Coordination as Constrained Interaction (Extended Abstract)

Peter Wegner

Brown University

1 Introduction

Protocols that coordinate interactive communication among software components have qualitatively different behavior from algorithms that progressively transform the state within a software component. Algorithm behavior is captured by Turing machines whose expressive power is that of computable functions. Coordination must handle temporal and other nonfunctional interactive properties that cannot be expressed algorithmically [We2]. For example, driving home from work can be viewed as a coordination problem that involves real-time coordination with other cars that cannot be expressed by noninteractive algorithms. Airline reservation systems and robots likewise must handle real-time coordination that cannot be expressed algorithmically. Coordination systems are open systems that must handle unpredictable external events that occur dynamically during the process of problem solving.

Coordination is constrained interaction: it constrains interaction protocols among communicating software components. Though explicitly specified coordination rules are limited by human ability to handle complexity, coordination patterns in nature and in practical software systems can be arbitrarily complex. Strongly constrained coordination models can be weaker than Turing machines, but models that express general-purpose interaction are more expressive than Turing machines [We1]. Practical coordination models must express not only simple static resource constraints specified by tokens of Petri nets or Prolog rewriting rules but also dynamic constraints of air traffic controllers and complex organizations. Coordination models can provide a unifying framework for World-Wide Web protocols like http, interface definition protocols for interoperability, and data exchange (DX) protocols for heterogeneous databases.

This extended abstract reviews some existing models of coordination from the viewpoint of an outsider looking in and considers the place of coordination in the broader context of models of interaction.

2 Coordination Models

Petri nets are a "pure" model of coordination. A Petri net has components (transitions) that fire by consuming resources at input places and delivering resources

to output places. Resources are represented by multisets and coordination is realized by rewriting rules for multisets. Petri nets are flexible in expressing distributed coordination that depends on the availability of resources at distributed locations (places). They abstract away from specific data structures by permitting only a single data structure (tokens) and abstract away from algorithms by permitting only computation by execution.of coordination rules.

Petri net tokens are nonreusable resources better modeled by linear logic than by traditional logics. Linear logic provides a framework for coordination of nonreusable resources that admits both sequential and parallel coordination, illustrated by interaction abstract machines in [ACP].

Linda [CG] is a less abstract model of coordination that represents data by tuples (records) accessed by associative pattern matching. It realizes a blackboard model of coordination by processes that reside in and communicate through a shared tuple space. Its simple set of coordination commands for process input and output is compatible with a wide variety of algorithmic programming languages. It neatly separates coordination and computation, focusing on the complete and precise specification of coordination primitives that can be naturally embedded in algorithmic languages. The operational semantics of Linda's coordination primitives is examined in [**?**].

Object-oriented programming languages coordinate communication among objects by message-passing protocols. Coordination primitives in object-oriented languages are less flexible than in Petri nets in that protocols whose execution depends on inputs from several objects cannot be directly specified. Messages specify only pairwise coordination of components. Linda has even weaker coordination primitives, specifying only unary coordination of components with a shared tuple space. Both Linda and object-based systems can build up multiway coordination by composite coordination patterns. Object-oriented systems can simulate a shared tuple space by objects whose state consists of associatively addressed tuples.

Coordination of heterogeneous distributed components can be realized by megaprogramming languages [WWC] that specify coordination among megamodules by programs whose statements specify both the sequence of operations to be executed and the transformation of messages from the format of senders to a format acceptable to receivers. Megaprograms for simple coordination tasks can be very simple. However, coordination of distributed concurrently executing tasks with atomicity and real-time constraints can be arbitrarily complex.

Megaprograms are examples of "middleware" [Surv] interspersed between components to realize coordination. Models of coordination elevate middleware to a first-class status, corresponding to the role of management in large organizations. Middleware mediates among software components by transforming data and coordinating actions: the view of middleware as mediators is developed in [Wi]. Explicit coordination becomes more important as systems become large, just as explicit management structures are more important for large than for small companies. However, experience shows that simple typeless coordination systems like UNIX pipes or http are often more effective than more elaborate

strongly-typed interface definition languages. Static type-compatilbility requirements that promote safety and efficiency appear to have an unacceptably high implementation cost in today's coodination technology.

Communication among heterogeneous components can be realized also by models of interoperability such as OMG's Common Object Request Broker Architecture [CORBA] and Microsoft's component object model [COM]. Interoperability solves the problem of reusability of resources specified in one environment by components in another environment. Most work on interoperability assumes a client-server, object-based model of communication [NT]. Coordination among heterogeneous distributed components may utilize protocols developed for client-server compatibility in more general contexts of coordination.

Coordination is concerned both with rules for scheduling and firing actions and with communicating and transforming data among components. Executing actions and communicating data, which require very different models (Petri nets and CORBA) need to be integrated:

coordination -¿ firing rules + data exchange

Designers of coordination languages must address the following issues [Proc]:

1. What are the entities being coordinated? procedures, objects, processes, components of a specific type, subsystems;
2. What are the media (architectures) for coordination? blackboard model, client-server, Petri net, Pi calculus, CORBA, COM, UNIX pipes, middleware;
3. What are the protocols and rules of coordination? multiset rewriting rules, message send and receive, object request brokers, megaprogramming, HTML;

In [Proc] it was further suggested that coordination languages be classified according to the dimensions of scalability, encapsulation, decentralization, dynamicity, open-endedness, generativeness, and semantic richness. These dimensions were used to classify and justify design decisions of existing languages. Though these dimensions are useful, the design space for coordination is not well understood and more work is needed to characterize and explore it.

3 Models of Interaction

The design space for coordination can be embedded in the broader context of models of interaction. Interactive systems can be modeled by interaction machines definable by extending Turing machines with input and output actions (read and write statements). Interaction machines are open systems that model external events occurring during the process of computation, while Turing machines are closed, noninteractive systems that shut out the external world while computing an output from an input. Interaction machines have richer behavior than Turing machines because they can react to real-time interactive behavior not expressible by Turing machines.

Greater richness follows formally from the fact that input streams of interaction machines cannot be modeled by finite tapes, since they can always

be dynamically extended. The computational distinction between Turing and interaction machines is expressed mathematically by that between enumerable finite sets of tapes and nonenumerable infinite sets of streams whose cardinality is that of the real numbers. It is entirely appropriate that the ability of interaction machines to express the "real" world is modeled mathematically by the "real" numbers. Infinite divisibility of continuous physical space and infinite extensibility of physical time give rise to dual nonenumerable abstractions of reality.

Real numbers are represented by infinite streams of digits that model infinite divisibility of continuous mathematical and physical space. Interaction machines turn this model inside-out, representing the external world by infinite streams of inputs. The infinite extensibility of outer temporal reality can be modeled by infinite divisibility of continuous space. Since the set of all infinite digit streams are in one-to-one correspondence with both the nonenumerable real numbers and the input streams of an interaction machine, interaction machines have a nonenumerable number of inputs.

Interaction machines cannot be specified by sound and complete logics: they are incomplete in the sense shown by Godel for the integers (the set of all true statements about them is not formally enumerable by theorems). Church's thesis that the intuitive notion of computing corresponds to formal computing by Turing machines is seen to be invalid or at least inapplicable, since interaction machines determine a very natural notion of computing more powerful than Turing machines. The Chomsky hierarchy of machines is extended beyond Turing machines to synchronous and asynchronous interaction machines, but mathematical characterization of machine behavior by sets or formal grammars cannot be similarly extended, showing that machines can specify more powerful forms of behavior than mathematical notations.

Nonenumerability captures the mathematical essence of interactive computing. Its operational essence is the control of actions by the external environment rather than by rules of inner computation. Interactive systems are operationally described by their observable behavior in terms of interaction histories.

Interaction histories of simple objects, like bank accounts with deposit and withdraw operations, are described by operation streams called traces. Operations whose effects depend on their time of occurrence, as in interest-bearing bank accounts, require time-stamped traces. Objects with inherently nonsequential interfaces, like joint bank accounts accessed from multiple automatic-teller machines, have inherently nonsequential interaction histories. Interaction histories of distributed systems, just like history in history books, consists of overlapping inherently nonsequential processes. Whereas interaction histories express the external unfolding of events in time, instruction-execution "histories" simply express an ordering of inner events of an algorithm without any relation to the actual passage of time.

Algorithms are time-independent transformations from inputs to outputs, while objects and interaction machines provide interactive services over time. Algorithmic time is measured by number of instructions executed rather than

by the actual time of execution to provide a hardware-independent (abstract) measure of logical complexity. In contrast, the duration and the time that elapses between the execution of operations may be interactively significant. Operation sequences have temporal as well as functional properties, while instruction sequences have a purely functional semantics.

Interaction machines can model both the inner algoritmhic behavior and the inter-component coordination behavior of objects, software engineering applications, robots, intelligent agents, distributed systems, and networks like the internet and World-Wide Web. Coordination models of the World-Wide-Web that refine current HTML protocols represent a challenge with a great practical payoff. HTML has proved itself as a ubiquitous data interchange language with trivial coordination and data interchange protocols. Extending the coordination power of HTML, for example to hot Java, without impairing its universality is an important research problem.

Coordination languages express interactive properties of software systems. Petri nets and Linda are interactive formalisms. Interaction protocols expressed by multiset rewriting rules have control structures that cannot be entirely captured by algorithmic control structures. Inference rules of linear logic cannot be expressed by sound and complete models. There are many small indications that the tidy examples of simple coordination protocols are only the tip of the iceberg of a much larger and less tidy space of real-world coordination protocols that we have not even begun to explore. The extension of models of coordination from toy examples of multiset rewriting to real-world coordination of distributed system and software engineering is a challenging problem.

References

[ACP] J. Andreoli, P. Ciancarini, and R. Pareschi, Interaction Abstract Machines. In Research Directions in Concurrent Object-Oriented Programming, Eds Agha, Wegner, Yonezawa, MIT Press, 1993.

[CG] N. Carriero and D. Gelernter, Coordination Languages and Their Significance, CACM, Feb 1992.

[CNY] Paolo Ciancarini, Oscar Nierstrasz, and Akinori Yonezawa, Proc. ECOOP '94 Workshop on Coordination Languages, LNCS 924, Springer Verlag, 1995, especially articles by Gelernter and Ciancarini.

[COM] Kraig Brockschimidt, Inside OLE 2, 2nd edition, Microsoft Press, 1995.

[CORBA] Architecture and Specification, Revision 2.0, Object Management Group, July 1995.

[NT] Oscar Nierstrasz and Dennis Tsichritzis Eds, Object-Oriented Software Composition, Prentice Hall, 1996. Especially Chapter 3 by Dimitri Konstantas.

[Proc] Proceedings of First Annual Workshop on Coordination, Imperial College Dept of Computer Science, December 1994.

[Surv] Research Directions in Software Engineering, Computing Surveys, June 1995.

[We1] Peter Wegner. Interactive Foundations of Object-Based Programming. IEEE Computer, Oct. 1995.

[We2] Peter Wegner. Foundations of Interactive Computing. Report CS-96-01, Jan 1996.

[Wi] Gio Wiederhold. Mediation in Information Systems. Computing Surveys, June 1995.

[WWC] Gio Wiederhold, Peter Wegner, and Stefano Ceri. Towards Megaprogramming. CACM Nov 1992.