What Do You Mean, Coordination?*

Farhad Arbab
CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
email: Farhad.Arbab@cwi.nl

Abstract

Coordination models and languages represent a new approach to design and development of concurrent systems. 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. The field is relatively new, and while many coordination models and languages form a tight cluster of very similar variants, some others are drastically different and they appear to have nothing in common with each other. All this makes it difficult for the uninitiated to discern the underlying similarities of various approaches to coordination. This paper is an "easy reader" introduction to coordination models and languages, their common aims and purpose, their relevance, and their place in the computing arena. The work on coordination at CWI is presented here as a specific example.

1 Introduction

The size, speed, capacity, and price of computers have all dramatically changed in the last half century. Still more dramatic are the subtle changes of the society's perception of what computers are and what they can, should, and are expected to do. Clearly, this change of perception would not have been possible without the technological advances that reduced the size and price of computers, while increasing their speed and capacity. Nevertheless, the social impact of this change of perception and its feedback influence on the advancement of computer technology itself, are too significant to be regarded as mere by-products of those technological advances.

The term "computer" today has a very different meaning than it did in the early part of this century. Even after such novelties as mechanical and electrical

*This article appeared in the March '98 issue of the Bulletin of the Dutch Association for Theoretical Computer Science (NVTI), which is available on-line at http://www.cwi.nl/NVTI/Nieuwsbrief/nieuwsbrief98.ps.gz.

calculators had become commonplace, book-keeping, in its various forms, was a time consuming and labor intensive endeavor for businesses and government agencies alike. Analogous to "typist pools" that lingered on until much later, enterprises such as accountant firms and insurance companies employed armies of people to process, record, and extract the large volumes of essentially numerical data that were relevant for their business. Since "computer" was the term that designated these people, the machine that could clearly magnify their effectiveness and held the promise of replacing them altogether, became known as the "electronic computer."

In spite of the fact that from the beginning, symbol manipulation was as much an inherent ability of electronic computers as arithmetic and juggling of numbers, the perception that computers are really tools for performing fast numerical computations was prevalent. Problems that did not involve a respectable amount of number crunching were either rejected outright as non-problems, or were considered as problems not worthy of attempts to apply computers and computing to. Subscribers to such views were not all naive outsiders, many an insider considered such areas as business and management, databases, and graphics, to be not only on the fringes of computer applications, but also on the fringes of legitimacy. As late as 1970, J. E. Thornton, vice president of Advanced Design Laboratory of Control Data Corporation, who was personally responsible for most of the detailed design of the landmark CDC 6600 computer system, wrote[38]:

> There is, of course, a class of problems which is essentially *noncomputational* but which requires a massive and sophisticated storage system. Such uses as inventory control, production control, and the general category of information retrieval would qualify. *Frankly, these do not need a computer.* There are, however, legitimate justifications for a large computer system as a "partner" with the computational usage. [*emphasis* added.]

Of course, by that time many people were not only convinced that legitimate *computational* applications need not involve heavy number crunching, but were already actively working to bring about the changes that turned fringe activities such as databases and graphics into the core of computing, both as "science" as well as in the domain of applications. Nevertheless, Thornton's statement at the time represented the views of a non-negligible minority that has only gradually diminished since. While the numerical applications of computing have steadily grown in number, size, and significance, its non-numerical applications have simply grown even faster and vaster.

2 Computing

The formal notions of computing and computability were introduced by Church, in terms of λ-calculus, and Turing, in terms of Turing Machines. Both Church and Turing were inspired by Hilbert's challenge to define a solid foundation for (mechanical) methods of finding mathematical truth. Hilbert's program consisted of finding a set of axioms as the unassailable foundation of mathematics, such that only mathematical truths could be derived from them by the application of any (truth preserving) mechanical operation, and that all mathematical truths could be derived that way. But, what exactly is a *mechanical operation*? This was what Church, Turing, and others were to define. Turing himself also intended for his abstract machine to formalize the workings of the human mind. Ironically, his own reasoning on the famous halting problem can be used to show that Turing Machines cannot find all mathematical truths, let alone model the workings of the human mind. Gödel's incompleteness theorem, which brought the premature end of Hilbert's program for mathematics, clearly shows the limits of formal systems and mechanical truth derivation methods. Interestingly, Gödel's incompleteness theorem and Turing's halting problem turned out to be equivalent in their essence: they both show that there are (even mathematical) truths that cannot be derived mechanically, and in both cases, the crucial step in the proof is a variation of the diagonalization first used by Cantor to show that the infinity of real numbers between any two numbers is greater than the infinity of natural numbers.

It is far from obvious why Turing's simple abstract machine, or Church's λ-calculus, is a reasonable formalization of what we intuitively mean by *any mechanical operation*. However, all extensions of the Turing Machine that have been considered, are shown to be mathematically equivalent to, and no more powerful than, the basic Turing Machine. Turing and Church showed the equivalence of Turing Machines and λ-calculus. This, plus the fact that other formalizations (e.g., Post's) have all turned out to be equivalent, has increased the credibility of the conjecture that a Turing Machine can actually be made to perform any mechanical operation whatsoever. Indeed, it has become reasonable to mathematically *define* a mechanical operation as any operation that can be performed by a Turing Machine, and to accept the view known as Church's thesis: that the notion of Turing Machines (or λ-calculus, or other equivalents) mathematically defines the concept of an algorithm (or an effective, or recursive, or mechanical procedure).

3 Interaction

Church's thesis can simply be considered as a mathematical definition of what computing is in a strictly technical sense. Real computers, on the other hand, do much more than mere computing in this restrictive sense. Among other things,

they are sources of heat and noise, and have always been revered (and despised) as (dis)tasteful architectural artifacts, pieces of furniture, or decoration mantlepieces. More interestingly, computers also *interact*: they can act as facilitators, mediators, and coordinators that enable the collaboration of other agents. These other agents may in turn be other computers (or computer programs), sensors and actuators that involve their real world environment, or human beings. The role of a computer as an agent that performs computing, in the strict technical sense of the word, should not be confused with its role as a mediator agent that, e.g., empowers its human users to collaborate with one another. The fact that the computer, in this case, may perform some computation in order to enable the collaboration of other agents, is ancillary to the fact that it needs to interact with these agents to enable their collaboration. To emphasize this distinction, Wegner proposes the concept of an Interaction Machine[39, 40].

A Turing Machine operates as a closed system: it receives its input tape, starts computing, and (hopefully) halts, at which point its output tape contains the result of its computation. In every step of a computation, the symbol written by a Turing Machine on its tape depends only on its internal state and the current symbol it reads from the tape. An Interaction Machine is an extension of a Turing Machine that can interact with its environment with new input and output primitive actions. Unlike other extensions of the Turing Machine (such as more tapes, more controls, etc.) this one actually changes the essence of the behavior of the machine. This extension makes Interaction Machines *open systems*.

Consider an Interaction Machine I operating in an environment described as a dynamical system E. The symbol that I writes on its tape at a given step, not only depends on its internal state and the current symbol it reads from the tape, but can also depend on the input it obtains directly from E. Because the behavior of E cannot be described by a computable function, I cannot be replaced by a Turing Machine. The best approximation of I by a Turing Machine, T, would require an encoding of the actual input that I obtains from E, which can be known only *after* the input operation. The computation that T performs, in this case, is the same as that of I, but I does more than T because it interacts with its environment E. What T does, in a sense, is analogous to predicting yesterday's weather: it is interesting that it can be done (assuming that it can be done), but it doesn't quite pass muster! To emphasize the distinction, we can imagine that the interaction of I with E is not limited to just this one direct input: suppose I also does a direct output to E, followed by another direct input from E. Now, because the value of the second input from E to I depends on the earlier interaction of E and I, no input tape can encode this "computation" for any Turing Machine.

It is the ability of computers (as Interaction Machines) to interact with the real world, rather than their ability (as mere Turing Machines) to carry on ever-more-sophisticated computations, that is having the most dramatic impact on our societies. In the traditional models of human-computer interaction, users

prepare and consume the information needed and produced by their applications, or select from the alternatives allowed by a rigid structure of computation. In contrast to these models, the emerging models of human-computer interaction remove the barriers between users and their applications. The role of a user is no longer limited to that of an observer or an operator: increasingly, users become active components of their running applications, where they examine, alter, and steer on-going computations. This form of cooperation between humans and computers, and between humans via computers, is a vital necessity in many contemporary applications, where realistic results can be achieved only if human intuition and common-sense is combined with raw, formal reasoning and computation. The applications of computer facilitated collaborative work are among the increasingly important areas of activity in the foreseeable future. They can be regarded as natural extensions of systems where several users simultaneously examine, alter, interact, and steer on-going computations. Interaction Machines are suitable conceptual models for describing such applications.

Interaction Machines have unpredictable input from their external environment, and can directly affect their environment, unpredictably, due to such input. Because of this property, Interaction Machines may seem too open for formal studies: the unpredictable way that the environment can affect their behavior can make their behavior underspecified, or even ill-defined. But, this view is misleading. Interaction Machines are both useful and interesting for formal studies.

On the one hand, the open-ness of Interaction Machines and their consequent underspecified behavior is a valuable true-to-life property. Real systems are composed of components that interact with one another, where each is an open system in isolation. Typically, the behavior of each of these components is ill-defined, except within the confines of a set of constraints on its interactions with its environment. When a number of such open systems come together as components to comprise a larger system, the topology of their interactions forms a context that constrains their mutual interactions and yields well-defined behavior.

On the other hand, the concept of Interaction Machines suggests a clear dichotomy for the formal study of their behavior, both as components in a larger system, as well as in isolation. Just like a Turing Machine, the behavior of an Interaction Machine can be studied as a computation (in the sense of Church's thesis) between each pair of its successive interactions. More interestingly, one can abstract away from all such computations, regarding them as internal details of individual components, and embark on a formal study of the constraints, contexts, and conditions on the interactions among the components in a system (as well as between the system and its environment) that ensure and preserve well-behavedness. And this material is the thread that weaves the fabric of coordination.

4 Concurrency

Interaction and concurrency are closely related concepts. Concurrency means that computations in a system overlap in time. The computations in a concurrent system may actually run in parallel (i.e., use more than one physical processor at a time) or be interleaved with one another on a single processor. The parallel computations in a system may or may not be geographically distributed. What distinguishes an interactive system from other concurrent systems is the fact that an interactive system has unpredictable inputs from an external environment that it does not control.

The study and the application of concurrency in computer science has a long history. The study of deadlocks, the dining philosophers, and the definition of semaphores and monitors were all well established by the early seventies. Theoretical work on concurrency, e.g., CSP[22, 23], CCS[28], process algebra[12], and π-calculus[29, 30], has helped to show the difficulty of dealing with concurrency, especially when the number of concurrent activities becomes large. Most of these models are more effective for describing closed systems. A number of programming languages have used some of these theoretical models as their bases, e.g., Occam[24] uses CSP and LOTOS[13] uses CCS. However, it is illuminating to note that the original context for the interest in concurrency was somewhat different than the demands of the applications of today in two respects:

- In the early days of computing, hardware resources were prohibitively expensive and had to be shared among several programs that had nothing to do with each other, except for the fact that they were unlucky enough to have to compete with each other for a share of the same resources. This was the *concurrency of competition*. Today, it is quite feasible to allocate tens, hundreds, and thousands of processors to the same task (if only we could do it right). This is the *concurrency of cooperation*. The distinction is that whereas it is sufficient to keep independent competing entities from trampling on each other over shared resources, cooperating entities also depend on the (partial) results they produce for each other. Proper passing and sharing of these results require more complex protocols, which become even more complex as the number of cooperating entities and the degree of their cooperation increase.

- The falling costs of processor and communication hardware only recently dropped below the threshold where having very large numbers of "active entities" in an application makes sense. Massively parallel systems with thousands of processors are a reality today. Current trends in processor hardware and operating system kernel support for threads[1] make it possible to efficiently have in the order of hundreds of active entities running

[1] Threads are preemptively-scheduled light-weight processes that run within one operating-system level process and share the same address space.

in a process on each processor. Thus, it is not unrealistic to think that a single application can be composed of hundreds of thousands of active entities. Compared to classical uses of concurrency, this is a jump of several orders of magnitude in numbers, and in our view, represents (the need for) a qualitative change.

The practical use of massively parallel systems is often limited to applications that fall into one of a few simple concurrency structures. In fact, although MIMD[2] systems have been available for some time, they are hardly ever used as MIMD systems in an application. The basic problem in using the MIMD paradigm in large applications is coordination: how to ensure proper communication among the hundreds and thousands of different pieces of code that comprise the active entities in a single application. A restriction of the MIMD model, called SPMD[3], introduces a *barrier* mechanism as the only coordination construct. This model simplifies the problem of concurrency control by allowing several processors, all executing the same program, but on different data, proceed at their own pace up to a common barrier, where they then synchronize.

There are applications that do not fit in the uniformity offered by the SPMD model and require more flexible coordination. Examples include computations involving large dynamic trees, symbolic computation on parallel machines, and dynamic pipelines. Taking full advantage of the potential offered by massively parallel systems in these and other applications requires massive, non-replicated, concurrency.

The primary concern in the design of a concurrent application must be its model of cooperation: how the various active entities comprising the application are to cooperate with each other. Eventually, a set of communication primitives must be used to realize whatever model of cooperation application designers opt for, and the concerns for performance may indirectly affect their design. Nevertheless, it is important to realize that the conceptual gap between the system supported communication primitives and a concurrent application must often be filled with a non-trivial model of cooperation.

The models of cooperation used in concurrent applications of today are essentially a set of ad hoc templates that have been found to be useful in practice. There is no paradigm wherein we can systematically talk about cooperation of active entities, and wherein we can compose cooperation scenarios such as (and as alternatives to) models like client-server, workers pool, etc., out of a set of primitives and structuring constructs. Consequently, programmers must directly deal with the lower-level communication primitives that comprise the realization of the cooperation model of a concurrent application.

[2]Multiple Instruction, Multiple Data
[3]Single Program, Multiple Data

5 Coordination

Coordination languages, models, and systems constitute a new field of study in programming and software systems, with the goal of finding solutions to the problem of managing the interaction among concurrent programs. Coordination can be defined as the study of the dynamic topologies of interactions among Interaction Machines, and the construction of protocols to realize such topologies that ensure well-behavedness. Analogous to the way in which topology abstracts away the metric details of geometry and focuses on the invariant properties of (seemingly very different) shapes, coordination abstracts away the details of computation in Interaction Machines, and focuses on the invariant properties of (seemingly very different) programs. As such, coordination focuses on *program patterns* that specifically deal with interaction.

Coordination is relevant in design, development, debugging, maintenance, and reuse of all concurrent systems. Coordination models and languages are meant to close the conceptual gap between the cooperation model of an application and the lower-level communication model used in its implementation. The inability to deal with the cooperation model of a concurrent application in an explicit form contributes to the difficulty of developing working concurrent applications that contain large numbers of active entities with non-trivial cooperation protocols. In spite of the fact that the implementation of a complex protocol is often the most difficult and error prone part of an application development effort, the end result is typically not recognized as a "commodity" in its own right, because the protocol is only implicit in the behavior of the rest of the concurrent software. This makes maintenance and modification of the cooperation protocols of concurrent applications much more difficult than necessary, and their reuse next to impossible.

A number of software platforms and libraries are presently popular for easing the development of concurrent applications. Such systems, e.g., PVM, MPI, CORBA, etc., are sometimes called *middleware*. Coordination languages can be thought of as the linguistic counterpart of these platforms which offer middleware support for software composition. One of the best known coordination languages is Linda[16, 27], which is based on the notion of a shared tuple space. The tuple space of Linda is a centrally managed space which contains all pieces of information that processes want to communicate. Linda processes can be written in any language augmented with Linda primitives. There are only four primitives provided by Linda, each of which associatively operates on (e.g., reads or writes) a single tuple in the tuple space.

Besides the "generative tuple space" of Linda, a number of other interesting models have been proposed and used to support coordination languages and systems. Examples include various forms of "parallel multiset rewriting" or "chemical reactions" as in Gamma [10], models with explicit support for coordinators as in MANIFOLD[7], and "software bus" as in ToolBus[11]. A significant number of these models and languages are based on a few common

notions, such as pattern-based, associative communication [1], to complement the name-oriented, data-based communication of traditional languages for parallel programming.

Coordination languages have been applied to the parallelization of computation intensive sequential programs in the fields of simulation of Fluid Dynamics systems, matching of DNA strings, molecular synthesis, parallel and distributed simulation, monitoring of medical data, computer graphics, analysis of financial data integrated into decision support systems, and game playing (chess). See [2, 17, 21] for some concrete examples.

6 Classification

Coordination models and languages can be classified as either *data-oriented* or *control-oriented*. For instance, Linda uses a data-oriented coordination model, whereas MANIFOLD is a control-oriented coordination language. The activity in a data-oriented application tends to center around a substantial shared body of data; the application is essentially concerned with what happens to *the data*. Examples include database and transaction systems such as banking and airline reservation applications. On the other hand, the activity in a control-oriented application tends to center around processing or flow of control and, often, the very notion of *the data*, as such, simply does not exist; such an application is essentially described as a collection of activities that genuinely consume their input data, and subsequently produce, remember, and transform "new data" that they generate by themselves. Examples include applications that involve work-flow in organizations, and multi-phase applications where the content, format, and/or modality of information substantially changes from one phase to the next.

Coordination models and languages can also be classified as either *endogenous* or *exogenous*. For instance, Linda is based on an endogenous model, whereas MANIFOLD is an exogenous coordination language. 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.

Endogenous models are sometimes more natural for a given application. However, they generally lead to intermixing of coordination primitives with computation code, which entangles the semantics of computation with coordination protocols. This intermixing tends to scatter communication/coordination primitives throughout the source code, making the cooperation model and the coordination protocol of an application nebulous and implicit: generally, there is no piece of source code identifiable as the cooperation model or the coordination

protocol of an application, that can be designed, developed, debugged, maintained, and reused, in isolation from the rest of the application code. On the other hand, exogenous models encourage development of coordination modules separately and independently of the computation modules they are supposed to coordinate. Consequently, the result of the substantial effort invested in the design and development of the coordination component of an application can manifest itself as tangible "pure coordinator modules" which are easier to understand, and can also be reused in other applications.

7 Coordination at CWI

Experimental research on coordination has been going on at CWI since 1990. This work has produced IWIM[4, 3], a novel model for control-oriented coordination; MANIFOLD[5, 3, 9, 7], a pure coordination language based on the IWIM model; preliminary studies on the formal semantics of MANIFOLD[34, 33]; and Visifold[15], a visual programming environment for MANIFOLD. The implementation of the second version of MANIFOLD is now complete and it is being used in a number of applications. Theoretical work on the formal semantics of MANIFOLD and the mathematical models underlying IWIM and MANIFOLD are presently on-going.

Theoretical and experimental work on coordination were unified in 1997 under the *Theme SEN3: Coordination Languages* within the *Software Engineering Cluster* at CWI. This Theme aims at cross-fertilization of formal and applied research on coordination. The activity in SEN3 is on (1) development of formal methods, notably (operational) semantic models as a unifying basis for the development of debugging and visualization tools for coordination languages; (2) enhancements to and experiments with the MANIFOLD language and its visual programming and debugging environment; and (3) using MANIFOLD to work on real applications of coordination programming in areas such as numerical computing, distributed constraint satisfaction, and shallow water modeling.

8 Manifold

MANIFOLD is a coordination language for managing complex, dynamically changing interconnections among sets of independent, concurrent, cooperating processes[5]. The processes that comprise an application are either computation or coordinator processes. Computation processes can be written in any conventional programming language. Coordinator processes are clearly distinguished from the others in that they are written in the MANIFOLD language. The purpose of a coordinator process is to establish and manage the communications among other (computation or coordinator) processes. MANIFOLD is a control-oriented, exogenous coordination language based on the IWIM model.

IWIM stands for *Idealized Worker Idealized Manager* and is a generic, abstract model of communication that supports the separation of responsibilities and encourages a weak dependence of workers (processes) on their environment. Two major concepts in IWIM are separation of concerns and anonymous communication. Separation of concerns means that computation concerns are isolated from the communication and cooperation concerns into, respectively, worker and manager (or coordinator) modules. Anonymous communication means that the parties (i.e., modules or processes) engaged in communication with each other need not know each other. IWIM-sanctioned communication is either through broadcast of events, or through point-to-point channel connections that, generally, are established between two communicating processes by a third party coordinator process.

MANIFOLD is a strongly-typed, block-structured, event driven language, meant for writing coordinator program modules. As modules written in MANIFOLD represent the idealized managers of the IWIM model, strictly speaking, there is no need for the constructs and the entities that are common in conventional programming languages; thus, semantically, there is no need for integers, floats, strings, arithmetic expressions, sequential composition, conditional statements, loops, etc.[4] The only entities that MANIFOLD recognizes are processes, ports, events, and streams (which are asynchronous channels), and the only control structure that exists in MANIFOLD is an event-driven state transition mechanism. Programming in MANIFOLD is a game of dynamically creating (coordinator and/or worker) process instances and dynamically (re)connecting the ports of some of these processes via streams, in reaction to observed event occurrences. The fact that computation and coordinator processes are absolutely indistinguishable from the point of view of other processes, means that 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.

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 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

[4]For convenience, however, some of these constructs, syntactically, do exist in the MANIFOLD language. Currently, only the front-end of the MANIFOLD language compiler knows about such "syntactic sugar" and translates them into processes, state transitions, etc., so that as far as the run-time system (or even the code generator of the MANIFOLD compiler) is concerned, these familiar constructs "do not exist" in MANIFOLD.

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[6, 18].

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, 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 successfully used to implement parallel and distributed versions of a semi-coarsened multi-grid Euler solver algorithm in MAS2 at CWI. This represents a real-life heavy-duty Computational Fluid Dynamics application where MANIFOLD enabled restructuring of existing sequential Fortran code using pure coordination protocols that allow it to run on parallel and distributed platforms. The results of this work are very favorable: no modification to the computational Fortran 77 code, simple, small, reusable MANIFOLD coordination modules, and linear speed-up of total execution time with respect to the number of processors (e.g., from almost 9 to over 2 hours)[19, 20].

Other applications of MANIFOLD include its use in modeling cooperative Information Systems[31, 32], coordination of Loosely-Coupled Genetic Algorithms on parallel and distributed platforms[36, 37], coordination of multiple solvers in a concurrent constraint programming system[8], and a distributed propositional theorem checker in the *Theme SEN2: Specification and Analysis of Embedded Systems* at CWI.

9 Examples

An interesting example of an exogenous coordination protocol is a MANIFOLD program described in [6] that receives two process type definitions (A, and M) and recursively creates instances of itself (processes x_i), A (processes a_i), and M (processes m_i), and (re)connects the streams routing the flow of information

among them. This abstract coordination protocol can be compiled separately, and linked with the object code of other processes to build an application. It is immaterial what exactly processes A and M do, and process instances x_i, a_i, and m_i, that are created at run time, can run on various hosts on a heterogeneous platform. The depth of the recursion (i.e., the exact number of x_i's) depends on the number of input units, and the number of units each instance of a_i decides for itself to consume. What matters is the pattern of communication among, and creation of, these process instances.

Entirely different applications can use (the object code of) this same MANIFOLD program as their coordination protocol[6]. For example, we have used this protocol to perform parallel/distributed sorting, by supplying:

- as A, a sorter, each instance of which takes in $n > 0$ input units, sorts them, and produces the result as its output; and
- as M, a merger, each instance of which produces as its output the merged sequence of the two sequences of units it receives as its input.

We have also used this protocol to perform parallel/distributed numerical optimization, e.g., of a complex function by supplying:

- as A, an evaluator, each instance of which takes in an input unit describing a (sub)domain of a function, and produces its best estimate of the optimum value of the function in that (sub)domain; and
- as M, a selector, each instance of which produces as its output the best optimum value it receives as its input.

10 Formal Models and Semantics

The formal (operational) semantics of MANIFOLD is defined in terms of a two-level transition system[14]. The first level consists of a (large) number of transition systems, each of which defines the semantics of a single process, independently of the rest. The second level consists of a single transition system that defines the interactions among the first-level transition systems. The details of the internal activity of the first-level transition systems (e.g., their computations) are irrelevant for, and therefore unobservable by, the second-level transition system. This two-level approach to formal semantics reflects the dichotomy of computation vs. coordination that is inherent in MANIFOLD, and represents a novel application of transition system in formal semantics. More generally, this approach is useful for the definition of the formal semantics of other coordination languages as well. The key concept here is that the second level abstracts away the (computational) semantics of the first level processes, and is concerned only with their (mutually engaging) externally observable behavior.

Related to transition systems are the notions of *bisimulation* and *bisimilarity* which reflect the intuitive concept of the equivalence (or similarity) of the externally observable behavior of concurrent systems. Bisimulation is used to define the formal notion of *coinduction* as the counterpart for the familiar principle of induction. Coinduction, bisimulation, and *coalgebras*[25, 35] comprise a mathematical machinery analogous to the more familiar notions of induction, congruence, and algebras, that is suitable for the study of concurrency and coordination.

Traditionally, *initial algebras* have been used as mathematical models for finite data types, such as finite lists. Their counterparts, *final coalgebras*, are used as mathematical models for infinite data types and for the semantics of object-oriented programming languages. More generally, coalgebras can serve as models for dynamical and transition systems. Coalgebraic models seem very appealing candidates for a mathematically sound foundation for the semantics of MANIFOLD. 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.

An alternative approach to a mathematical foundation for the semantics of MANIFOLD can be sought in other category-theoretical models. The essence of the semantics of a coordinator process in MANIFOLD can be described as transitions between states, each of which defines a different topology of information-carrying streams among various sets of processes. What is defined in each such state is reminiscent of an (asynchronous) electronic circuit. Category theoretical models have been used to describe simple circuit diagrams[26]. Extensions of such models to account for the dynamic topological reconfiguration of MANIFOLD is a non-trivial challenge which, nevertheless, points to an interesting model of computation.

References

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

[2] Andreoli, J.-M., Hankin, C., and Le Métayer, D., Eds. *Coordination Programming: Mechanisms, Models and Semantics*. Imperial College Press, 1996.

[3] Arbab, F. Coordination of massively concurrent activities. Tech. Rep. CS–R9565, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, November 1995. Available on-line http://www.cwi.nl/ftp/CWIreports/IS/CS-R9565.ps.Z.

[4] Arbab, F. The IWIM model for coordination of concurrent activities. In *Coordination Languages and Models* (April 1996), P. Ciancarini and C. Hankin, Eds., vol. 1061 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 34–56.

[5] Arbab, F. Manifold version 2: Language reference manual. Tech. rep., Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, 1996. Available on-line http://www.cwi.nl/ftp/manifold/refman.ps.Z.

[6] Arbab, F., Blom, C. L., Burger, F. J., and Everaars, C. T. H. Reusable coordinator modules for massively concurrent applications. In *Proceedings of Euro-Par '96* (August 1996), L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, Eds., vol. 1123 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 664–677.

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

[8] Arbab, F., and Monfroy, E. Using coordination for cooperative constraint solving. In *Proceedings of the 1998 ACM Symposium on Applied Computing; Special Track on Coordination Models, Languages and Applications* (Atlanta, Georgia, February-March 1998), ACM.

[9] Arbab, F., and Rutten, E. P. B. M. Manifold: a programming model for massive parallelism. In *Proceedings of the IEEE Working Conference on Massively Parallel Programming Models* (Berlin, September 1993).

[10] Banâtre, J.-P., and Le Métayer, D. Programming by multiset transformations. *Communications of the ACM 36*, 1 (January 1993), 98–111.

[11] Bergstra, J., and Klint, P. The ToolBus Coordination Architecture. In *Proc. 1st Int. Conf. on Coordination Models and Languages* (Cesena, Italy, April 1996), P. Ciancarini and C. Hankin, Eds., vol. 1061 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 75–88.

[12] Bergstra, J. A., and Klop, J. W. Process algebra for synchronous communication. *Information and Control 60* (1984), 109–137.

[13] Bolognesi, T., and Brinksma, E. Introduction to the ISO specification language LOTOS. *Computer Networds and ISDN Systems 14* (1986), 25–59.

[14] Bonsangue, M. M., Arbab, F., de Bakker, J. W., Rutten, J. J. M. M., and Scutellá, A. A transition system semantics for a control-driven coordination language. Tech. Rep. to appear, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, 1998.

[15] Bouvry, P., and Arbab, F. Visifold: A visual environment for a coordination language. In *Coordination Languages and Models* (April 1996), P. Ciancarini and C. Hankin, Eds., vol. 1061 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 403–406.

[16] Carriero, N., and Gelernter, D. LINDA in context. *Communications of the ACM 32* (1989), 444–458.

[17] Ciancarini, P., and Hankin, C., Eds. *1st Int. Conf. on Coordination Languages and Models*, vol. 1061 of *Lecture Notes in Computer Science*. Springer-Verlag, April 1996.

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

[19] Everaars, C. T. H., Arbab, F., and Burger, F. J. Restructuring sequential Fortran code into a parallel/distributed application. In *Proceedings of the International Conference on Software Maintenance '96* (November 1996), IEEE, pp. 13–22.

[20] Everaars, C. T. H., and Koren, B. Using coordination to parallelize sparse-grid methods for 3D CFD problems. *Parallel Computing* (1998). to appear in the special issue on Coordination.

[21] Garlan, D., and Le Métayer, D., Eds. *2nd Int. Conf. on Coordination Languages and Models*, vol. 11282 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1997.

[22] Hoare, C. Communicating Sequential Processes. *Communications of the ACM 21* (August 1978).

[23] Hoare, C. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science. Prentice-Hall, 1985.

[24] INMOS Ltd. *OCCAM 2, Reference Manual*. Series in Computer Science. Prentice-Hall, 1988.

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

[26] Katis, P., Sabadini, N., and Walters, R. The bicategory of circuits. Tech. Rep. 94-22, University of Sydney, 1994. To appear in: Journal of Pure and Applied Algebra.

[27] Leler, W. LINDA meets UNIX. *IEEE Computer 23* (February 1990), 43–54.

[28] Milner, R. *Communication and Concurrency*. Prentice Hall International Series in Computer Science. Prentice Hall, 1989.

[29] Milner, R. The polyadic π-calculus: A tutorial. Tech. Rep. Res. Report LFCS-91-180, Lab. for Foundations of Computer Science, Edinburgh University, 1991.

[30] Milner, R. Elements of interaction. *Communications of the ACM 36*, 1 (January 1993), 78–89.

[31] Papadopoulos, G., and Arbab, F. Control-based coordination of human and other activities in cooperative information systems. In *Proceedings of the Second International Conference on Coordination Languages and Models* (September 1997), Lecture Notes in Computer Science, Springer-Verlag, pp. 422–425.

[32] Papadopoulos, G. A., and Arbab, F. 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* (Atlanta, Georgia, February-March 1998), ACM.

[33] Rutten, E. P. B. M. Minifold: a kernel for the coordination language Manifold. Tech. Rep. CS-R9252, Centrum voor Wiskunde en Informatica, Amsterdam, November 1992.

[34] Rutten, E. P. B. M., Arbab, F., and Herman, I. Formal specification of Manifold: a preliminary study. Tech. Rep. CS–R9215, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, 1992.

[35] Rutten, J. J. M. M. Universal coalgebra: A theory of systems. Tech. Rep. CS-R9652, Centrum voor Wiskunde en Informatica, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands, 1996. Available on-line http://www.cwi.nl/ftp/CWIreports/AP/CS-R9652.ps.Z.

[36] Seredynski, F., Bouvry, P., and Arbab, F. Distributed evolutionary optimization in Manifold: the rosenbrock's function case study. In *FEA'97 - First International Workshop on Frontiers in Evolutionary Algorithms (part of the third Joint Conference on Information Sciences)* (Mar. 1997). Duke University (USA).

[37] Seredynski, F., Bouvry, P., and Arbab, F. Parallel and distributed evolutionary computation with Manifold. In *Proceedings of PaCT-97* (September 1997), V. Malyshkin, Ed., vol. 1277 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 94–108.

[38] Thornton, J. E. *Design of a Computer: The Control Data 6600*. Scott, Foresman and Company, 1970.

[39] Wegner, P. Interaction as a basis for empirical computer science. *ACM Computing Surveys 27*, 1 (Mar. 1995), 45–48.

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