by

$$\mathit{EvnOcc} = \mathit{PrcNm} \times \mathit{EvtNm}\,.$$

Once an event name is *broadcast* by a process, the process continues with its processing, while the resulting event occurrence propagates through the environment independently. Any process that is interested in that event occurrence will receive it and can, subsequently, react on it. In our semantic model, the actual broadcasting of event occurrences is the responsibility of the second-level transition system which models an entire **Manifold** system (Section 5).

Once an event occurrence is received by a process, it is stored in its *event memory*, i.e a subset of $EvnOcc$. We denote the set of all event memories by $EvnMem$, with E as its typical element. A coordinator process may react to the event occurrences stored in its event memory. The reaction takes place by evaluating an event pattern. Recall that event patterns appear as guards in commands, so the evaluation of an event pattern influences the choice of which component of a command is chosen. In principle, such an evaluation may or may not be successful. If it succeeds, then it consumes an event occurrence from the event memory. All of this is modeled by the following evaluation function:

$$eval\colon (EvnPat \times EvnMem) \to \mathcal{P}(EvnOcc)\,,$$

defined, for $p \in PrcNm$, $e \in EvtNm$, and $E \in EvnMem$ as follows:

$$\begin{aligned}
eval((0,\mathtt{begin}),E) &= E \cap \{(0,\mathtt{begin})\}\\
eval((p,e),E) &= \{(x,y) \in E \mid x = p \text{ and } y = e\}\\
eval((p,*e),E) &= \{(x,y) \in E \mid x = p\}\\
eval((*p,e),E) &= \{(x,y) \in E \mid y = e\}\\
eval((*p,*e),E) &= E\,.
\end{aligned}$$

This function takes an event pattern as input and evaluates it on the basis of the current event occurrences stored in the event memory E of the process. We say that the evaluation is *successful* if it returns a non-empty set of event occurrences. This evaluation mechanism is important because it permits to dynamically substitute actual parameters for formal ones. The only other way for communicating names is at the moment of process creation (where new names are also created).

In the definition of a transition for coordinator processes, below, we encounter the need for a slightly extended syntax for partially executed programs. To this end, we introduce a set of *resumptions* $(R \in)\ Res_C$, defined by the grammar

$$R ::= \mathbf{stop} \mid \ \mathbf{wait};R \mid\, !G \mid\ S;R \mid\ G;R$$

where G is a command and S is a statement. A resumption indicates the part of the program that is yet to be executed by the coordinator process: **stop** indicates that the activity of the process has terminated, **wait**;R indicates that the process is waiting to be activated before executing the resumption

R, and $!G$ indicates that a new execution of the body G of a program has to start. The other cases correspond to when the process has to execute a statement S, or has to execute a command G, respectively.

Certain syntactically different resumptions will have the same operational meaning. In order to simplify the presentation of the transition system (notably, the number of rules, which are numerous, anyway), we define a congruence relation $\equiv$ on Res_C as the least equivalence relation such that:

$$!G \equiv G;!G \quad \text{and} \quad (G_1 + G_2);R \equiv (G_2 + G_1);R\,.$$

The above relation forces a repeated execution of the command G and the '+' operator to be commutative. The associativity of the '+' operator can be deduced from its meaning given in the rule '(C20) *Command choice*', below.

We describe the behavior of coordinator processes by means of a transition system of the form

$$\langle Conf_C, Obs_C, \longrightarrow_C \rangle,$$

consisting of a set $Conf_C$ of configurations, a set Obs_C of observations, and a transition relation $\longrightarrow_C$. They are defined in detail below.

The set $Conf_C$ of configurations consists of tuples

$$\langle p, P, T, E, R \rangle\,,$$

where $p \in PrcNm$ is the name of the coordinator process, $P \subseteq \texttt{Ports}$ is a set containing the active ports of p, $T \in Unlocks^*$ is a stack containing the unlock information, $E \in EvnMem$ is the event memory, and $R \in Res_C$ is the resumption. As usual, we denote by $\cdot$ the concatenation operation for strings.

The set Obs_C, ranged over by ω, is defined as follows:

$$\begin{aligned}
Obs_C = \{&\tau\} \\
&\cup \{(p, halt) \mid p \in PrcNm\} \\
&\cup \{dscnct(\alpha) \mid \alpha \in PrcNm \times \texttt{Ports}\} \\
&\cup \{(p, receive(eo)) \mid p \in PrcNm, eo \in EvnOcc\} \\
&\cup \{evn(e) \mid e \in EvtNm\} \\
&\cup \{(p, raise(e)) \mid p \in PrcNm, e \in EvtNm\} \\
&\cup \{(p, post(e)) \mid p \in PrcNm, e \in EvtNm\} \\
&\cup \{prc(p{:}\texttt{P}, \vec{n}) \mid p \in PrcNm, \texttt{P} \in \texttt{CType} \cup \texttt{AType}, \vec{n} \in Nm^*\} \\
&\cup \{start(p) \mid p \in PrcNm\} \\
&\cup \{finish(p) \mid p \in PrcNm\} \\
&\cup \{str(s{:}\texttt{S}, \alpha, \beta) \mid s \in StrNm, \texttt{S} \in \texttt{SType}, \alpha, \beta \in PrcNm \times \texttt{Ports}\}
\end{aligned}$$

$\cup \{break(s) \mid s \in StrNm\}$
$\cup \{src(s, \alpha) \mid s \in StrNm, \alpha \in PrcNm \times \mathtt{Ports}\}$
$\cup \{snk(s, \alpha) \mid s \in StrNm, \alpha \in PrcNm \times \mathtt{Ports}\}$
$\cup \{unlock(U) \mid U \in Unlocks\}$.

Intuitively, τ denotes some internal activity; $(p, halt)$ denotes the termination of the process p; $dscnct(p.\mathtt{i})$ is the disconnection of all streams connected to the port $\mathtt{i}$ of the process p before its termination; $(p, receive(eo))$ is the receiving of the event occurrence eo by process p; $evn(e)$ is the declaration of a new event name e; $(p, raise(e))$ is the broadcasting of the event name e by process p; $(p, post(e))$ is the sending of the event name e only to the process p; $prc(p{:}\mathtt{P}, \vec{n})$ denotes the creation of a process of type $\mathtt{P}$; assigning it the name p, and passing it the actual parameters $\vec{n}$, $start(p)$ is the activating of the process p; $finish(p)$ is the deactivating of the process p; $str(s{:}\mathtt{S}, \alpha, \beta)$ is the creation of a stream with name s of type $\mathtt{S}$ from the port α to the port β; $break(s)$ is the breaking of the connections of the stream s; $src(s, \alpha)$ is the reconnection of the source of the stream s to the port α; $snk(s, \alpha)$ is the reconnection of the sink of the stream s to the port α; and $unlock(U)$ is the unlocking of the ports in $dom(U)$.

Next, we define the transition relation $\longrightarrow_C \subseteq Conf_C \times Obs_C \times Conf_C$. It describes the local steps of a coordinator process *in isolation*. Transitions that model the interaction of a process with its environment (for instance, reacting to an event) should be interpreted only as *attempts* to make such steps. Whether or not a specific step is actually possible will depend on whether or not the environment is willing to cooperate, for instance, by providing the event to which the process can react. This actual interaction with the environment is modeled by the second-level transition system for **Manifold**, described in Section 5.

(C1) *Syntactic identity:*

$$\frac{R_1 \equiv R_2 \text{ and } \langle p, P, T, E, R_1\rangle \xrightarrow{\omega}_C \langle p, P', T', E', R_1'\rangle \text{ and } R_1' \equiv R_2'}{\langle p, P, T, E, R_2\rangle \xrightarrow{\omega}_C \langle p, P', T', E', R_2'\rangle}$$

Syntactically different resumptions that are related by the equivalence relation $\equiv$ defined above, have the same operational meaning. This rule allows us to substitute a resumption with an equivalent one in a given configuration.

(C2) *Port disconnection:* If $\mathtt{i} \in P$ then

$$\langle p, P, T, E, \mathbf{halt};R\rangle \xrightarrow{dscnct(p.\mathtt{i})}_C \langle p, P \setminus \{\mathtt{i}\}, \varepsilon, \emptyset, \mathbf{halt};R\rangle$$

Before terminating, a process destroys the connections with all its ports. The stack containing the information about the ports to be unlocked is set to empty, as well as the event memory. This action is repeated until the set P of the ports of the process is empty. Each such action will result, in the transition system for the **Manifold** system, in the disconnection of all streams connected with its corresponding port.

(C3) *Internal process termination:*

$$\langle p, \emptyset, T, E, \mathbf{halt};R\rangle \xrightarrow{(p,halt)}_C \langle p, \emptyset, T, E, \mathbf{stop}\rangle$$

If the process has no ports or it has already destroyed the connections with all its ports, then the execution of the statement **halt** causes the termination of the process activity. This action will result, in the transition system for the MANIFOLD system, in the elimination of the process from the set of active processes.

(C4) *Process start:* If $(\mathtt{0}, \mathtt{begin}) \in E$ then

$$\langle p, P, T, E, \mathbf{wait};R\rangle \xrightarrow{\tau}_C \langle p, P, T, E, R\rangle$$

If the coordinator process p has received the event occurrence $(\mathtt{0}, \mathtt{begin})$ and it is waiting to start its execution, then it may begin its activity. The process is now ready to start the execution of its resumption R. Note that the event occurrence $(\mathtt{0}, \mathtt{begin})$ is not consumed from the event memory E, since it may be necessary for actually starting the execution of the process (see axiom (C19)).

(C5) *External process termination:* If $(\mathtt{0}, \mathtt{die}) \in E$ then

$$\langle p, P, T, E, R\rangle \xrightarrow{\tau}_C \langle p, P, T, E \setminus \{(\mathtt{0}, \mathtt{die})\}, \mathbf{halt};R\rangle$$

When the process p has received the event occurrence $(\mathtt{0}, \mathtt{die})$ then it may cease its activity by executing a **halt** statement.

(C6) *Event receiving:*

$$\langle p, P, T, E, R\rangle \xrightarrow{(p,receive(eo))}_C \langle p, P, T, E \cup \{eo\}, R\rangle$$

An event occurrence is received and, consequently, stored in the event memory of the coordinator process.

(C7) *Event declaration:*

$$\langle p, P, T, E, \mathbf{evn}(e).S;R\rangle \xrightarrow{evn(x)}_C \langle p, P, T, E, S[e/x];R\rangle$$

where $x \in \mathit{EvtNm}$ is an event name that does not occur in the resumption $S;R$ nor in the set E. In this step a new event name x is declared and substituted for e in S. The rule (M4) in the transition system for the MANIFOLD system, subsequently uses the locality rule (M21) to ensure that x is indeed a fresh event name in the whole system.

(C8) *Event broadcasting:*

$$\langle p, P, T, E, \mathbf{raise}(e).S;R\rangle \xrightarrow{(p,raise(e))}_C \langle p, P, T, E, S;R\rangle$$

By executing the statement $\mathbf{raise}(e)$ the process p broadcasts the name e. This will result, in the transition system for the MANIFOLD system, in a broadcast of the event occurrence (p, e) to all coordinator and atomic processes except p itself (see rule '(M5) *Event broadcasting*' in Section 5).

(C9) *Event posting:*

$$\langle p, P, T, E, \mathbf{post}(e).S;R\rangle \xrightarrow{(p,post(e))}_C \langle p, P, T, E, S;R\rangle$$

This step is similar to the rule (C8), above. The difference, visible only in the transition system for the whole MANIFOLD system, is that the event occurrence (p, e) will be sent only to the process p itself; it will not be broadcast to any other process.

(C10) *Process creation:* Let $\mathtt{P} \in \mathtt{CType} \cup \mathtt{AType}$ be a process type. If either $\mathtt{P}(\vec{x}) = \langle G_{\mathtt{P}}, P_{\mathtt{P}} \rangle$ and $G_{\mathtt{P}}[\vec{x}/\vec{m}]$ is a syntactically correct command, or alternatively $\mathtt{P}(\vec{m}) = \langle \Sigma_{\mathtt{P},\vec{m}}, Act_{\mathtt{P},\vec{m}}, \longrightarrow_{\mathtt{P},\vec{m}}, \sigma_{\mathtt{P},\vec{m}}, P_{\mathtt{P},\vec{m}} \rangle$, then

$$\langle p, P, T, E, \mathbf{prc}(q{:}\mathtt{P}, \vec{n}).S;R \rangle \xrightarrow{prc(q'{:}\mathtt{P},\vec{m})}_C \langle p, P, T, E, S[q/q'];R \rangle$$

where q' is a fresh process name in $PrcNm(\mathtt{P})$ that does not occur in the resumption $S;R$ nor in the set E, and $\vec{m}$ is the list $\vec{n}$ where q is replaced by q' and `self` by p. The rule (M9) in the transition system for the Manifold system, subsequently uses the locality rule (M21) to ensure that q' is indeed a fresh process name in the whole system. This step represents the intention of process p to create a new process. Unbeknownst to p, if $\mathtt{P} \in \mathtt{CType}$, then this is an attempt to create a coordinator process; if $\mathtt{P} \in \mathtt{AType}$, then this is an attempt to create an atomic process. A new process name q' for this process is created and substituted for q in S. In this way, the process p will later be able to refer to its child within S, while the new process q' initially knows the names in $\vec{m}$ already in use in the system passed to q' as actual parameters. The syntactic correctness of the command of the newly created process is verified at run-time, because the type P may be known only after a (possible) substitution for a type name [8].

(C11) *Process activation:*

$$\langle p, P, T, E, \mathbf{start}(q).S;R \rangle \xrightarrow{start(q)}_C \langle p, P, T, E, S;R \rangle$$

By executing $\mathbf{start}(q)$, the process p signals that the process q may begin its activity. Locally, the execution of the statement $\mathbf{start}(q)$ has no effect on p.

(C12) *Process deactivation:*

$$\langle p, P, T, E, \mathbf{finish}(q).S;R \rangle \xrightarrow{finish(q)}_C \langle p, P, T, E, S;R \rangle$$

Similar to the preceding rule '(C11) *Process activation*', the execution of $\mathbf{finish}(q)$ signals that the process q is to finish its activity.

(C13) *Stream creation:* Let $\mathtt{S} \in \mathtt{SType}$, $\alpha = q.\mathtt{i}$, $\beta = q'.\mathtt{o}$, $\mathtt{P} = ClassOf(q)$, and $\mathtt{P}' = ClassOf(q')$. If `i` is a port of P and `o` is a port of P′ then

$$\langle p, P, T \cdot U, E, \mathbf{str}(s{:}\mathtt{S}, \alpha, \beta).S;R \rangle \xrightarrow{str(s'{:}\mathtt{S},\alpha,\beta)}_C \langle p, P, T \cdot U', E, S[s/s'];R \rangle$$

where s' is a new stream name not occurring in the resumption $S;R$ nor in the sets E and P, and U', with $dom(U') = dom(U) \cup \{\alpha\}$, is defined by

$$U' = \begin{cases} U[\alpha \mapsto 1] & \text{if } \alpha \notin dom(U) \\ U[\alpha \mapsto u+1] & \text{if } U(\alpha) = u \end{cases}$$

The rule (M13) in the transition system for the Manifold system, subsequently uses the locality rule (M21) to ensure that s' is indeed a fresh stream

[8] In practice, compilers perform such type checking statically, e.g., by using structured names as identifiers that include (parameter- and, in Manifold, also port-) signatures. Although it is straight-forward to use a similar scheme in our formal semantics, we skip such "matters of efficiency" here to avoid distracting non-essential detail.

name in the whole system. In this step, a new stream of type `S` is created, connecting port `i` of process q to port `o` of process q'. The stream receives the new stream name s'. This name is also substituted for s in S for future reference. A new stack is obtained from the old one by pushing the information that port `i` of process q must be locked (once more, if it is already locked). Note that this step is executed only if the port names `i` and `o` are declared in the classes that the processes q and q', respectively, are instances of.

(C14) *Breaking a stream:*

$$\langle p, P, T, E, \mathbf{break}(s).S;R\rangle \xrightarrow{break(s)}_C \langle p, P, T, E, S;R\rangle$$

By executing the statement $\mathbf{break}(s)$, the process p indicates that it wants to sever the break-type connections of the stream s.

(C15) *Source reconnection:* Let $\mathtt{P} = \mathit{ClassOf}(q)$. If `i` is a port of `P` then

$$\langle p, P, T \cdot U, E, \mathbf{src}(s, q.\mathtt{i}).S;R\rangle \xrightarrow{src(s,q.\mathtt{i})}_C \langle p, P, T \cdot U', E, S;R\rangle$$

where the formal definition of U' is the same as under the rule '(C13) *Stream creation*', above. The execution of the statement $\mathbf{src}(s, q.\mathtt{i})$ indicates that the source of the existing stream named s should be connected to the port `i` of process q. The new stack of unlock information is obtained by replacing the top of the old one with the updated information about the port i of process q. The action blocks if the port name `i` is not declared in the class `P` of which q is an instance.

(C16) *Sink reconnection:* Let $\mathtt{P} = \mathit{ClassOf}(q)$. If `o` is a port of `P` then

$$\langle p, P, T, E, \mathbf{snk}(s, q.\mathtt{o}).S;R\rangle \xrightarrow{snk(s,q.\mathtt{o})}_C \langle p, P, T, E, S;R\rangle$$

The execution of the statement $\mathbf{snk}(s, q.\mathtt{o})$ is meant to connect the sink of the existing stream named s to the port `o` of process q. As before, the action blocks if the port name `o` is not declared in the class `P` of which q is an instance.

(C17) *Block entrance:*

$$\frac{\langle p, P, T, E, G;R\rangle \xrightarrow{\tau}_C \langle p, P, T', E', S;R\rangle}{\langle p, P, T, E, \{\!|\ G\ |\!\}.S';R\rangle \xrightarrow{\tau}_C \langle p, P, T', E', S;(S';R)\rangle}$$

A block is entered if one of the event patterns of the guarded command G is positively evaluated. By the axiom '(C19) *Event pattern evaluation*' and the rule '(C20) *Command choice*' it follows that a new empty unlock information is put on the top of the stack. The control remains within the command body of the block until it is completely executed. This rule shows that the block construct is a syntactic realisation in the language of the sequentiality concept captured by the ';' operator in the resumptions. The execution of the statement in the block and its termination are described as those of an ordinary statement in the two rules below.

(C18) *Statement termination:*

$$\langle p, P, T \cdot U, E, \mathbf{end};R\rangle \xrightarrow{unlock(U)}_C \langle p, P, T, E, R\rangle$$

When a block statement is completely executed, then all ports locked during the execution of the statement are unlocked; this is indicated by the action $\mathit{unlock}(U)$. The top of the stack of is popped and the control goes to the resumption R. The actual unlocking takes place in the transition system for the **Manifold** system.

(C19) *Event pattern evaluation:* Let $ep = (pr, ev)$ with $pr = q$ or $pr = *q$, and $ev = e$ or $ev = *e$. If $(q', e') \in eval(ep, E)$ then

$$\langle p, P, T, E, ep?S;R\rangle \xrightarrow{\tau}_C \langle p, P, T \cdot \mathbf{0}, E \setminus \{(q', e')\}, S[q/q'][e/e'];R\rangle$$

If an event pattern matches an event occurrence stored in the event memory, then the event occurrence is removed from the event memory and an empty unlock information is pushed on top of the stack. The control goes to the statement S where the process name q' and the event name e' are substituted for the formal parameters q and e, respectively.

(C20) *Command choice:*

$$\frac{\langle p, P, T, E, G_1;R\rangle \xrightarrow{\tau}_C \langle p, P, T', E', S;R\rangle}{\langle p, P, T, E, (G_1 + G_2);R\rangle \xrightarrow{\tau}_C \langle p, P, T', E', S;R\rangle}$$

The choice between two guarded commands is nondeterministic. Using this rule we can easily prove that if in a configuration we substitute a resumption $(G_1 + (G_2 + G_3));R$ with the resumption $((G_1 + G_2) + G_3);R$ (or vice-versa), then the configurations obtained after one transition are the same.

4.2 Atomic processes

Atomic processes are instances of computation classes. In Section 3, we saw that computation classes are specified by a transition system extended with some extra components. In order to be able to describe the behavior of instances of computation classes *within the* **Manifold** *system*, we adapt these transition systems somewhat, essentially by naming the process, remembering the actual parameters to be substituted for the formal ones, and by adding some information on the status of the process.

More formally, we consider a new transition system

$$\langle Conf_A, Obs_A, \longrightarrow_A\rangle$$

defined as follows. The set $Conf_A$ consists of tuples

$$\langle p, P, \sigma, R\rangle\,,$$

where $p \in PrcNm$ is the name associated with the atomic process; $P \subseteq \mathtt{Ports}$ is a set containing the ports of the process; σ is an internal state of one of the transition systems $\mathtt{A}(\vec{n})$, for some $\mathtt{A} \in \mathtt{AType}$ and list of names $\vec{n}$; and $R \in \{\mathbf{wait}, \mathbf{run}, \mathbf{stop}\}$ is the execution mode of the process (either waiting to be activated, already activated and executing, or finished). Because the set of internal states of computational classes declared with different atomic types or parameters are disjoint, we can retrieve from a configuration the type of the atomic process being executed and the parameters passed at the moment of its creation from the internal state σ.

The set Obs_A of observable actions is defined as follows:

$$
\begin{aligned}
Obs_A = \{\tau\} \\
&\cup \{(p, halt) \mid p \in PrcNm\} \\
&\cup \{dscnct(\alpha) \mid \alpha \in PrcNm \times \texttt{Ports}\} \\
&\cup \{(p, receive(eo)) \mid p \in PrcNm, eo \in EvnOcc\} \\
&\cup \{(p, raise(e)) \mid p \in PrcNm, e \in EvtNm\} \\
&\cup \{(p, get(p.\texttt{i}, v)) \mid p \in PrcNm, \texttt{i} \in \texttt{Ports}, v \in \texttt{Val}\} \\
&\cup \{(p, put(p.\texttt{o}, v)) \mid p \in PrcNm, \texttt{o} \in \texttt{Ports}, v \in \texttt{Val}\}\,.
\end{aligned}
$$

These actions are similar to the actions in $Act_{\texttt{A},\vec{n}}$ of some computation class of type $\texttt{A} \in \texttt{AType}$ with parameter $\vec{n}$, except for the action $dscnct(\alpha)$ which denotes the disconnection of all streams connected with the port α. Furthermore, we now add the identity of the process executing the action everywhere, except for the internal actions τ.

Finally, the new transition relation $\longrightarrow_A \subseteq Conf_A \times Obs_A \times Conf_A$ is given by the following axioms and rules.

(A1) *Internal activity:*

$$\frac{\sigma \xrightarrow{\tau}_{\texttt{A},\vec{n}} \sigma'}{\langle p, P, \sigma, \mathbf{run}\rangle \xrightarrow{\tau}_A \langle p, P, \sigma', \mathbf{run}\rangle}$$

An atomic process embedded into the MANIFOLD system changes its local state by executing an internal action exactly in the same way as it would have done without being embedded in the MANIFOLD system.

(A2) *Port disconnection:* If $R \neq \mathbf{wait}$ and $\texttt{i} \in P$ then

$$\frac{\sigma \xrightarrow{halt}_{\texttt{A},\vec{n}} \sigma'}{\langle p, P, \sigma, R\rangle \xrightarrow{dscnct(p.\texttt{i})}_A \langle p, P \setminus \{\texttt{i}\}, \mathbf{0}, \sigma, \mathbf{stop}\rangle}$$

Before terminating, a process changes its execution mode to **stop** and, without changing state, destroys the connections with all its ports. Because its execution mode is changed to **stop**, no further transitions are possible from this configuration, with the exception of those transitions specified by this rule and the next one. In the transition system for the MANIFOLD system, all streams connected with the port `i` will be disconnected.

(A3) *Process termination:* If $R \neq \mathbf{wait}$ then

$$\frac{\sigma \xrightarrow{halt}_{\texttt{A},\vec{n}} \sigma'}{\langle p, \emptyset, \sigma, R\rangle \xrightarrow{(p,halt)}_A \langle p, \emptyset, \sigma', \mathbf{stop}\rangle}$$

If all connection with the ports of the process are destroyed, the execution of the action halt is completed, and hence the atomic process p terminates. No further transitions are possible from this new configuration.

(A4) *Process activation:*

$$\langle p, P, \sigma, \textbf{wait}\rangle \xrightarrow{(p, receive((\mathtt{0}, \mathtt{begin})))}_A \langle p, P, \sigma, \textbf{run}\rangle$$

If the event `begin` is received from the 'system' `0` and the process is waiting to be activated, then it may start its execution. In this case, its execution mode **wait** is changed to **run**.

(A5) *Receiving events (I):* Let $q \in \mathit{PrcNm}$. If $R \neq \textbf{stop}$ then

$$\frac{\sigma \xrightarrow{receive(e)}_{\mathtt{A},\vec{n}} \sigma'}{\langle p, P, \sigma, R\rangle \xrightarrow{(p, receive((q,e)))}_A \langle p, P, \sigma', R\rangle}$$

where $R = \textbf{wait}$ implies $eo \neq (\mathtt{0}, \mathtt{begin})$. The process q is arbitrary and merely represents an *assumption* about the identity of the sender of the event e, which at this stage is unknown. An atomic process is willing to receive an event in which it is interested any time after its creation, even when it is waiting to be activated. However, receiving events cannot change the execution mode of an atomic process. The only exception is given in the rule '(A4) *Process activation*'.

(A6) *Receiving events (II):* Let $R \neq \textbf{stop}$ and $\sigma \in \Sigma_{\mathtt{A},\vec{n}}$. If $e \notin \vec{n}$ then for all $q \in \mathit{PrcNm}$

$$\langle p, P, \sigma, R\rangle \xrightarrow{(p, receive((q,e)))}_{\langle} p, P, \sigma, R\rangle$$

where $R = \textbf{wait}$ implies $q \neq \mathtt{0}$ or $e \neq \mathtt{begin}$. If an event is received and the process is not interested in it, then nothing happens. As before, the only exception is when the process has not yet started its activity and it receives for the first time the event `begin` from `0`.

Rules (A4) and (A5) together with the axiom (A6) imply that an atomic process that is in an execution mode other than **stop**, is always event-enabled. This can be explained as follows. From the moment of its creation, a process is part of the system, and hence able to receive event occurrences. These may be stored for a later reaction, after the start of its execution, causing a change of 'initial' state of the atomic process. Dually, when a process finishes its activity, it is eliminated from the system as soon as all stream connections to its ports are broken up. Since termination is irreversible, when its execution mode is **stop** it no longer makes sense for the atomic process to be event-enabled.

(A7) *Raising events:*

$$\frac{\sigma \xrightarrow{raise(e)}_{\mathtt{A},\vec{n}} \sigma'}{\langle p, P, \sigma, \textbf{run}\rangle \xrightarrow{(p, raise(e))}_A \langle p, P, \sigma', \textbf{run}\rangle}$$

Raising an event name e causes the broadcast of the event occurrence (p, e) to all processes except itself.

(A8) *Getting values:*

$$\frac{\sigma \xrightarrow{get(\mathtt{i},v)}_{\mathtt{A},\vec{n}} \sigma'}{\langle p, P, \sigma, \textbf{run}\rangle \xrightarrow{(p, get(p.\mathtt{i},v))}_A \langle p, P, \sigma', \textbf{run}\rangle}$$

A value v is read by the process p from its port `i`, assuming that it is available.

(A9) *Putting values:*

$$\frac{\sigma \xrightarrow{put(\mathtt{o},v)}_{\mathtt{A},\vec{n}} \sigma'}{\langle p, P, \sigma, \mathbf{run}\rangle \xrightarrow{(p,put(p.\mathtt{o},v))}_{A} \langle p, P, \sigma', \mathbf{run}\rangle}$$

A value v is put by the process p on its port $\mathtt{o}$. In the second-level transition system for the MANIFOLD system, this value is replicated, if necessary, and inserted into all (outgoing) streams connected to this port.

4.3 Streams

Next we define a transition system for streams. Streams are active entities in charge of transporting values between ports of processes in the MANIFOLD system. Values flow in a stream from its source to its sink. The source and the sink of a stream are elements of the set

$$StrCn = (PrcNm \times \mathtt{Ports}) \cup \{\perp\}.$$

Here $\perp$ denotes no connection (with $\perp \notin Nm$), while a pair in $PrcNm \times \mathtt{Ports}$ denotes a connection with a port of a process.

We describe the behavior of a stream by a transition system

$$\langle Conf_S, Obs_S, \longrightarrow_S\rangle ,$$

consisting of a set $Conf_S$ of configurations, Obs_S of observations, and a transition relation $\longrightarrow_S$. Streams are not programmable: their transitions depend only on their type and their content.

The set $Conf_S$ of configurations for a stream consists of tuples

$$\langle s, \mathtt{S}, \alpha, \beta, w\rangle ,$$

where $s \in StrNm$ is the name of the stream, $\mathtt{S} \in \mathtt{SType}$ is the type of the stream, $\alpha, \beta \in StrCn$ are the source and the sink of the stream, respectively, and $w \in \mathtt{Val}^*$ is the list of values still to be delivered to its sink.

The set Obs_S is defined as follows:

$$\begin{aligned} Obs_S = \ & \{(s, get(\alpha, v)) \mid s \in StrNm, \alpha \in PrcNm \times \mathtt{Ports}, v \in \mathtt{Val}\} \\ \cup \ & \{(s, put(\alpha, v)) \mid s \in StrNm, \alpha \in PrcNm \times \mathtt{Ports}, v \in \mathtt{Val}\} \\ \cup \ & \{(s, break(\alpha, \beta)) \mid s \in StrNm, \alpha, \beta \in StrCn\} \\ \cup \ & \{(s, src(\alpha, \beta)) \mid s \in StrNm, \alpha \in StrCn, \beta \in PrcNm \times \mathtt{Ports}\} \\ \cup \ & \{(s, snk(\alpha, \beta)) \mid s \in StrNm, \alpha \in StrCn, \beta \in PrcNm \times \mathtt{Ports}\} \\ \cup \ & \{(s, dscnct(\alpha, \beta)) \mid s \in StrNm, \alpha \in PrcNm \times \mathtt{Ports}, \beta \in StrCn\}. \end{aligned}$$

Intuitively, $(s, get(\alpha, v))$ denotes the stream s getting the value v from the port α; $(s, put(\beta, v))$ is the delivering of the value v to the port β by the stream s; $(s, break(\alpha, \beta))$ is the breaking of the connections α and β of the stream s; $(s, src(\alpha, \beta))$ is the reconnection of the source α of the stream s to the new source β; $(s, snk(\alpha, \beta))$ is the reconnection of the sink α of the stream s to the new sink β; and $(s, dscnct(\alpha, \beta))$ is the disconnection of the stream s from α and, perhaps, β, due to the termination of the process that owns the port α.

The relation $\longrightarrow_S$ is defined as the least relation in $Conf_S \times Obs_S \times Conf_S$ satisfying the following axioms and rules.

(S1) *Get value:*

$$\langle s, \mathtt{S}, \alpha, \beta, w\rangle \xrightarrow{(s,get(\alpha,v))}_S \langle s, \mathtt{S}, \alpha, \beta, v \cdot w\rangle$$

At any stage, a value v can be inserted in the queue of values of a stream s, assuming that it is available from the port connected at the source of the stream.

(S2) *Put value:*

$$\langle s, \mathtt{S}, \alpha, \beta, w \cdot v\rangle \xrightarrow{(s,put(\beta,v))}_S \langle s, \mathtt{S}, \alpha, \beta, w\rangle$$

A stream s can always remove a value v from its queue of values to be delivered, and make this value available to the port connected at its sink.

(S3) *Breakup (I):* If $\mathtt{S} \in \{\mathtt{KK}, \mathtt{BK}\}$ then

$$\langle s, \mathtt{S}, \alpha, \beta, w\rangle \xrightarrow{(s,break(\alpha',\perp))}_S \langle s, \mathtt{S}, \alpha'', \beta, w\rangle$$

where if $\mathtt{S} = \mathtt{KK}$ then $\alpha' = \perp$ and $\alpha'' = \alpha$, otherwise $\alpha' = \alpha$ and $\alpha'' = \perp$. The break-type (source and/or sink) connections of a stream s may be broken up at any time. This rule primarily deals with all cases where the sink cannot be disconnected. The source is disconnected only if it is of break-type. We will see in Section 5 that if $\alpha' \neq \perp$, the port manager associated with this port updates its information.

(S4) *Breakup (II):* If $\mathtt{S} \in \{\mathtt{BB}, \mathtt{KB}\}$ then

$$\langle s, \mathtt{S}, \alpha, \beta, w\rangle \xrightarrow{(s,break(\alpha',\beta))}_S \langle s, \mathtt{S}, \alpha'', \perp, w\rangle$$

where if $\mathtt{S} = \mathtt{KB}$ then $\alpha' = \perp$ and $\alpha'' = \alpha$, otherwise $\alpha' = \alpha$ and $\alpha'' = \perp$. This rule deals with the cases where the sink can be disconnected. The source is disconnected only if it is of break-type.

(S5) *Source reconnection:*

$$\langle s, \mathtt{S}, \alpha, \beta, w\rangle \xrightarrow{(s,src(\alpha,\alpha'))}_S \langle s, \mathtt{S}, \alpha', \beta, w\rangle$$

where $\alpha' \in StrCn$ is arbitrary and merely represents an assumption about the new connection at the source of the stream s. In any configuration, a stream s can disconnect its source (regardless of its type) and reconnect it to a new port. In the transition system for the **Manifold** system, the port managers associated with the processes connected at the source of the stream before and after this transition step will update their information about their affected ports.

(S6) *Sink reconnection:*
$\langle s, \mathtt{S}, \alpha, \beta, w\rangle \xrightarrow{(s,snk(\beta,\beta'))}_S \langle s, \mathtt{S}, \alpha, \beta', w\rangle$
where $\beta' \in \mathit{StrCn}$ and, as before, it represents an assumption about the new connection at the sink of the stream s.

(S7) *Source disconnection (I):* Let $\alpha \neq \bot$. If $\mathtt{S} \in \{\mathtt{BB}, \mathtt{KB}\}$ then
$\langle s, \mathtt{S}, \alpha, \beta, w\rangle \xrightarrow{(s,dscnct(\alpha,\beta))}_S \langle s, \mathtt{S}, \bot, \bot, w\rangle$
The stream s disconnects from its source if the process that owns this port terminates its activity. If the connection to the sink is of break-type, then it is also disconnected.

(S8) *Source disconnection (II):* Let $\alpha \neq \bot$. If $\mathtt{S} \in \{\mathtt{BK}, \mathtt{KK}\}$ then
$\langle s, \mathtt{S}, \alpha, \beta, w\rangle \xrightarrow{(s,dscnct(\alpha,\bot))}_S \langle s, \mathtt{S}, \bot, \beta, w\rangle$
The stream s disconnects from its source because the process that owns this port terminates its activity. The sink is not disconnected because it is of keep-type.

(S9) *Sink disconnection (I):* Let $\beta \neq \bot$. If $\mathtt{S} \in \{\mathtt{BB}, \mathtt{BK}\}$ then
$\langle s, \mathtt{S}, \alpha, \beta, w\rangle \xrightarrow{(s,dscnct(\beta,\alpha))}_S \langle s, \mathtt{S}, \bot, \bot, w\rangle$
A stream disconnects from its sink if the process that owns this port terminates its activity. If the connection to the source is of break-type, then it is also disconnected.

(S10) *Sink disconnection (II):* Let $\beta \neq \bot$. If $\mathtt{S} \in \{\mathtt{KB}, \mathtt{KK}\}$ then
$\langle s, \mathtt{S}, \alpha, \beta, w\rangle \xrightarrow{(s,dscnct(\beta,\bot))}_S \langle s, \mathtt{S}, \alpha, \bot, w\rangle$
A stream disconnects from its sink because the process that owns this port terminates its activity. The source is not disconnected because it is of keep-type.

5 Semantics of Manifold: second level

In Section 4, we defined a collection of transition systems that specify the behavior of coordinator processes, atomic processes, and streams, as autonomous entities that can compute and interact with their environment. This collection defines the first-level transition system

$$\langle \mathit{Conf}_1, \mathit{Obs}_1, \longrightarrow_1\rangle,$$

where $\mathit{Conf}_1 = \mathit{Conf}_C \cup \mathit{Conf}_A \cup \mathit{Conf}_S$ is the set of all configurations of coordinator processes, atomic processes, and streams; $\mathit{Obs}_1 = \mathit{Obs}_C \cup \mathit{Obs}_A \cup \mathit{Obs}_S$ is the set of all observable actions of coordinator processes, atomic processes, and streams; and $\longrightarrow_1$ is the least relation in $\mathit{Conf}_1 \times \mathit{Obs}_1 \times \mathit{Conf}_1$ including $\longrightarrow_C$, $\longrightarrow_A$, and $\longrightarrow_S$.

In this section, we define another transition system that describes the behavior of an entire **Manifold** system. In this new transition system, the interaction between different processes and streams will be modeled by combining different transition steps of the first-level systems into a single second-level transition step.

The activity of an entire **Manifold** system is modeled using configurations consisting of three components: the active processes, a list for each active atomic or coordinator process of the pending messages that have already been broadcast but not yet received by that process, and a port manager for each atomic or coordinator process. These concepts are defined below.

Active processes

A **Manifold** system starts its execution with an instance of a coordinator process. This coordinator process may create and activate new instances of atomic processes, streams, and/or coordinator processes. The latter may in turn create new instances of processes, and so on. To describe the configurations of each process active in a **Manifold** system, we define a set $APrc$ of collections of *active processes* as the set of all finite subsets of $Conf_1$.

We consider only sets of active processes $A \in APrc$ with different names. Therefore, we need a function $Id\colon APrc \to \mathcal{P}(PrcNm \cup StrNm)$ that, for each set $A \in APrc$ of active processes, returns the set of process or stream names of the configurations in A:

$$\begin{aligned} Id(A) = &\ \{p \mid \langle p, P, T, E, R\rangle \in A \cap Conf_C\} \\ &\cup \{p \mid \langle p, P, \sigma, R\rangle \in A \cap Conf_A\} \\ &\cup \{s \mid \langle s, \mathtt{S}, \alpha, \beta, w\rangle \in A \cap Conf_S\}\,. \end{aligned}$$

Given a set of active processes $A \in APrc$, we say that a *process, stream or event name n occurs in A* if either $n \in Id(A)$ or there exists $C \in A$ such that

- $C = \langle p, P, T, E, R\rangle \in Conf_C$ and n occurs in E or R;
- $C = \langle p, P, \sigma, R\rangle \in Conf_A$ and $n \in \vec{m}$ for $\sigma \in \Sigma_{\mathtt{A},\vec{m}}$ (recall here that an intenal state of an atomic process may belong to only one computational class of type $\mathtt{A}$ with actual parameter $\vec{m}$).

Lists of pending messages

We already saw in the Section 4 that active processes of the **Manifold** system may generate and receive event occurrences. When an event occurrence is generated, it is broadcast to all other active atomic and coordinator processes which, eventually, will receive it. To model this broadcast mechanism, we associate with each configuration of the **Manifold** system, a sequence of pending event occurrences for every process. These represent the event occurrences that have already been broadcast but not yet received by that process. Recall that a broadcast event occurrence may be received by different processes at different moments in time. They will be received one at a time when a process performs a '*receive*' transition, whereby the received event occurrence is deleted from the list of pending event occurrences of that process.

In **Manifold**, the reception of event occurrences respects the order in which they were broadcast by their source processes only. Thus, two event occurrences broadcast by different source processes may be received by a process in any order, while two event occurrences broadcast by the same source process are received by any process that observes them both, in the same order as they were produced. We model this aspect of the semantics of **Manifold** by considering Mazurkiewicz traces [35] of pending event occurrences, rather than sequences of them.

First, we define a binary relation $\approx$ on $EvnOcc$ that relates two event occurrences with the same process as their common source:

$$eo \approx eo' \quad \text{if and only if} \quad eo = (p, e) \text{ and } eo' = (p, e') \,.$$

The relation $\approx$ is reflexive, commutative, and also transitive.

Let now $\mathcal{F}(EvnOcc, \approx)$ be the partially commutative monoid obtained by considering all strings in $EvnOcc^*$ modulo the least congruence (with respect to string concatenation) such that, for all eo and eo' in $EvnOcc$,

$$eo \cdot eo' = eo' \cdot eo \qquad \text{if } eo \not\approx eo' \,.$$

Elements of $\mathcal{F}(EvnOcc, \approx)$, typically denoted by t, are called traces [35]. We write $\odot$ for the string concatenation $\cdot$ modulo the above congruence, and $\widetilde{eo}$ for the congruence class containing the one-element string $eo \in EvnOcc$. The congruence class containing the empty string is denoted by ε.

In other words, we consider lists of pending event occurrences such that occurrences with different source processes may commute. In each configuration, we associate such a list to every atomic or coordinator process. Formally, we consider partial functions M in

$$Msg = PrcNm \rightharpoonup \mathcal{F}(EvnOcc, \approx) \,.$$

The partiality of functions in Msg is necessary because the set of active processes can dynamically change.

Port managers

Every port has a port manager that regulates the flow of values into or out of the process that owns the port. Specifically, a port manager associates with its port a natural number l and two finite sets of stream names, I and O. The number l indicates how many times the port has been locked, while the two sets I and O contain the names of the streams connected to the port at their sink or source, respectively. Following is a more detailed description of the relevance of these components.

If $l = 0$ then the port is said to be *unlocked* and values can flow through the port. However, if $l > 0$ then the port is *locked* and values cannot flow through the port. We see below that every time a new stream is created or the source of a stream is reconnected, its source port is locked, i.e., its port manager increments the l value of the port by 1. The l value of a port is decremented by 1 or more when the transition system of a coordinator process (that has already locked the port) executes the action '$unlock(U)$'. The increment and decrement of the l value of a port is formally defined in the transition system below. It is important to note that the '*unlock*' action is not programmable and takes place every time a coordinator process completes the actions within a block (see the rule '(C18) *Statement termination*' in Section 4.1).

A port manager of a port α stores in the set I the names of the streams whose common sink is α, and in the set O the names of the streams whose common source is α. This information is dynamically updated, for example, every time a new stream is created. The information in O is used, for example, every time a value flows through the port out of the process, enabling the port manager to deliver a copy of the value to each stream in O.

A port manager cannot be considered an autonomous active process but rather a store of information available for each process. However, the execution of a process is independent of the activity of its port managers. For example, a process may be executing some internal activity, while one of its port managers updates the information about its port.

Formally, within a configuration of the MANIFOLD system, we model a collection of port managers by a partial function Pt in

$$PrtMng = (PrcNm \times \mathtt{Ports}) \rightharpoonup (\mathbb{N} \times \mathcal{P}(StrNm) \times \mathcal{P}(StrNm))\,.$$

The partiality of a function Pt is necessary because the set of active processes (and hence of their ports) can dynamically change. If $Pt(\alpha)$ is defined, then we refer to it as the port manager associated with the port α. In this case we say that its left component is the lock-value of α and its other two components are the sets of streams with the port α as their common sink (I) and source (O), respectively.

A transition system for a MANIFOLD *system*

In this section we define a transition system to model the behavior of a MANIFOLD system. The basic scheme is as follows. Given a collection of configurations of active processes (either coordinator processes, atomic processes, or streams), we define a transition for a MANIFOLD system whenever one or more of these processes make a transition at the first level. Some of these transitions may need to be synchronized with others before they can actually take place, resulting in a coordinated activity. Also, we must take care of the broadcasting of event occurrences to all processes, and manage the information stored in the port managers.

We describe the behavior of a MANIFOLD system through a transition system

$$\langle \mathit{Conf}_2, \mathit{Obs}_2, \longrightarrow_2 \rangle,$$

consisting of a set Conf_2 of configurations, a set Obs_2 of observables, and a transition relation $\longrightarrow_2$. Each of these components is defined in detail, below.

The set Conf_2 of configurations consists of tuples

$$\langle A, M, \mathit{Pt} \rangle\ ,$$

where $A \in \mathit{APrc}$ is a set of active processes, $M \in \mathit{Msg}$ is a partial function for the lists of pending messages, and $\mathit{Pt} \in \mathit{PrtMng}$ is a partial function for the port managers.

We say that a configuration $\langle A_1, M_1, \mathit{Pt}_1 \rangle$ is *disjoint* from another configuration $\langle A_2, M_2, \mathit{Pt}_2 \rangle$ if $\mathit{Id}(A_1) \cap \mathit{Id}(A_2) = \emptyset$, $\mathit{dom}(M_1) \cap \mathit{dom}(M_2) = \emptyset$ and $\mathit{dom}(\mathit{Pt}_1) \cap \mathit{dom}(\mathit{Pt}_2) = \emptyset$.

Given a designated coordinator type $\mathtt{C} \in \mathtt{CType}$ with its associated declaration $\mathtt{C}(\varepsilon) = \langle G, P \rangle$, a MANIFOLD system starts its execution from the configuration

$$\langle \{\langle p, P, \mathbf{0}, \emptyset, !G \rangle\}, \mathbf{0}[p \mapsto (\mathtt{0}, \mathtt{begin})], \mathit{Pt} \rangle$$

where p is a new process name not occurring in G, and $\mathit{Pt}(p.\mathtt{i}) = \langle 0, \emptyset, \emptyset \rangle$ for all $\mathtt{i} \in P$ and it is undefined otherwise. In other words, the MANIFOLD system starts its execution by activating an instance of the class with type $\mathtt{C}$, which receives a new name p. Because the execution of a coordinator process is triggered by event occurrences, the system sends to p a starting event occurrence $(\mathtt{0}, \mathtt{begin})$. The locality rule (M21) in the transition system for the MANIFOLD system, ensures that p indeed remains a globally unique process name in the whole system.

The set Obs_2 is defined as follows:

$$Obs_2 = \{\tau\} \cup EvnOcc \cup PrcNm \cup StrNm \cup EvtNm \,.$$

Intuitively, τ denotes some internal activity, $eo \in EvnOcc$ is the broadcasting of an event occurrence, and $n \in PrcNm \cup StrNm \cup EvtNm$ denotes a new name created by some process.

The transition relation $\longrightarrow_2$ is defined as the least relation in $Conf_2 \times Obs_2 \times Conf_2$ satisfying the axioms and rules introduced below. We introduce each transition step, below, by conditions on the minimal set of resources necessary for that step to take place. Then, we can embed these resources into a broader context, using the locality rules '(M19)-(M21)', defined below. In this way, our definition of the transition relation $\longrightarrow_2$ reflects the true time and space decoupling of a **Manifold** system.

(M1) *Internal activity:*

$$\frac{C \xrightarrow{\tau}_1 C'}{\langle\{C\}, \mathbf{0}, \mathbf{0}\rangle \xrightarrow{\tau}_2 \langle\{C'\}, \mathbf{0}, \mathbf{0}\rangle}$$

A single process may execute an internal action, about which no further knowledge is necessary. Since the action is executed locally by the process, it is not visible to the environment and no resource from the environment is required by the process.

(M2) *Process halting:* Let $dom(M) = \{p\}$.

$$\frac{C \xrightarrow{(p,halt)}_1 C'}{\langle\{C\}, M, \mathbf{0}\rangle \xrightarrow{(p,\mathtt{stop})}_2 \langle\emptyset, \mathbf{0}, \mathbf{0}\rangle}$$

When a process terminates its execution, it ceases to be an active process and broadcasts an event occurrence denoting its termination. Every other active process will eventually receive this event occurrence, thus learning that the process p has terminated its activity.

(M3) *Port disconnection:* Let $dom(Pt) = \{\alpha\} \cup \{f(s) \mid s \in IO, f(s) \neq \perp\}$, where $Pt(\alpha) = \langle l, I, O\rangle$, $IO = I \cup O$, and $f\colon IO \to StrCn$ is defined below. We have

$$\frac{C \xrightarrow{dscnct(\alpha)}_1 C' \text{ and } C_s \xrightarrow{(s,dscnct(\alpha,\beta_s))}_1 C'_s \text{ for all } s \in IO}{\langle\{C\} \cup \{C_s \mid s \in IO\}, \mathbf{0}, Pt\rangle \xrightarrow{\tau}_2 \langle\{C'\} \cup \{C'_s \mid s \in IO\}, \mathbf{0}, Pt'\rangle}$$

where $f(s) = \beta_s$ if $C_s \xrightarrow{(s,dscnct(\alpha,\beta_s))}_1 C'_s$, and, for $x \in dom(Pt') = dom(Pt) \setminus \{\alpha\}$,

$$Pt'(x) = \langle l', I' \setminus f^{-1}(x), O' \setminus f^{-1}(x)\rangle \quad \text{if } Pt(x) = \langle l', I', O'\rangle \,.$$

When a process disconnects a port, every stream whose source or sink is connected to this port must break its connection, regardless of its type. Depending on its type, the other end of each such stream may also be broken. This is why all port managers in $f(IO)$ are involved in this step. Note that the port manager of the port that is disconnected in this rule is destroyed.

(M4) *Event declaration:*

$$\frac{C \xrightarrow{evn(e)}_1 C'}{\langle\{C\}, \mathbf{0}, \mathbf{0}\rangle \xrightarrow{e}_2 \langle\{C'\}, \mathbf{0}, \mathbf{0}\rangle}.$$

A new event name e is declared by a process in the configuration C. The label of the above transition is used in the locality rule (M21) to ensure that e is a fresh event name in the whole system.

(M5) *Event broadcasting:* If $dom(M) = \{p\}$ then

$$\frac{C \xrightarrow{(p,raise(e))}_1 C'}{\langle\{C\}, M, \mathbf{0}\rangle \xrightarrow{(p,e)}_2 \langle\{C'\}, M, \mathbf{0}\rangle}.$$

The process p broadcasts an event occurrence to all active processes. As explained in the rule '(M20) *Locality (II)*' the requirement that the function M is defined only for p implies that this event occurrence will not be sent to the process p itself.

(M6) *Event posting:*

$$\frac{C \xrightarrow{(p,post(e))}_1 C'}{\langle\{C\}, \mathbf{0}[p \mapsto t], \mathbf{0}\rangle\} \xrightarrow{\tau}_2 \langle\{C'\}, \mathbf{0}[p \mapsto t \odot \widetilde{(p, e)}], \mathbf{0}\rangle}$$

The event occurrence (p, e) is sent only to its source process p. This operation is asynchronous, and hence, the event occurrence need not be received immediately by the process.

(M7) *Input synchronization:* Let $dom(Pt) = \{\alpha\}$. If $Pt(\alpha) = \langle l, I, O\rangle$ and $s \in I$ then

$$\frac{C_1 \xrightarrow{(s,put(\alpha,v))}_1 C_1' \text{ and } C_2 \xrightarrow{(p,get(\alpha,v))}_1 C_2'}{\langle\{C_1, C_2\}, \mathbf{0}, Pt\rangle \xrightarrow{\tau}_2 \langle\{C_1', C_2'\}, \mathbf{0}, Pt\rangle}$$

A value flows through the port α from a stream s into a process p. The condition $s \in I$ guarantees that s is a stream, but p can be an atomic process or a stream (not necessarily distinct from s). Note the nondeterminism caused by the possibility of having several streams with their respective sinks connected to the same port.

(M8) *Output synchronization:* Let $dom(Pt) = \{\alpha\}$. If $Pt(\alpha) = \langle 0, I, O\rangle$ and $O \neq \emptyset$ then

$$\frac{C \xrightarrow{(n,put(\alpha,v))}_1 C' \text{ and } C_s \xrightarrow{(s,get(\alpha,v))}_1 C_s' \text{ for all } s \in O}{\langle\{C\} \cup \{C_s \mid s \in O\}, \mathbf{0}, Pt\rangle \xrightarrow{\tau}_2 \langle\{C'\} \cup \{C_s' \mid s \in O\}, \mathbf{0}, Pt\rangle}$$

A process or a stream offers to put a value v to the set of streams that share the unlocked port α as their common source. The name n may refer to either the process that owns the port α itself, or one of the streams in I, i.e., one of the streams whose sinks are connected to the port α. In the first case, n is an atomic process (because coordinator processes do not produce values) and the value flows in a manner that is the dual of the one described in the previous rule: from an atomic process to one or more streams connected to one of its ports. In the second case, the process that owns the port α may be an atomic or a coordinator process, and its port is used only to let values flow through from some streams to some other streams.

(M9) *Coordinator process creation:* If $\mathtt{C} \in \mathtt{CType}$ and $\mathtt{C}(\vec{x}) = \langle G, P \rangle$ then

$$\frac{C \xrightarrow{prc(p:\mathtt{C},\vec{n})}_1 C'}{\langle \{C\}, \mathbf{0}, \mathbf{0} \rangle \xrightarrow{p}_2 \langle \{C', \langle p, P, \varepsilon, \emptyset, \mathbf{wait};! G[\vec{x}/\vec{n}][self/p] \rangle\}, \mathbf{0}[p \mapsto \varepsilon], Pt \rangle}$$

where $Pt(p.\mathtt{i}) = \langle 0, \emptyset, \emptyset \rangle$ for all $\mathtt{i} \in P$ and it is undefined otherwise. In this step, a process creates a new coordinator process. An initial configuration for the new coordinator process is created, and assigned to the name p. The stack for unlock information of p is initially empty, and so is its event memory. The resumption of p indicates that before proceeding with the execution of the command G, it must first wait until it is activated (rule '(M11) *Process activation*', below). At the moment of its birth, a new process has no pending messages to be received. An initial port manager is also created for every port of the process.

The label of the above transition indicates that p is a new name created during this step and is used in the locality rule (M21) to ensure that it is a fresh name in the whole **Manifold** system.

(M10) *Atomic process creation:* If $\mathtt{A} \in \mathtt{AType}$ and $\mathtt{A}(\vec{n}) = \langle \Sigma, Act, \longrightarrow, \sigma, P \rangle$ then

$$\frac{C \xrightarrow{prc(p:\mathtt{A},\vec{n})}_1 C'}{\langle \{C\}, \mathbf{0}, \mathbf{0} \rangle \xrightarrow{p}_2 \langle \{C', \langle p, P, \sigma, \mathbf{wait} \rangle\}, \mathbf{0}[p \mapsto \varepsilon], Pt \rangle}$$

where $Pt(p.\mathtt{i}) = \langle 0, \emptyset, \emptyset \rangle$ if $\mathtt{i} \in P$ and it is undefined otherwise. This step is similar to the previous rule (M9), except that an atomic process is created instead of a coordinator process. The initial configuration for a new atomic process consists of its name p, an initial binding of its formal parameters to the actual ones, and an initial state. Before starting its execution, the process must receive the event '`begin`' from the system '`0`'.

As in the previous rule, the new process has no pending messages to be received and an initial port manager is created for each of its ports.

(M11) *Process activation:*

$$\frac{C \xrightarrow{start(p)}_1 C'}{\langle \{C\}, \mathbf{0}[p \mapsto t], \mathbf{0} \rangle \xrightarrow{\tau}_2 \langle \{C'\}, \mathbf{0}[p \mapsto t \odot \widetilde{(\mathtt{0}, \mathtt{begin})}], \mathbf{0} \rangle}$$

The event name `begin` is sent to the process p to announce that it can start its execution.

(M12) *Process deactivation:*

$$\frac{C \xrightarrow{finish(p)}_1 C'}{\langle \{C\}, \mathbf{0}[p \mapsto t], \mathbf{0} \rangle \xrightarrow{\tau}_2 \langle \{C'\}, \mathbf{0}[p \mapsto t \odot \widetilde{(\mathtt{0}, \mathtt{die})}], \mathbf{0} \rangle}$$

This rule is the dual of the previous rule (M11): to terminate the execution of a process, the event name `die` must be sent to it.

(M13) *Stream creation:* If $dom(Pt) = \{\alpha, \beta\}$ then

$$\frac{C \xrightarrow{str(s:\mathtt{S},\alpha,\beta)}_1 C'}{\langle \{C\}, \mathbf{0}, Pt \rangle \xrightarrow{s}_2 \langle \{C', \langle s, \mathtt{S}, \alpha, \beta, \varepsilon \rangle\}, \mathbf{0}, Pt' \rangle}$$

where, for $x \in \{\alpha, \beta\}$,

$$Pt'(x) = \begin{cases} \langle l+1, I, O \cup \{s\}\rangle & \text{if } x = \alpha \text{ and } Pt(\alpha) = \langle l, I, O\rangle \\ \langle l, I \cup \{s\}, O\rangle & \text{if } x = \beta \text{ and } Pt(\beta) = \langle l, I, O\rangle\,. \end{cases}$$

A new stream is created with name s. Its source is connected to the port α and its sink to the port β. Initially, the new stream has no value to deliver. The port managers of the two ports connected at the ends of the stream update their information. The label of the above transition is used in the locality rule (M21) to ensure that s is a fresh stream name in the whole system.

(M14) *Unlocking ports:* If $dom(Pt) = dom(U)$ then

$$\frac{C \xrightarrow{unlock(U)}_1 C'}{\langle\{C\}, \mathbf{0}, Pt\rangle \xrightarrow{\tau}_2 \langle\{C'\}, \mathbf{0}, Pt'\rangle}$$

where $dom(Pt') = dom(Pt)$ and $Pt'(\alpha) = \langle l - U(\alpha), I, O\rangle$ if $Pt(\alpha) = \langle l, I, O\rangle$. We saw in the previous rule (M13) how the lock-value of a port is incremented by its port manager. This rule describes the inverse operation that takes place when a coordinator process executes an 'unlock' action, typically, when a statement is completely executed and the process exits from a block construct. The function $U \in Unlocks$ contains the information about the number of times each port must be unlocked. Hence, every port manager associated with a port in U must update its information.

During an execution of a **Manifold** system, the lock-value of each port manager is always greater than or equal to 0. This is because it is decremented only if it has previously been incremented, for example, by rule (M13) or (M17).

(M15) *Stream breakup:* Let $dom(Pt) = \{\alpha, \beta\} \setminus \{\perp\}$.

$$\frac{C_1 \xrightarrow{break(s)}_1 C_1' \text{ and } C_2 \xrightarrow{(s, break(\alpha,\beta))}_1 C_2'}{\langle\{C_1, C_2\}, \mathbf{0}, Pt\rangle \xrightarrow{\tau}_2 \langle\{C_1', C_2'\}, \mathbf{0}, Pt'\rangle}$$

where, for $x \in dom(Pt') = dom(Pt)$,

$$Pt'(x) = \begin{cases} \langle l, I, O \setminus \{s\}\rangle & \text{if } x = \alpha \text{ and } Pt(\alpha) = \langle l, I, O\rangle \\ \langle l, I \setminus \{s\}, O\rangle & \text{if } x = \beta \text{ and } Pt(\beta) = \langle l, I, O\rangle \end{cases}$$

In Section 4.3, we saw that if a stream is connected at both ends with a break-type connection, then its connections break through this operation. Hence, the port managers of the ports at the two ends must update their information. Note that we update the information about a port only when there is a real connected port (i.e., not $\perp$) and the connection is not keep-type. For example, if both ends of the stream are either disconnected or have a keep-type connection then $dom(Pt) = \emptyset$ and hence no port manager involvement is necessary.

(M16) *Sink reconnection:* Let $dom(Pt) = \{\alpha, \beta\} \setminus \{\perp\}$.

$$\frac{C_1 \xrightarrow{snk(s,\beta)}_1 C_1' \text{ and } C_2 \xrightarrow{(s, snk(\alpha,\beta))}_1 C_2'}{\langle\{C_1, C_2\}, \mathbf{0}, Pt\rangle \xrightarrow{\tau}_2 \langle\{C_1', C_2'\}, \mathbf{0}, Pt'\rangle}$$

where, for $x \in dom(Pt') = dom(Pt)$,

$$Pt'(x) = \begin{cases} \langle l, I \setminus \{s\}, O\rangle & \text{if } x = \alpha \text{ and } Pt(\alpha) = \langle l, I, O\rangle \\ \langle l, I \cup \{s\}, O\rangle & \text{if } x = \beta \text{ and } Pt(\beta) = \langle l, I, O\rangle\,. \end{cases}$$

In this step, the stream s is disconnected from its current sink and is subsequently reconnected to the port β instead. Note that if the stream is disconnected at its sink, i.e. $\alpha = \perp$, then only one port manager is involved in this step.

(M16) *Source reconnection:* Let $dom(Pt) = \{\alpha, \beta\} \setminus \{\perp\}$.

$$\frac{C_1 \xrightarrow{src(\alpha,\beta)}_1 C_1' \text{ and } C_2 \xrightarrow{(s,src(\alpha,\beta))}_1 C_2'}{\langle\{C_1, C_2\}, \mathbf{0}, Pt\rangle \xrightarrow{\tau}_2 \langle\{C_1', C_2'\}, \mathbf{0}, Pt'\rangle}$$

where, for $x \in \{\alpha, \beta\}$,

$$Pt'(x) = \begin{cases} \langle l, I, O \setminus \{s\}\rangle & \text{if } x = \alpha \text{ and } Pt(\alpha) = \langle l, I, O\rangle \\ \langle l+1, I, O \cup \{s\}\rangle & \text{if } x = \beta \text{ and } Pt(\beta) = \langle l, I, O\rangle\,. \end{cases}$$

This step is similar to the previous one, except that the source of a stream is reconnected instead of its sink. This is why the lock number of the new port connected at the source is incremented.

(M18) *Receiving event occurrences:*

$$\frac{C \xrightarrow{(p,receive(eo))}_1 C'}{\langle\{C\}, \mathbf{0}[p \mapsto \widetilde{eo} \odot t], \mathbf{0}\rangle \xrightarrow{\tau}_2 \langle\{C'\}, \mathbf{0}[p \mapsto t], \mathbf{0}\rangle}$$

An event occurrence that is in the list of pending messages associated with a process can be received by this process at any stage by performing a '*receive*' transition. Once received, the event occurrence is removed from the list of pending messages of the process.

Recall that because we use traces as lists of pending messages, event occurrences with different process sources may be received in any order, and event occurrences broadcast by the same source process are received in the same order as they were broadcast.

(M19) *Locality (I):* Let $\langle A_0, M_0, Pt_0\rangle, \langle A_1, M_1, Pt_1\rangle$ and $\langle A_2, M_2, Pt_2\rangle$ be three configurations in *Conf*$_2$. If both $\langle A_0, M_0, Pt_0\rangle$ and $\langle A_1, M_1, Pt_1\rangle$ are disjoint from $\langle A_2, M_2, Pt_2\rangle$ then

$$\frac{\langle A_0, M_0, Pt_0\rangle \xrightarrow{\tau}_2 \langle A_1, M_1, Pt_1\rangle}{\langle A_0 \cup A_2, M_0 \oplus M_2, Pt_0 \oplus Pt_2\rangle \xrightarrow{\tau}_2 \langle A_1 \cup A_2, M_1 \oplus M_2, Pt_1 \oplus Pt_2\rangle}$$

where $\oplus$ is the union of partial functions (defined in Appendix A). This rule and the next two, below, reflect the decoupling of the independent activities of processes in a system: each active process can independently and locally take one step, possibly broadcasting event occurrences to other processes. The disjointness conditions ensure that a new name is not used by two different processes.

(M20) *Locality (II):* Let $\langle A_0, M_0, Pt_0\rangle, \langle A_1, M_1, Pt_1\rangle$ and $\langle A_2, M_2, Pt_2\rangle$ be three configurations in *Conf*$_2$ such that both $\langle A_0, M_0, Pt_0\rangle$ and $\langle A_1, M_1, Pt_1\rangle$ are disjoint from $\langle A_2, M_2, Pt_2\rangle$. If $eo \in EvnOcc$ then

$$\frac{\langle A_0, M_0, Pt_0\rangle \xrightarrow{eo}_2 \langle A_1, M_1, Pt_1\rangle}{\langle A_0 \cup A_2, M_0 \oplus M_2, Pt_0 \oplus Pt_2\rangle \xrightarrow{eo}_2 \langle A_1 \cup A_2, M, Pt_1 \oplus Pt_2\rangle}$$

where

$$M(p) = \begin{cases} M_1(p) & \text{if } p \in dom(M_1) \\ M_2(p) \odot \widetilde{eo} & \text{if } p \in dom(M_2)\,. \end{cases}$$

Names broadcast by processes in $dom(M_1)$ are added to the list of pending messages of every process in $dom(M_2)$.

(M21) *Locality (III):* Let $\langle A_0, M_0, Pt_0\rangle, \langle A_1, M_1, Pt_1\rangle$ and $\langle A_2, M_2, Pt_2\rangle$ be three configurations in $Conf_2$ such that both $\langle A_0, M_0, Pt_0\rangle$ and $\langle A_1, M_1, Pt_1\rangle$ are disjoint from $\langle A_2, M_2, Pt_2\rangle$. If the (process, stream, or event) name n does not occur in A_2 then

$$\frac{\langle A_0, M_0, Pt_0\rangle \xrightarrow{n}_2 \langle A_1, M_1, Pt_1\rangle}{\langle A_0 \cup A_2, M_0 \oplus M_2, Pt_0 \oplus Pt_2\rangle \xrightarrow{n}_2 \langle A_1 \cup A_2, M_1 \oplus M_2, Pt_1 \oplus Pt_2\rangle}$$

This rule guarantees that the new name n created by some process in A_0 is also new for processes in A_2.

6 Conclusions and further work

Manifold is a pure coordination language that abstracts computation away as internal, unobservable details inside atomic processes. As such, it is a good model and language for the formal study of the core concepts and issues involved in interactive systems (as in [42]) and coordination programming. In this regard, **Manifold** is especially interesting as an example of a control-oriented coordination language. The fact that **Manifold** has a practical, real implementation and has already been used in a number of applications, also makes the formal study of this language necessary as well as interesting.

The separation of computation and coordination concerns that is inherent in **Manifold** is the basis for the two-level transition system model we use for its formal semantics in this paper. We believe such two-level models are useful not only for the formal study of control-oriented coordination languages such as **Manifold**, but more generally, for all coordination languages. Such an approach separates the concerns for the proper behavior of each individual component process in a system, from the concerns for the proper behavior of the system as a whole. The first class of concerns are addressed in the first level of the semantic model, assuming each component process is embedded in an ideal supportive environment. The second class of concerns are addressed in the second level of the semantic model where the collective behavior of the system emerges only from the interactions among its components, whose individual behavior mutually engage and constrain that of their peers.

One of the important reasons for the study of the formal semantics of the MANIFOLD language is proving properties of programs written in this language. This requires a precise definition of the 'observables' produced by the second-level transition rules presented in this paper. The formal definition of these observables may depend on the specific investigation of the properties and proofs they are to be used for. For instance, one may or may not wish to include a local time-stamp in each observable, which will then collectively imply a virtual global clock and reflect a partial order on all observables. With or without such time-stamps, however, the two level semantic model we use for MANIFOLD already dictates much of the essence of the observables that can be produced by our second-level transition system: these observables can only reflect such activities as creation and death of processes and streams, breakup and (re)connection of streams and ports, production, flow, and consumption of values through ports and streams, broadcast and reception of events, etc. A trace of such observables produced as the outcome of our formal semantics for a MANIFOLD program is thus conceptually indistinguishable from an actual trace of the execution of the MANIFOLD program by the MANIFOLD run-time system.

The set of all possible permutations of the trace of a MANIFOLD program that are permissible under the partial order of its observables, then, corresponds to the set of all possible executions of that MANIFOLD program on any platform. This makes the traces produced by formal semantics useful for the study of the behavior of actual programs. Our experience with writing real life MANIFOLD programs indicates that a majority of the bugs we have observed in MANIFOLD program modules are of the type that could have been detected by an investigation of their individual behavior in isolation. Most such bugs could have been revealed through asking reachability questions: which guarded commands are reachable from a given guarded command? From which other guarded commands can a given guarded command be reached? In practice, reachability questions can also reveal many of the important aspects of the interaction behavior of the component processes that comprise a MANIFOLD application: can certain states (e.g., representing deadlocks) ever be reached? The formal semantics presented in this paper can be used as the basis for the development of tools for visual debugging, as well as analysis and (semi)automatic verification of the coordination protocols of concurrent programs. This is part of our on-going work on building a visual programming environment for MANIFOLD, called Visifold [13].

The semantic model presented in this paper already has a great deal of modularity: the second-level transition system is defined essentially as a composition of the first-level transition systems. The practical significance of compositionality is that it lends itself to better modular software design. A target for further research is the design of other fully modular compositional semantic models for MANIFOLD, to support analysis and proof techniques for reasoning about the properties of programs. For instance, a denotational semantics for MANIFOLD may be derivable from the operational semantics we present in this paper following the approach of [1]. There a three-level denotational semantics is given for a distributed object-oriented language: a first level for statements, one for objects in isolation, and a third one for the whole system of objects running in parallel.

On a more theoretical level, the work presented in this paper is being used as a basis for the development of a more abstract calculus for coordination. Furthermore, the insight we gained through the work reported in this paper indicates that coalgebraic models [32,40] seem very appealing as a mathematically sound foundation for the semantics of MANIFOLD. Such mathematical models can also benefit other coordination models and languages with similar constructs, or those that can in turn be modeled by constructs analogous to our semantic model for MANIFOLD. As our formal semantics presented in this paper demonstrates, MANIFOLD's strict separation of computation from communication, plus the fact that it is based on an exogenous model of coordination, leads to a clear dichotomy of internal vs. externally observable behavior of each process. This, in turn, corresponds directly with the inherent 'strict information hiding' property of coalgebras. On the other hand, coalgebraic models for the semantics of MANIFOLD raise interesting challenges in the field of coalgebras: to reflect the compositionality of MANIFOLD, a suitable theory of composition of coalgebras is necessary.

Acknowledgment

We are thankful to the former and present members of the MANIFOLD group at CWI for their contributions towards the development, implementation, and application of this language, through which process many of the abstract notions presented in this paper were congealed and tested. We are also grateful for the comments of Frank de Boer and the Amsterdam Coordination Group on earlier versions of this paper. We are indebted to the anonymous referees, who have written extensive and detailed reports, suggesting several ways of improving our presentation.

References

[1] P. America and J.J.M.M. Rutten. A layered semantics for a parallel object-oriented language. *Formal aspects of computing*, 4:376–408, 1992.

[2] J.-M. Andreoli, P. Ciancarini, and R. Pareschi. Interaction Abstract Machines. In *Trends in Object-Based Concurrent Computing*, pages 257–280. MIT Press, 1993.

[3] F. Arbab. Coordination of massively concurrent activities. Technical Report CS-R9565, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1995. Available on-line at `http://www.cwi.nl/ftp/CWIreports/IS/CS-R9565.ps.Z`.

[4] F. Arbab. The IWIM model for coordination of concurrent activities. In [18], pages 34–56.

[5] F. Arbab. Manifold version 2: Language reference manual. Technical report, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1996. Available on-line at `http://www.cwi.nl/ftp/manifold/refman.ps.Z`.

[6] F. Arbab. What do you mean, coordination? *Bulletin of the Dutch Association for Theoretical Computer Science, NVTI*, pages 11–22, 1998. Available on-line at `http://www.cwi.nl/NVTI/Nieuwsbrief/nieuwsbrief.html`.

[7] F. Arbab, C.L. Blom, F.J. Burger, and C.T.H. Everaars. Reusable coordinator modules for massively concurrent applications. *Software: Practice and Experience*, 28(7):703–735, 1998. Extended version.

[8] F. Arbab, P. Ciancarini, and C. Hankin. Coordination languages for parallel programming. In *Parallel Computing*, 24(7):989–1004, special issue on Coordination Languages for Parallel Programming, July 1998.

[9] F. Arbab, I. Herman, and P. Spilling. An overview of Manifold and its implementation. *Concurrency: Practice and Experience*, 5(1):23–70, 1993.

[10] F. Arbab and E. Monfroy. Using coordination for cooperative constraint solving. In *Proceedings of the 1998 ACM Symposium on Applied Computing; Special Track on Coordination Models, Languages and Applications*, pages 139–148. ACM press, 1998.

[11] M.M. Bonsangue, J.N. Kok, M. Boasson, and E. de Jong. A software architecture for distributed control systems and its transition system semantics. In *Proceedings of the 1998 ACM Symposium on Applied Computing; Special Track on Coordination Models, Languages and Applications*, pages 159–168. ACM press, 1998.

[12] M.M. Bonsangue, J.N. Kok, and G. Zavattaro. Comparing coordination models based on shared distributed replicated data. In *Proceedings of the 1999 ACM Symposium on Applied Computing; Special Track on Coordination Models, Languages and Applications*, ACM press, 1999.

[13] P. Bouvry and F. Arbab. Visifold: A visual environment for a coordination language. In [18], pages 403–406.

[14] N. Busi, R. Gorrieri, and G. Zavattaro. Three semantics of the output operation for generative communication. In [28], pages 205–219.

[15] N. Busi, R. Gorrieri, and G. Zavattaro. A process algebraic view of Linda coordination primitives. *Theoretical Computer Science*, 192(2):167–199, 1998.

[16] P. Butcher. A behavioral semantics for Linda-2. *IEEE Software Engineering Journal*, 4(6):196–204, July 1991.

[17] P. Ciancarini, R. Gorrieri, and G. Zavattaro. An alternative semantics for the parallel operator of the calculus of Gamma programs. In *Coordination Programming: Mechanisms, Models and Semantics*, pages 232–248. Imperial College Press, 1996.

[18] P. Ciancarini and C. Hankin, editors. *Coordination Languages and Models, Proceedings of the 1st international conference COORDINATION'96*, volume 1061 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[19] P. Ciancarini, K. Jensen, and D. Yankelevich. On the operational semantics of a coordination language. In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[20] R. De Nicola and R. Pugliese. An observational semantics for Linda. In *Structures in Concurrency Theory*, Workshops in Computing, pages 129–143. Springer-Verlag, 1995.

[21] R. De Nicola and R. Pugliese. A process algebra based on Linda. In [18], pages 160–178.

[22] J.-Y. V. Dury, L. Bellissard, and V. Marangozov. A component calculus for modelling the Olan configuration language. In [28], pages 392–409.

[23] C.T.H. Everaars and F. Arbab. Coordination of distributed/parallel multiple-grid domain decomposition. In A. Ferreira, J. Rolim, Y. Saad, and T. Yang, editors, *Proceedings of Irregular'96*, volume 1117 of *Lecture Notes in Computer Science*, pages 131–144. Springer-Verlag, 1996.

[24] C.T.H. Everaars, F. Arbab, and F.J. Burger. Restructuring sequential Fortran code into a parallel/distributed application. In *Proceedings of the international conference on Software Maintenance '96*, pages 13–22. IEEE, November 1996.

[25] C.T.H. Everaars and B. Koren. Using coordination to parallelize sparse-grid methods for 3D CFD problems. In *Parallel Computing*, 24(7):1081–1106, special issue on Coordination Languages for Parallel Programming, 1998.

[26] C.T.H. Everaars and B. Lisser. Coordination of a Parallel Proposition Solver. *Proceedings of the 3rd international conference on coordination, COORDINATION '99*, volume ??? of *Lecture Notes in Computer Science*. Springer-Verlag, April 1999.

[27] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming.* Prentice-Hall, 1990.

[28] D. Garlan and D. Le Métayer, editors. *Coordination Languages and Models, Proceedings of the 2nd international conference COORDINATION'97*, volume 1282 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[29] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[30] C.L. Hankin, D. Le Métayer, and D. Sands. A calculus of Gamma programs. In *Proceedings of the 5th workshop on Languages and Compilers for Parallel Computing*, volume 757 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[31] P. Inverardi, A.L. Wolf, and D. Yankelevich. Checking assumption in component dynamics at the architectural level. In [28], pages 46–63.

[32] B. Jacobs and J.J.M.M. Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of EATCS*, 62:222–259, 1997. Available on-line at `http://www.cs.kun.nl /~bart/PAPERS/JR.ps.Z`.

[33] J.-M. Jacquet and K. De Bosschere. On the semantics of *μlog*. *Future Generation Computer Systems*, 10(1):93–136, 1994.

[34] J. Kramer. Configuration programming – a framework for the development of distributable systems. In *Proceedings of the IEEE international conference on Computer Systems and Software Engineering (CompuEuro '90)*, pages 374–384, 1990.

[35] A. Mazurkiewicz. Concurrent program schemes and their interpretations. Technical Report DAIMI 113-78, Aarhus University, 1977.

[36] G.D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[37] G. Papadopoulos and F. Arbab. Control-based coordination of human and other activities in cooperative information systems. In [28], pages 422–425.

[38] G.A. Papadopoulos and F. Arbab. Modelling activities in information systems using the coordination language Manifold. In *Proceedings of the 1998 ACM Symposium on Applied Computing; Special Track on Coordination Models, Languages and Applications*, pages 185–193. ACM press, 1998.

[39] E.P.B.M. Rutten, F. Arbab, and I. Herman. Formal specification of Manifold: a preliminary study. Technical Report CS–R9215, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1992.

[40] J.J.M.M. Rutten. Universal coalgebra: A theory of systems. Technical Report CS-R9652, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1996. Available on-line at `http://www.cwi.nl/ftp/CWIreports /AP/CS-R9652.ps.Z`.

[41] F. Seredynski, P. Bouvry, and F. Arbab. Parallel and distributed evolutionary computation with Manifold. In V. Malyshkin, editor, *Proceedings of PaCT '97*, volume 1277 of *Lecture Notes in Computer Science*, pages 94–108. Springer-Verlag, 1997.

[42] P. Wegner. Interactive foundations of computing. *Theoretical Computer Science*, 192(2):315–351, 20, 1998.

A Appendix: basic notation

In this appendix we give the basic notations for partial functions that we used in the paper.

For a partial function $f\colon X \rightharpoonup Y$ we denote by $dom(f)$ the subset of X on which f is defined. The partial function with an empty domain is denoted by $\mathbf{0}$. As usual, application of a function to a set is done element-wise, and application to a list is done component-wise.

For any partial function $f\colon X \rightharpoonup Y$, $x \in X$ and $y \in Y$, we use the notation $f[x \mapsto y]$ to denote the function mapping x to y and otherwise acting as f. Since the domain of f may or may not contain x we have that $dom(f[x \mapsto y]) = dom(f) \cup \{x\}$. For $\vec{x} = x_1 \cdots x_n$ and $\vec{y} = y_1 \cdots y_n$ we denote by $f[\vec{x} \mapsto \vec{y}]$ the function $((f[x_1 \mapsto y_1])[x_2 \mapsto y_2]) \cdots [x_n \mapsto y_n]$.

If $f\colon X_1 \rightharpoonup Y_1$ and $g\colon X_2 \rightharpoonup Y_2$ are two partial functions defined on disjoint domains then we denote by $f \oplus g\colon (X_1 \cup X_2) \rightharpoonup (Y_1 \cup Y_2)$ the partial function defined by:

$$(f \oplus g)(x) = \begin{cases} f(x) & \text{if } x \in dom(f) \\ g(x) & \text{if } x \in dom(g)\,. \end{cases}$$

Clearly, $f \oplus g = g \oplus f$. Note that $f \oplus \mathbf{0} = \mathbf{0} \oplus f = f$ for every $f\colon X \rightharpoonup Y$.