BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-91-M14

Object Models

by
Robert Haring-Smith

Authorization to lend and reproduce this thesis

As the sole author of this thesis, I authorize Brown university to lend it to other institutions or individuals for the purpose of scholarly research.

[illegible]
Robert Henry Haring-Smith

5/20/91
(Date)

I further authorize Brown University to reproduce this thesis by photocopying or other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

[illegible]
Robert Henry Haring-Smith

5/20/91
(Date)

Object Models

by

Robert Henry Haring-Smith

B.A., Swarthmore College, 1974
Sc.M., University of Illinois, 1977
Ph.D., University of Illinois, 1981

Thesis

Submitted in partial fulfillment of the requirements for
the Degree of Master of Science
in the Department of Computer Science at Brown University

May 1991

This thesis by Robert Henry Haring-Smith is accepted in its present form by
the Department of Computer Science as satisfying the
thesis requirement for the degree of
Master of Science

Date May 2 1991 Peter Wegner
Peter Wegner

Approved by the Graduate Council

Date 5/15/91 [illegible]

Acknowledgments

I owe a great debt of thanks to the many people who made it possible for me to complete this thesis.

The Departments of Chemistry and Geological Sciences at Brown University very generously allowed me to rearrange my work schedule to leave time for research and writing. They also provided the well-maintained computer facilities with which this document was prepared.

My advisor, Professor Peter Wegner, was extraordinarily patient during several abortive attempts to combine thesis work with very demanding jobs. He provided many useful pointers, not least of which was the example and content of his own writing. His insight and ability to make sense of a large body of knowledge by finding useful taxonomies for it is remarkable.

My son Whitney has been touchingly understanding during the past few months of intense work on this thesis, allowing me to monopolize the computer at home and accepting that I could not come out and play as much as usual. I owe him not only thanks, but also a day at the zoo and one at the circus and ...

For all the contributions made by these people, though, I would never have completed my degree work without the gentle prodding and support of my wife Tori. She has been a partner in this work right from my initial decision to pursue a degree in computer science (alarmingly many years ago). She has looked out for my interests far better than I have myself, and adjusted her life to accommodate everything from the all-nighters I spent programming when completing course work to my inattentive wool-gathering when pondering some problem in my thesis. I owe her my gratitude and freely give her my love.

Contents

List of Figures

Chapter 1

Introduction

Object-based systems and programming languages have attracted a lot of attention in recent years. Proponents see them as the most promising means of systematizing the development of large-scale software projects by facilitating the encapsulation and reuse of existing code. But it has proven difficult to create languages that readily permit reuse of code without violating encapsulation of program modules in some fashion. Success in the creation of object-based systems may ultimately depend on the construction of models for the object environment that permit careful specification of systems, provide a well-defined semantics, and enforce the essential properties of object-based systems that make them so intriguing.

Object-based systems arose from simulation of real-world systems. Their origins can be traced to the language Simula [8], developed in the late sixties to expedite the coding of simulation programs. Later, with the introduction of classes of objects and other means of reusing code, object-based systems were seen as useful tools for software engineering of large systems. More recently, object-based systems have been studied as a means of exploiting the parallel-processing capabilities of multiprocessing or distributed computing systems. [31]

In turning their attention from reuse to concurrency in object-based systems, researchers left many fundamental problems of software reuse unsolved. It proved easier to adapt object-based systems to a parallel processing environment than to develop a flexible but safe method for sharing and reusing code in such systems, perhaps because there had been a decade or more of independent research on concurrency before it was applied to object-based programming. Now we are beginning to see models that attempt to deal simultaneously with issues of concurrency and sharing.

1.1 Granules and Aggregates

In exploiting both concurrency and code-sharing, key concerns are granularity and composition. What is the appropriate size of the smallest shareable component of an object-based system, and how can we specify aggregates of such components that can be shared *en masse*? What degree of concurrency is appropriate for an object-based system and how can one describe the interoperation of collections of concurrent agents?

As in most areas of computer science, the choice of granularity involves trade-offs. Sharing that is too coarsely grained can be inflexible and unresponsive to the evolving needs of a system. It may tempt programmers to create *ad hoc* arrangements whose semantics are at odds with usual expectations for a system's behavior, as when "inherited" properties of a class are overridden in a subclass (meaning, of course, that the properties actually are not inherited). On the other hand, very fine-grained sharing can become unmanageable or inefficient. The sheer number of shareable components may grow overwhelming, complicating the programming process and introducing a lot of overhead in the run-time environment.

Of course, the construction of software tools to deal with the complexity of a system can help. A crucial tool, in this context, is a well-designed means of assembling shareable components into modules that realize on a larger scale some of the properties of the system's atomic components, principally their encapsulation against uncontrolled access to their internal structure. This is where the model-builder's role is most evident–in determining how to abstract the basic components, what operations should be used to combine components, and what the semantics of the combination is.

Many of the comments just directed toward sharing apply equally well to concurrency. Coarse-grained concurrency (such as none at all) misses a chance to exploit the seemingly very natural fit between object-based software technology and parallel processing hardware technology. But very fine grained concurrency increases the overhead of protecting critical shared resources from concurrent access while ensuring that no agent is starved for lack of access to these resources.

When dealing with concurrency, unlike sharing, we can draw on widely recognized, largely successful existing models. If there were a "Calculus of Shared Modules (CSM)" to accompany Milner's Calculus of Communicating Systems (CCS), the state of object-oriented programming would be very different. Of course, the task of integrating these two calculi into one model might still remain.

1.2 Looking Ahead

This thesis, then, will examine some approaches to modeling the object environment in object-based systems, considering especially their success at handling concurrency and sharing.

Chapter 2 discusses several aspects of object-based systems that must be dealt with in any successful model of such systems—encapsulation, concurrency, communication, and sharing. While only the use of objects and perhaps classification are regarded as defining characteristics of an object-based system, features like message passing and concurrency are often critical to the practical success of a system.

Chapter 3 explores the modeling of concurrency in some detail, both for the insight that it offers on modeling generally and because two of the object models that we examine in subsequent chapters are rooted in the modeling of concurrency. In chapter 3, too, a semantics for CCS is developed, a process that exemplifies the construction of semantics for other models.

Chapters 4 through 6 each examine one model at some length—the Actors model of Hewitt and Agha [2], the Abacus model of Nierstrasz [37], and the Maude model of Meseguer [29]. The Actors model successfully represents a dynamic, highly concurrent object environment, but provides little support for classification, inheritance, or other forms of sharing. Abacus addresses some of the deficiencies of Actors by augmenting CCS with facilities for encapsulating complex, multi-agent modules. Maude exemplifies a complementary approach to modeling object-based systems, one that begins with a strong foundation in type theory, then tackles issues of concurrency.

Throughout the discussion, we must keep in mind the role of modelling in computer science, an issue that Milner discusses in his recent exposition of CCS [33]. Whereas physical and social scientists construct models of pre-existing systems over which they have little control, computer scientists seek useful models to guide their own future construction of systems. The models should be judged by their ability to clarify the important issues represented by the class of systems being modelled and to guide the programmer in her use of such systems.

In many respects, all work in computer science is modelling–down to the construction of the simplest programs. These modelling activities differ in their level of abstraction. So, while an object-oriented program may itself be a model of some real-world situation, it also exists within the framework of a more abstract model implemented in the object-oriented language used to write the program. That model, in turn, may be the outcome of design choices from a universe of possible designs that adhere to the object-oriented view of computation.

Our concern is not with individual programs or languages (though they provide essential illustrations and tests of one's ideas), but with the nature of object-oriented programming and the prospects for providing it with a rigorous conceptual foundation. The latter is an extraordinarily difficult problem, lying at a nexus of several of contemporary computer science's greatest challenges—the development of formal programming language semantics, the construction and management of complex software systems, and the exploitation of concurrency.

Chapter 2

Aspects of object models

What features of object-based systems must a model reflect? Some of those discussed here are intrinsic features of such systems, while others are critical to the practical success of a system. In this chapter we briefly examine several important aspects of object-based systems that should be considered when constructing a model—the objects themselves and their encapsulation, concurrency, communication, and different forms of sharing. The key issue is how each aspect of a model of objects can be sustained without permitting a violation of the fundamental aspect of encapsulation.

Of these different aspects, concurrency has received the most extensive study, culminating in several formal models of concurrent computation. These well-developed models can, in turn, be the foundation of models for object-based programming environments. For this reason, and because of the growing importance of concurrent systems, models of concurrent computation are covered in much more detail in the following chapter.

2.1 Objects and Encapsulation

Some of the appeal of object-based systems derives from the notion that the programmer's intuitions of the real world developed over decades of living in it can be brought to bear in the artificial world of an object-based program.

In the real world are people, organizations, machines, and structures. Each has an internal state determined by its mental condition or memory or the arrangement of its components. Each provides services to others and demands services of others in its environment. Requests for service must be expressed in a manner that the recipient of the request can understand—e.g., through suitable use of language for a human recipient. or appropriate use of control mechanisms for mechanical recipients of service requests.

In abstracting the complexity of the real world to the artificial world of a program, considerable simplification takes place. At the heart of the object-based environment is a collection of objects, each with an internal state represented. perhaps, by the value of *instance variables* associated with the object. Typically, an object will respond to some form of message directed to it. The response may involve changing the state of the object or generating messages directed to other objects or to itself.

The existence of objects with an internal state is a defining characteristic of object-based systems. [38,47] There is also universal agreement that the internal state of an object should be encapsulated in some fashion, so that state changes occur in a carefully controlled way. One way of expressing this is to say that an object is an abstract data type with state [43]. The abstract type defines the interface through which other entities interact with the object and can affect the state of the object.

This notion of object also recalls automata which change state only as a result of processing input symbols. In fact, an automaton equipped with an unbounded set of internal states provides a reasonable model of the behavior of a single object. But the difficulty in developing a viable model for object-based systems lies in modelling *collections* of objects and the ways in which they interact.

A related alternative model might view the entire system as an automaton, with the objects as a scheme for providing some structure for the representation of the automaton's state and for partitioning the state into separately controllable components [47, p. 23]. An indication of how this arrangement might be implemented can be seen in the work on Statecharts and Objectcharts. [6,21,22,23]

It is now more common, however. to regard objects as active agents rather than passive automata slavishly processing inputs in lockstep with other components of the system in which they are embedded. The active agents can delay processing some inputs while giving other inputs priority treatment. Their actions need not be synchronized with other objects, except when synchronization is essential to the task at hand. And they can be charged with guarding their own encapsulation.

The objects in an object-based system inevitably become a fundamental unit of granularity and construction in the system. The granularity of concurrency is measured against objects (e.g., whether concurrency is permitted within or merely between objects) and program modules are typically aggregates of objects.

Key design issues for a model are the amount of structure allowed an object and the pervasiveness of objects in the system. Are objects "flat" constructs containing only their instance variables, or can objects contain other objects? Is "everything" in the system an object, or is the system built from a number of fundamentally different units? A homogeneous model in which all constructs in a system are objects certainly has aesthetic appeal and may simplify theoretical analysis, though perhaps at the cost of rather unnatural representation of some concepts.

2.2 Concurrency

While creating encapsulated objects is not difficult in itself, preserving that encapsulation as a system is endowed with other features can be challenging, as many authors have pointed out. [46,36]

One desideratum for object-oriented systems that has had a great influence on the development of models for such systems is the ability to exploit concurrency. Great strides have been made in developing computing hardware that enjoys the benefits of parallel processing, either through the inclusion of several processors in a single machine or by use of a network of machines working in close cooperation. Creation of software to simplify the task of programming for such hardware environments has lagged behind. But object-oriented systems promise to be a big step in the right direction because a software environment of multiple, independent objects lends itself so naturally to a division of labor among multiple processors.

The issue to be resolved is how to permit concurrent activity of many objects without violating the encapsulation of their internal states. As in any concurrent system, precautions must be taken to ensure that a data structure being manipulated by one process cannot be accessed by another process when the structure is in an inconsistent state. Nierstrasz [36] identifies three approaches to meeting this need: In the *orthogonal approach*, methods for assuring mutually exclusive access to critical data (such as locks, semaphores, or monitors) are separate from facilities for encapsulating an object. The contrasting *homogeneous approach* uses the same mechanism to control all access to an object's internal state, typically by using active objects endowed with facilities for their own protection. The *heterogeneous approach* combines the first two approaches by providing both active objects that protect themselves and passive objects whose use is confined to single-threaded active objects.

The orthogonal approach is common in object-based languages in which concurrency has been grafted onto a pre-existing language. The homogeneous approach is aesthetically more pleasing and likely to be theoretically more tractable than the other approaches. The heterogeneous approach offers certain efficiencies by allowing passive objects to be maintained with reduced overhead.

The granularity of concurrency with respect to objects is another parameter which varies from one object-based system to another. An object might permit just a single thread of computation to enter it at a time, or could handle multiple threads. Alternatively, it might appear externally to be single-threaded, but spawn multiple threads internally in order to respond to a request.

For modeling purposes—in particular, to take advantage of existing models of concurrency—it is convenient to allow concurrency at least between individual objects. If an object's structure permits it to contain other objects, then concurrency within objects would be provided as a corollary, while if objects are flat structures then they might be modelled as internally sequential.

Once the units of concurrency in a model are established, one can consider how to assemble these units into more complex concurrent systems. Again, established models of concurrency provide considerable guidance, but care must be taken that encapsulation of objects or modules is not compromised because of the way that they are embedded in a concurrent system.

2.3 Communication

Virtually every object-based system is said to use *message passing* to let objects request services of other objects. In many respects, message passing is not so much a special feature of object-based systems as it is an unavoidable corollary of the encapsulation of an object's internal state. If direct manipulation of the internal state is to be avoided, then interaction with the object must occur at a distance. A client object sends a message to request a service rather than directly accessing the internals of a server object.

This behavior is no different from what one might see in a well-written program in a procedural language using abstract data types. A program module needing to use a data structure will invoke the appropriate procedure rather than manipulating the data structure itself. Where the procedural and object-based languages may differ is in the flexibility of the communication mechanisms and the extent to which their use is enforced. The procedural program, in most cases, *could* access the data structure without invoking the procedure; the object-based system, by contrast, will typically provide no means of using information internal to an object without sending that object a message.

There are variations in the message-passing protocols used by different object-based systems, often depending on the degree of concurrency anticipated by system designers. A system running on a single processor might do very well by broadcasting messages, perhaps by making them available in an area of memory shared by all objects. But such a mechanism could create an intolerable bottleneck in a system that is highly distributed. In the latter case, point-to-point communication may make more sense, although some means of addressing objects and assuring the delivery of messages is then needed.

The synchronicity of communication is another variable in object-based systems, particularly those that provide for some concurrent operations. In synchronous communication, both sender and receiver must be ready for the exchange before a message is passed. Some systems require the sender to await a response before proceeding with other activity. This may be the means of assuring mutually exclusive access to encapsulated data in the presence of concurrent computation. Other systems synchronize sender and receiver at the moment of communication, then allow each to proceed independently.

If asynchronous communication is permitted, then some provision must be made for the possibility of an object receiving a message while it is busy responding to an earlier message. The receiving object could be required to manage its own flow of communications, or there might be a uniform facility provided to all objects by the system in which they are embedded.

Some of these issues fall more in the realm of implementation than modelling. The model may imply that asynchronous communication is permitted, or require that any implementation provide assured delivery of messages, but typically says little more.

2.4 Sharing

The prototypical object-based languages, Simula [8] and Smalltalk-80 [19], provide mechanisms for classifying objects that share common properties and for allowing a class of objects to bequeath its properties to a subclass (which might then augment those properties with others specific to the subclass). For a time, the use of classification and inheritance was treated as a defining characteristic of object-based systems. More recently, systems that rely heavily on encapsulation of data objects without providing these forms of sharing have been developed, and recognized as object-based. But systems using classes and inheritance occupy an important place among object-based systems. Wegner [47] identifies this category of system as *object-oriented.*

Classes and inheritance are just two of a large number of sharing mechanisms used by different object-based systems. Each of these mechanisms is something of a two-edged sword. They simplify the construction of many similar objects and help to enforce consistency of behavior between such objects. But in reducing the locality of data by establishing complex interrelationships among different collections of objects, they complicate the reuse of modules in different contexts and risk violating the encapsulation of objects [46].

The two most widely used sharing mechanisms are class and type hierarchies. Two objects of the same class can be constructed from a template giving their implementation in terms of the data structures and procedures needed for their operation. Objects of the same type share identical specifications of external interface, indicating the structure of messages that objects of that type can accept. For example, a description of the *class* ComplexNumbers might indicate that each complex number is to be represented by its real and imaginary part, and that complex numbers will have available to them a method for drawing graphical representations of themselves. *Type* information for the same collection of objects would say nothing about their internal representation, but would indicate that they will respond to messages asking for computation of the modulus or argument of a complex number, or for arithmetic combinations of two complex numbers. It might also specify a message requesting that a complex number be depicted graphically. The depiction would likely be accomplished using the previously mentioned graphical representation method available to the ComplexNumbers class.

It is easy to confuse class and type because often the methods used by a class are just the operations published for use by clients of the class's objects. Generally, in fact, a class is also a type, but the converse is not true. There is a tendency to merge the type and class hierarchies into one, as was done in Smalltalk, even though the information provided by the two hierarchies is logically distinct and the merged hierarchy may be inflexible and not readily amenable to the development of rigorous semantics.

Researchers have had greater success in developing a careful theory of types alone, perhaps because type information is purely behavioral and only partially specifies an object. Nierstrasz [40] and Danforth and Tomlinson [9] survey the study of types for object-based languages. An algebraic theory of types forms the underpinnings of Meseguer's approach to concurrent object-based computation, as we shall see in chapter 6.

The modeling of sharing seems particularly difficult, no doubt partly because there is no widely accepted solution to the problem. At times, particularly when talking of code sharing, we appear to need a metalanguage allowing us to describe which parts of a program are to be accessible to which objects. On the other hand. the complexity of sharing may just be a reflection of the complexity of reality. If a simple block structure is not enough to describe and thus control access to program modules, perhaps that is because the program seeks to model complicated, changing interactions among real-world entities that refuse to adhere to a straightforward hierarchical pattern of communication. A neat and tidy semantics may be possible only by settling for programs that less faithfully represent the domains that the programs seek to model.

Still, even "less faithful" representations would benefit from a systematic approach to sharing. The model for sharing should anticipate from the outset the different degrees of sharing that might be needed. While inheritance can be represented in a hierarchical structure, inheritance that can be overridden is not hierarchical. Some other structure, such as a lattice, should be the basis for the model in the latter case. Software tools can help manage sharing by keeping track of shareable modules in the environment, but without a clear semantics to enforce, such tools may only simplify the misuse of the modules.

Efficient sharing of class information is perhaps the great unsolved problem of object-oriented semantics. Many researchers defer the problem by building models whose foundation is a representation of concurrency, the hope being that classification mechanisms can be added later. Others, such as Meseguer [29], create a well-supported type structure, then implement classes as types by defining messages to assign or retrieve the value of each constituent of an object in that class. This does not offer the flexibility of logically distinct type and class structures.

Chapter 3

Models of concurrent systems

As a starting point for models of concurrent object-oriented systems, most researchers have turned to the two best known and most highly refined models of concurrent systems, Hoare's Communicating Sequential Processes (CSP) [26] and Milner's Calculus of Communicating Systems (CCS) [33]. We will first consider object-based systems as examples of a more general class of systems called reactive systems, then discuss some of the requirements for models of reactive systems, and finally examine some of the details of CCS as it applies to object-oriented systems.

3.1 Reactive and Transformational Systems

Pnueli [45] contrasts two basic types of systems—transformational systems and reactive systems. Transformational systems are exemplified by early computer systems that provided just one processor on which each process would run to completion (or abortion) before another process was allowed to commence. A program running in batch mode can also be viewed as a transformational system. Such systems are modelled readily as functions transforming an input into an output. The function can be implemented as a machine or program in which a series of instructions is executed sequentially.

Transformational systems are closed systems. Once they have received the necessary input, all the actions that they take are internal until the output is produced. Whatever the actual timing of inputs and outputs for a transformational system, one can regard all inputs as being available when a process is initiated and all outputs as being generated at the completion of the process. The system's environment, which is the source of the input and the target of the output, plays such a limited role that there is no need to include it explicitly in models of transformational systems.

Time, too. plays little role in modelling transformational systems since the value of a function does not depend on when it is evaluated. When the function is implemented as a machine or program, there is a rudimentary notion of time implicit in the sequence of transitions or instructions executed, but this involves merely a simple linear ordering in which quantitative details of timing are irrelevant. Variables and other objects in a model of a transformational system are under the sole control of the process being modelled. They cannot change between instructions of the modelled process, so time *is*, in effect, the sequence of instructions or transitions.

One is often interested in the correctness of a given implementation of a transformational system—whether the output produced is indeed the result that would be obtained by applying the given function to the machine's or program's input. Formal methods of program verification that provide an adequate framework for reasoning about transformational systems have been developed by Floyd and Hoare [4,14]. These methods involve decomposing a system into components each of which is itself transformational. The input/output relationship for the entire system is then inferred from the input/output behavior for the components.

The features of a transformational system that allow it to be modelled by a function are that no events of extrinsic interest occur between the initial input and the final output, and that, aside from determining the original input, the system has total control over the computation that produces the output. Complex systems that can be subdivided into simple transformational components can also be treated successfully by a functional model.

Reactive systems, on the other hand, either share resources with an external agent or are in frequent communication with their environments. Complex systems whose components are reactive are also best regarded as reactive even if they might also be viewed as transformational systems. For example, an object-oriented program to compute some function is, at its top level of abstraction, a transformational system. But any analysis of the program is likely to involve components (objects) whose behavior cannot easily be modelled functionally.

A functional model of computation does not serve well for many modern systems, which may involve multiprogramming or multiprocessing to permit several processes to coexist in the system and interact with one another. Nor is the functional model well suited to programs such as operating systems or database management systems which, in principle, never terminate but continually respond to input from their environments. Representing the semantics of a system by a function suppresses all information about intermediate states of the system—information crucial to anticipating system response to actions that might be taken by the environment at any time. Implementations of functions introduce intermediate states, but formalisms such as automata provide no systematic means of describing a complex system in a structured way or of indicating concurrent operation of several processes.

3.2 Requirements for a Model of a Reactive System

A model of reactive systems should provide some means of specifying the behavior of a system and of proving that an implementation of the system meets its specification. Thus, implementations and specifications must have a well-defined semantics that lends itself to some method of proving correctness of the implementation. To facilitate correctness proofs, modelling must, most likely, be compositional. That is, it should be possible to describe a system as an assembly of interacting components, each of which can be specified and proved correct in its own right. Compositionality is one means of managing the complexity of concurrent systems, and, of course, compositionality lends itself well to object-oriented programming and its software engineeing goal of creating components that can safely be reused in a variety of contexts.

Reactive systems present special problems when it comes to proving their correctness because of vagaries in the timing of different processes. One must not only confirm that processes will behave correctly when running, but also ensure that the system cannot become deadlocked and cannot produce behavior in which some processes are denied access to critical resources, whatever the relative speeds of these processes are.

The model should also allow a system to be described at an appropriate level of abstraction. The functional model of an isolated process abstracts away from details of circuitry or programming to treat automata or programs as black boxes that merely exhibit appropriate input/output behavior. Similarly, we would most likely want to characterize interactive systems in terms of the observable aspects of their behavior, and regard as equivalent any implementations whose observable behaviors are essentially the same. This approach is in accord with maintaining the encapsulation of the objects in an object-oriented system.

Furthermore, in combining different modules to form a larger system, we may want to suppress details of the modules' specifications that are unimportant to the observable behavior of the system as a whole. That is, an action regarded as observable for a module may be an internal communication of the overall system. Similarly, our choice and arrangement of modules may be an implementation detail when viewed from the level of the system.

The model will necessarily abstract some features of the real world for the sake of simplification or to focus on certain aspects of real systems. Most models of reactive systems include some notion of an event without specifying the nature of an event in great detail. The most important property of an event is that it is atomic as far as the system is concerned; it need not be instantaneous. Typical events are the receipt or transmission of messages. The current state or configuration of a system is also handled abstractly by most models of concurrent systems. Details of how a state is represented (say, by a collection of variables with certain values) are largely ignored. Quantitative aspects of time are likewise omitted from most models, which instead use qualitative relationships among events, like precedence or co-occurrence. Hewitt and Baker [25] and Clinger [7] present axiomatic approaches to timing considerations in concurrent systems.

There are, of course, certain specific types of behavior that a model should be able to describe in detail, including sequential, nondeterministic, and concurrent operation. Sequential operation is a sort of base case, familiar from the description of non-interactive systems. It may reflect the temporal ordering of events within a process or the completion of one process before the start of another.

The possibility of concurrency is at the heart of modeling reactive systems. A multiprocessor machine would certainly include concurrent processes, but even in the simple case of a single program interacting with its environment, we may want to regard the environment as a process operating concurrently with the program. Such an approach allows the environment to be described in the same language used for all other processes, lending an attractive economy to the model.

Nondeterminism arises in two ways: First, a collection of processes—each, perhaps, corresponding to an object in an object-oriented system—constitutes a non-deterministic system because, in ignoring timing considerations, we cannot (and may not want to) specify which of the processes will act next. From a given state of the system, the next action taken may be constrained to lie in the set of actions that individual processes can carry out next, but within that set the next action is undetermined. Models that record a system's behavior using a total ordering of the observed events must regard as equivalent behaviors that differ only in the order in which events from such sets of undetermined actions occur.

Another source of nondeterminism in a system is the environment in which it operates. A system currently anticipating an input from its environment has little control over what that input might be. A model of the system would reflect this lack of control in non-deterministic behavior. While this non-determinism seems a more intrinsic feature of reactive systems than that arising from the uncertain timing of events, it is, to some extent, an artifact of our wish to develop compositional approaches. An overall system might be deterministic, yet be built from components which, when viewed in isolation, are non-deterministic.

There is considerable variety in the mathematical models of reactive systems that have been proposed. Graphical models include Petri nets and statecharts. Milner's CCS and SCCS, and Hoare's CSP are algebraic, while models like event structures represent concurrent systems using relations or partial orders. Semantics for these models may be operational or denotational. We will be concerned here principally with CCS, but it is also convenient to introduce a notation—transition systems—that might be regarded as the assembly language of reactive system modelling, but which nonetheless allows a simple graphical representation of system behavior.

3.3 Transition Systems

Transition systems are related to non-deterministic automata. To be specific, a *labelled transition system* is a structure

$$T = (Q, \Sigma, q_0, E, \lambda),$$

where Q is a set of *states*, Σ is a finite *alphabet* or set of *labels*, $q_0 \in Q$ is the *initial state*, $E \subseteq Q \times Q$ is the *transition relation* or set of *events*, and $\lambda : E \longrightarrow \Sigma$ is the *labelling function.*

A labelled transition system differs from the usual notion of a non-deterministic automaton in several ways. A transition system may have an infinite set of states and lacks any designation of final states. This is consistent with our informal notion of a reactive system as one which runs indefinitely with no intention of producing a final result. Also, the input symbols of a non-deterministic automaton have been replaced by labels which may correspond to message-passing or other events related to operation of the system. But we can still regard a transition system as a computing device that starts in state q_0 and successively changes state as events occur.

An advantage of using labelled transition systems as a model of computation is that they can be represented effectively as a directed graph with vertex set Q, the set of states, and edges given by E, the transition relation. Figures 3.1 and 3.2 are two labelled transition systems over the alphabet $\{a, b\}$. The first has four states and models counting modulo 4, with events labelled a corresponding to incrementing the counter and those labelled b being decrements of the counter. The second system is an infinite binary tree in which each node has outgoing edges (or transitions) labelled by a and b.

A *computation* or *path* in a labelled transition system is a sequence of states $p = (q_1, q_2, \ldots, q_n)$ such that, for each i with $1 \leq i < n$, $(q_i, q_{i+1}) \in E$. The *length* of the path is $n - 1$. When $n = 1$, we have the *empty path* at q_1.

We can extend the labelling function λ to computations as follows: Let Σ^* be the set of all finite

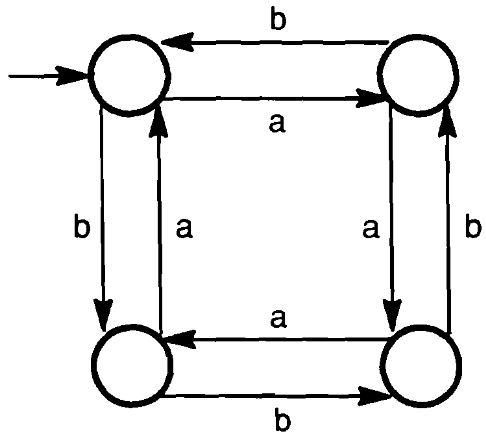

Figure 3.1: A labelled transition system, T

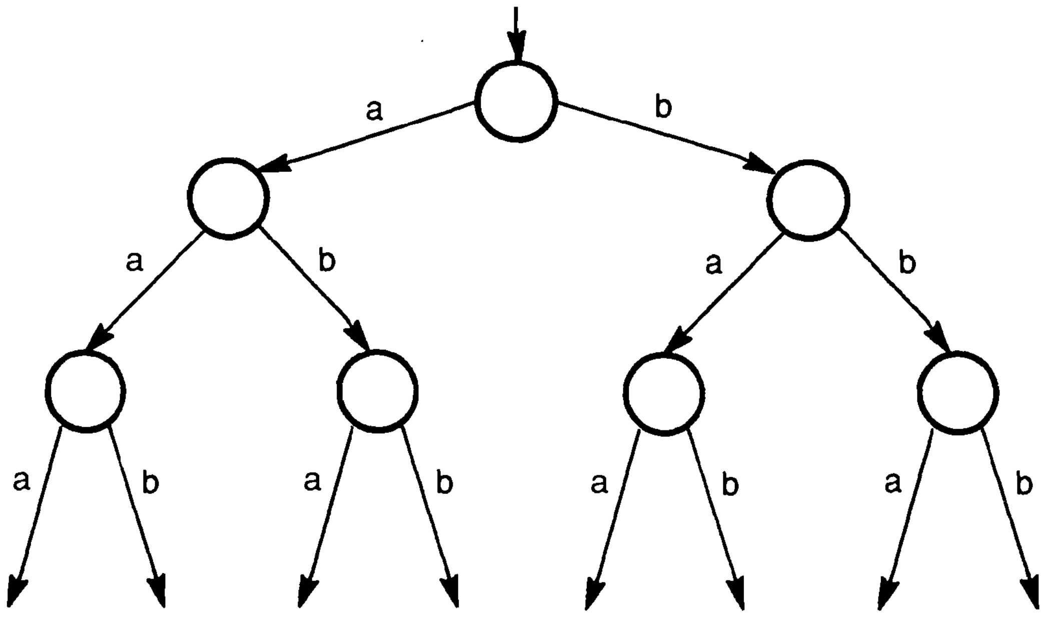

Figure 3.2: Another transition system, UT

strings of labels, including the empty string ε. For the arbitrary path p given above, define

$$\lambda^*(p) = \begin{cases} \varepsilon & \text{if } n = 1 \\ \lambda^*((q_1, \ldots, q_{n-1}))\lambda((q_{n-1}, q_n)) & \text{otherwise} \end{cases}$$

That is, $\lambda^*(p)$ is the string formed by concatenating the labels on the individual edges of p. If σ is the label on a path from state q to state q', we write $q \xrightarrow{\sigma} q'$.

Any transition system T can be "unrolled" to create a new transition system $\mathcal{U}T$ that records all of the computation histories of the original system. Starting with transition system

$$T = (Q, \Sigma, q_0, E, \lambda),$$

the unrolled system is

$$\mathcal{U}T = (Q', \Sigma, q_0', E', \lambda'),$$

where Q' is the set of all computations of T that begin at the initial state of T, and the initial state q_0' of $\mathcal{U}T$ is the empty path at q_0. An event of $\mathcal{U}T$ is the extension of a computation of T by one more event of T. That is, $(p, p') \in E'$ iff $p = (q_0, q_1, \ldots, q_n)$ and $p' = (q_0, q_1, \ldots, q_n, q_{n+1})$ are computations of T. The label $\lambda'((p, p'))$ in $\mathcal{U}T$ is the same as the label $\lambda((q_n, q_{n+1}))$ on the final step of p'. As an example, notice that the transition system of Figure 3.2 is the result of unrolling the system in Figure 3.1.

There is a one-to-one correspondence between states of $\mathcal{U}T$ and computations of $\mathcal{U}T$ beginning in the initial state. This indicates both that $\mathcal{U}T$ has the graph structure of a tree and that unrolling $\mathcal{U}T$ will not produce an essentially different transition system. $\mathcal{U}T$ and $\mathcal{U}\mathcal{U}T$ are isomorphic in an easily-defined sense.

3.3.1 Modelling with Transition Systems

While transition systems will be a useful device for describing computation, they are not in themselves suitable models for reactive systems. Most importantly, we lack a language for specifying behavioral properties of transition systems and composing them from simpler systems. We also must indicate how transition systems represent concurrent behavior and how we can prove that a transition system has a desired property.

In some ways, transition systems are too detailed to be good models of reactive systems. What we really need to know about a reactive system is the way that it interacts with its environment. The system's states are important only insofar as they facilitate the appropriate behavior.

Consider a simple vending machine providing a small assortment of expensive beverages—cans of Coke or Sprite, each costing a dollar. The customer can insert a dollar bill or push one of two buttons. The machine must recognize the customer's action and dispense the appropriate drink. We will assume that if the customer puts in a second dollar bill instead of selecting a drink, then the machine will return the bill. Sometimes the machine may just return the first bill, too—behavior familiar to anyone who has used bills in a vending machine.

Figure 3.3 presents a transition system that represents the machine. Events labelled D, C, and S correspond to the machine's recognition of the customer's actions of inserting a dollar, selecting a Coke, and selecting a Sprite, respectively. Events labelled DD, DC, or DS correspond to dispensing a dollar, Coke, or Sprite. The labels thus indicate the visible behavior of the machine, and the transition system describes the sequences of visible events to which the machine responds.

Of course, the machine need not be built in the way that the transition system of Figure 3.3 indicates. For example, it might be that when a dollar is inserted, the machine enters a state in which it attempts to verify the genuineness of the bill. Only when the bill is found to be good would the machine enter a state in which it will acknowledge the customer's button pushing. Unobservable events such as checking the bill are usually indicated by a special label τ. A transition system that models such checking of the dollar bill is shown in Figure 3.4.

Just by noting the observable events, the customer cannot tell which of the two designs were used. In either design, the customer could either select a drink or have the dollar returned after inserting the first bill. While checking the bill may take time, there is no assurance that it will take enough time to be noticeable. In any case, our models are to be abstracting from quantitative timing details. All we require is that, if the system does not take any visible action in a certain state, it will eventually take an invisible action if such an action is among the options for the current state.

3.3.2 Equivalence Classes of Transition Systems

If systems are specified according to the behavior that they exhibit externally but are built by arranging internal states or subsystems, then there may be many systems satisfying a given specification, as was indicated above. Even more, two systems may be identical as far as any external observer can tell, so that both would satisfy all the same specifications. The systems are equivalent in some sense.

To submit this informal notion of equivalence to mathematical analysis, we must be precise about the sort of experiments that our external observer can perform. Varying the experiments allowed

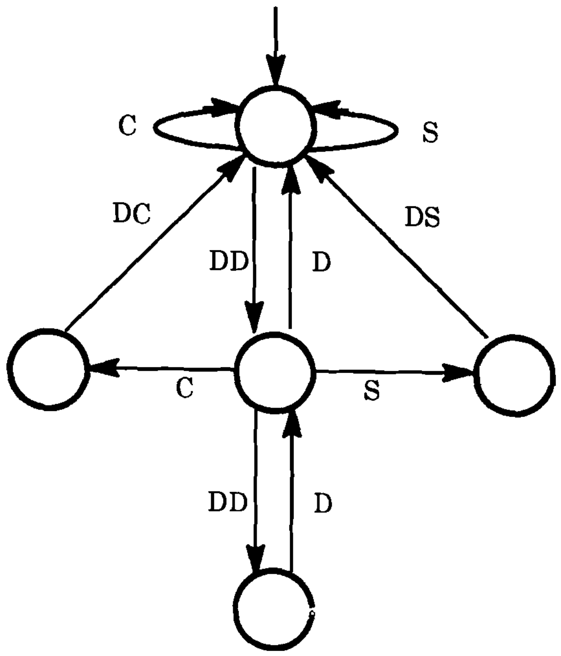

Figure 3.3: Transition system for vending machine

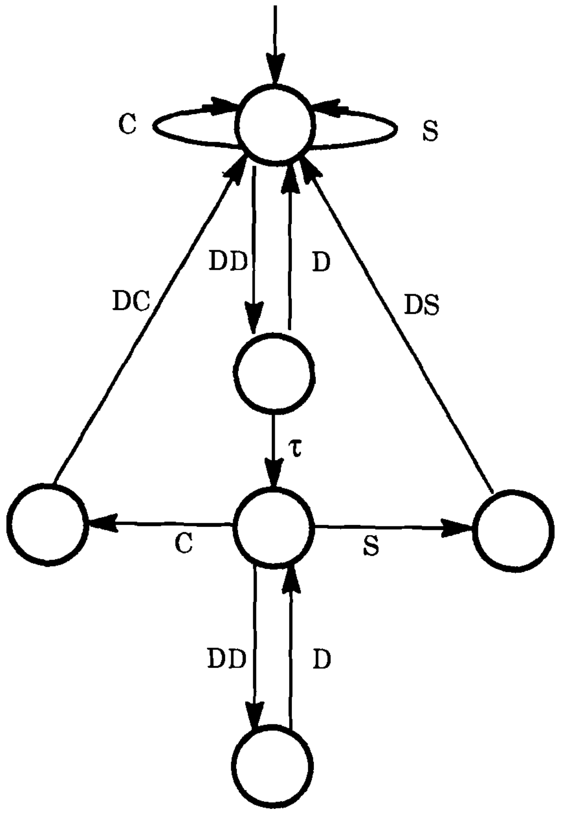

Figure 3.4: Modified vending machine

the observer leads to differing models of reactive systems. Shall the observer consider the effects of individual actions, bounded sequences of actions, or arbitrary sequences of actions? May the observer place the systems under study in arbitrary contexts? How should information about the *failure* of a system to perform an action be used? In the presence of nondeterminism, conclusions drawn from an experiment are necessarily provisional. That a nondeterministic system *didn't* take some action need not mean that it *can't* take that action. All that may be required is a slightly different sequence of nondeterministic choices during the preceding computation.

The answers to these questions are nearly as diverse as the models of concurrent systems, but many authors rely on some notion of *bisimulation*, the ability of two systems to simulate certain of one another's computations in some fashion.

3.3.3 Bisimulation

Let

$$T_i = (Q_i, \Sigma, q_i, E_i, \lambda_i), i = 1, 2$$

be two transition systems on the same alphabet. A *strong bisimulation* between T_1 and T_2 is a relation $R \subseteq Q_1 \times Q_2$ satisfying these conditions:

(i) The initial states are related: $(q_1, q_2) \in R$.

(ii) If $(s,t) \in R$ and $s \xrightarrow{\sigma} s'$ in T_1 for some $\sigma \in \Sigma$, then there is a state $t' \in Q_2$ such that $t \xrightarrow{\sigma} t'$ in T_2 and $(s', t') \in R$.

(iii) If $(s,t) \in R$ and $t \xrightarrow{\sigma} t'$ in T_2 for some $\sigma \in \Sigma$, then there is a state $s' \in Q_1$ such that $s \xrightarrow{\sigma} s'$ in T_1 and $(s', t') \in R$.

The systems T_1 and T_2 are *strongly bisimilar* if there is a strong bisimulation between them. In that case, we can write $T_1 \sim T_2$. One can show that the relation $\sim$ is an equivalence relation which is itself a strong bisimulation. Indeed, it is the largest strong bisimulation between two transition systems.

The existence of a strong bisimulation implies that a computation of any length beginning at a state of one system can be matched, step for step, by a computation with the same label beginning at a related state of the other system. In particular, any computation beginning at the initial state of one system can be simulated by the other system. For example, the systems in Figure 3.5 are strongly bisimilar, while those in Figures 3.6 and 3.7 are not. Figure 3.7 presents perhaps the trickiest of the three examples. If there were a bisimulation between them, both of the states 1 and

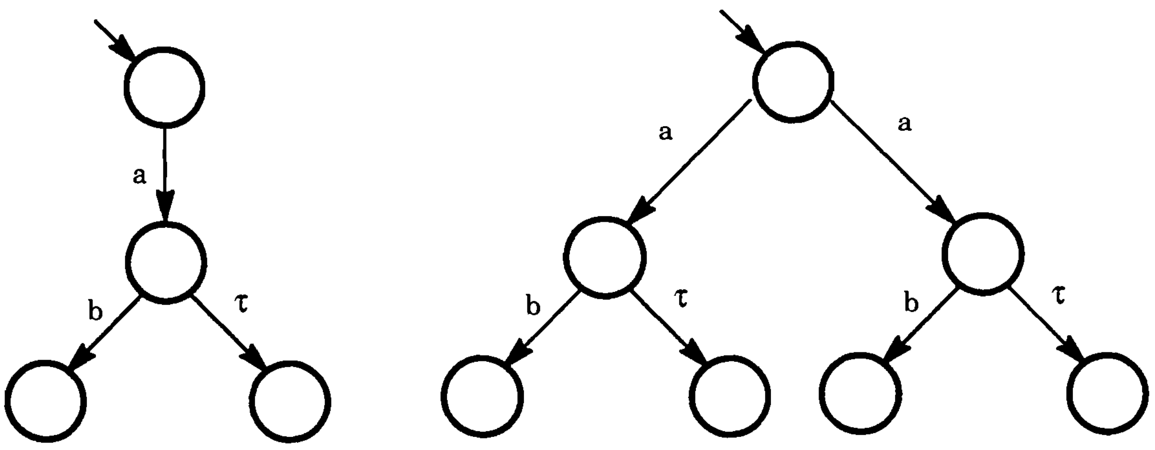

Figure 3.5: Strongly bisimilar systems

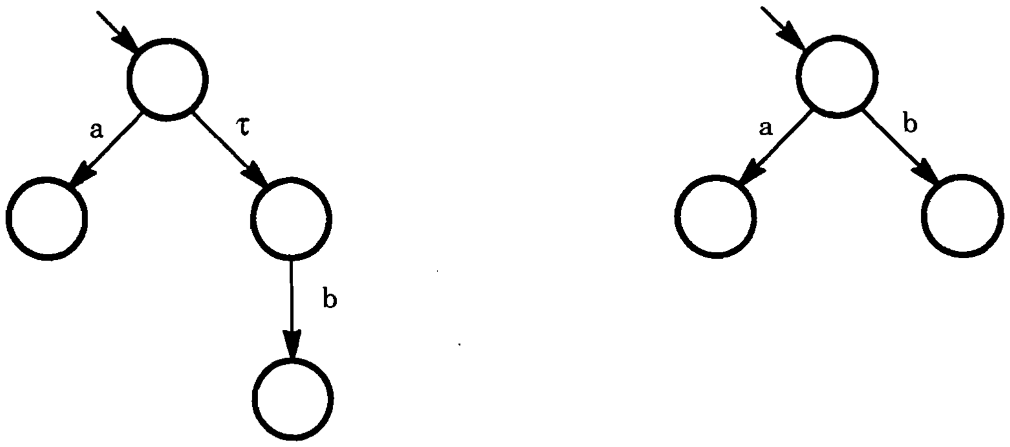

Figure 3.6: Weakly bisimilar systems that are not strongly bisimilar

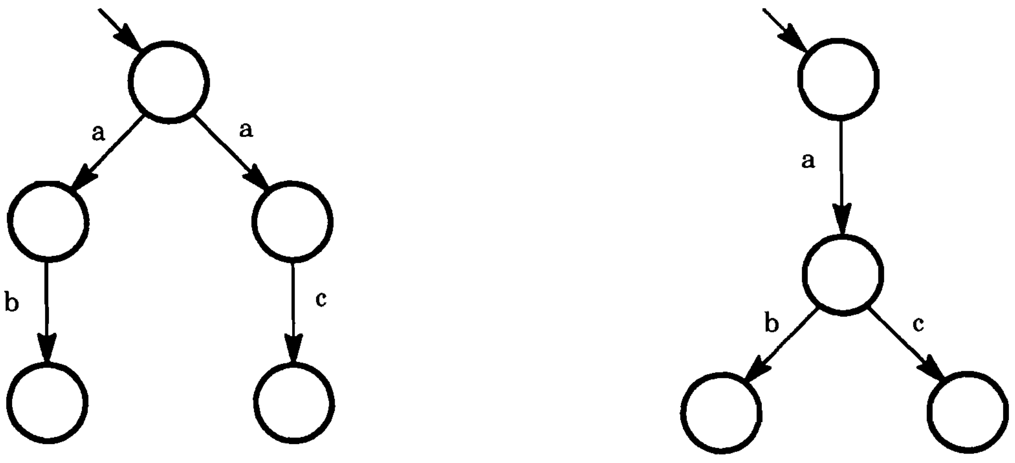

Figure 3.7: Non–weakly–bisimilar systems

2 of the first system must be related to state $1'$ of the second system. But then there is a transition from state $1'$ labeled b that is not matched by a similar transition from state 1. As another example of strong bisimulation, notice that any transition system T is strongly bisimilar to its unrolling $\mathcal{U}T$.

Strong bisimulation suggests that an observer of a system can detect the occurrence or failure of any action, including an invisible action, but cannot examine the internal state of a system. A weaker form of bisimulation posits an observer who can sense visible actions, but not invisible actions. For any $\sigma \in \Sigma^*$, define the *visible part* of σ to be the string $\sigma_v \in (\Sigma - \{\tau\})^*$ obtained from σ by removing all occurrences of τ. Write $q \overset{\sigma}{\Rightarrow} q'$ if $q \xrightarrow{\rho} q'$ for some $\rho \in \Sigma^*$ such that $\sigma_v = \rho_v$.

Weak bisimulation allows a computation labelled by σ in one system to be simulated by a computation in the other system whose visible part is σ_v. To be precise, if T_1 and T_2 are two transition systems, as above, then a weak bisimulation between T_1 and T_2 is a relation $R \subseteq Q_1 \times Q_2$ satisfying these conditions:

(i) The initial states are related: $(q_1, q_2) \in R$.

(ii) If $(s,t) \in R$ and $s \xrightarrow{\sigma} s'$ in T_1 for some $\sigma \in \Sigma$, then either $\sigma = \tau$ and $(s',t) \in R$, or there is a state $t' \in Q_2$ such that $t \overset{\sigma}{\Rightarrow} t'$ in T_2 and $(s',t') \in R$.

(iii) If $(s,t) \in R$ and $t \xrightarrow{\sigma} t'$ in T_2 for some $\sigma \in \Sigma$, then either $\sigma = \tau$ and $(s,t') \in R$, or there is a state $s' \in Q_1$ such that $s \overset{\sigma}{\Rightarrow} s'$ in T_1 and $(s',t') \in R$.

The systems T_1 and T_2 are *weakly bisimilar* if there is a weak bisimulation between them.

It is clear from the definitions that every strong bisimulation is a weak bisimulation. The systems in Figure 3.6 provide an example of systems that are weakly bisimilar but not strongly bisimilar. Under weak bisimilarity, the computation labelled τb in the first system of that figure can be simulated by the computation labelled b in the second system. Weak bisimilarity equates an invisible action with inaction, so a system all of whose actions are invisible is weakly bisimilar to a system making no transitions.

Strong and weak bisimilarity are both equivalence relations, so they partition the class of all transition systems over a given alphabet into disjoint equivalence classes of systems. System behavior is a characteristic of the class, while individual members of the class indicate different means of achieving that behavior. The equivalence class is a single semantic object associated with certain system specifications. By changing the equivalence relation used to classify transition systems, one can tune the semantics of a model so that it is in accord with the intended meaning of the specification language and so that it is mathematically tractable.

3.4 Calculus of Communicating Systems

Milner's Calculus of Communicating Systems (CCS) [32,33] was an early, highly influential attempt to provide a model for concurrent systems that would have the properties outlined in section 3.2. It provides a framework within which much of the later work in the area of concurrent systems can be understood, and illustrates many of the concepts introduced in connection with transition systems.

3.4.1 Syntax and Informal Semantics of CCS

Milner adopted the point of view that a model of concurrent systems that strove for compositionality would most likely be algebraic. His task, then, was to find a small set of simple systems and primitive operations from which all other systems could be constructed. These systems and operations would be the constants and operations in his algebra of communicating systems.

CCS models asynchronous processes that synchronize by communicating. Asynchronicity means that, while each process takes actions in a certain sequence, there is no time metric that would allow the occurrence of an event in one process to be compared to events in another process. Our only means of synchronizing such processes is to use communication channels through which two processes can rendezvous at certain points in their computation. A process requiring an input from a particular source must wait for the source process to reach the point where it can output the needed information. The synchronization of the output of one process with the input of another gives a loose control of the ordering of events between the two processes.

We can develop CCS using the language of transition systems. We require a label set Σ that can be expressed as the disjoint union $\Delta \cup \bar{\Delta} \cup \{\tau\}$ of three sets of actions. Δ and $\bar{\Delta}$ are in bijection with one another. In fact, we let

$$\bar{\Delta} = \{\bar{\alpha} \mid \alpha \in \Delta\}.$$

The actions α and $\bar{\alpha}$ are *complementary.* We can, for example, imagine that α represents sending an output signal on a certain channel, while $\bar{\alpha}$ corresponds to the receipt of an input signal on the same channel.

We shall also restrict our attention to transition systems whose underlying graph structure is that of a tree. Milner terms these systems *synchronization trees.* The effect of this restriction is to suppress details about states since they are characterized by the paths through which they can be reached. The restriction is, up to strong bisimulation, no loss in modelling power since every equivalence class of strongly bisimilar systems contains the unrollings of the systems in the class, and those unrollings are trees. Ignoring details about a system's state is also consistent with the goal of encapsulating the state.

We can now give a recursive definition of the syntax of CCS, along with an informal description of the intended semantics. For this purpose, let X be a countable set of *variables*, Σ an alphabet with the structure described above, and I a countable index set. Assume that B, B', and B_i (for each $i \in I$) are well-formed CCS expressions. Then the following expressions are also well-formed:

(i) *nil*. This expression is meant to denote a totally inactive system, one with a single state and no transitions.

(ii) x, for any $x \in X$. The variable x is free in this expression. Variables are used to express recursive relationships, as we will see below.

(iii) $\sigma.(B)$, for any $\sigma \in \Sigma$. This *action* expression describes a system whose initial action is labelled σ and which thereafter exhibits the behavior indicated by B. Expressions of this sort describe sequential behavior. Any variables that are free in B are also free in $\sigma.(B)$.

(iv) $\sum_I(B_i)$. This *summation* over the index set I represents a non-deterministic choice among the systems represented by the B_i. A synchronization tree for this expression could be formed from trees for the B_i by identifying all of their initial states as the initial state of the summed system. Any variable free in some B_i is free in the sum.

(v) $(B) \mid (B')$. This expression represents concurrent *composition* of the systems represented by B and B'. Free variables of $(B) \mid (B')$ are those variables free in either B or B'. The semantics of this expression includes elements of concurrency, nondeterminism, and abstraction. This configuration of processes also gives the constituent processes the opportunity to communicate with one another.

(vi) $(B)\backslash\alpha$, where $\alpha \in \Sigma$. This *restriction* expression is intended to describe a process that behaves as described by B, but that takes no actions labelled α or $\bar{\alpha}$. The free variables of $(B)\backslash\alpha$ are the same as the free variables of B.

(vii) $(B)[S]$, where $S : \Sigma \rightarrow \Sigma$ is a *relabelling* map satisfying $S(\tau) = \tau$ and $S(\bar{\alpha}) = \overline{S(\alpha)}$. (Here, we regard the overline operator as an involution, so that $\bar{\bar{\alpha}} = \alpha$.) A relabelling is more a convenience than an essential part of the CCS model. It allows many processes differing only by a systematic relabelling of their actions to be generated from a single prototype. Variables free in B will also be free in $(B)[S]$.

(viii) rec $x.(B)$. This expression using recursion permits the definition of systems with infinite sets of states. Occurrences of the variable x that are free in B are bound by this use of the recursion operator. The process described by this expression can be thought of as a solution of the equation $x = B$ up to strong bisimilarity. In order to ensure the existence of a unique solution, we insist that free occurrences of x in B be *guarded*—that is, that they appear within subexpressions of the form $\sigma.(B')$. The examples given below may help to clarify the use of recursion in CCS.

To reduce the number of parentheses needed to disambiguate expressions, we will use the following operator precedence (from tightest binding to least binding): restriction and relabelling, action and recursion, composition, summation.

Before giving a formal semantics for CCS, it is worth looking at a few simple examples using synchronization trees. For instance, the smaller system in Figure 3.5 can be described by the expression

$$a.(b.nil + \tau.nil)$$

If we let B denote this expression, then the larger system in the same figure can be described by $B + B$. The strong bisimilarity between the two systems can then be expressed as $B \sim B + B$. In fact, the relationship $B \sim B + B$ is a property of summation that does not depend on the definition of B. The two systems in Figure 3.7 can be described by the expressions $a.b.nil + a.c.nil$ and $a.(b.nil + c.nil)$. Since, as was pointed out earlier, the two systems are not strongly bisimilar, we see that action does not distribute simply over summation.

The previous examples were all finite systems. We must use recursion to describe an infinite system like that in Figure 3.2. Let x denote that system. Both of the subtrees of the root node are isomorphic to the tree as a whole, so they could be represented by x as well. The relationship between the whole tree and these two subtrees is expressed by the equation $x = a.x + b.x$, so an expression for the tree is rec $x.(a.x + b.x)$. The free occurrences of x in $a.x + b.x$ are guarded by the actions a and b.

Conversely, suppose that we have an expression such as rec $x.(a.nil + b.x)$ and want to determine the system that it represents. We can proceed as follows. Since x represents the system we want, we can regard it as referring to the initial state of the system. From that state, a transition by action a leads to a *nil* state having no further transitions, while a transition labelled b leads back to x. Such behavior is also exhibited by the two-state transition system in Figure 3.8. Its unrolling to a synchronization tree is given in the same figure.

We stated earlier that guards are needed in recursive expressions to ensure uniqueness of the systems represented by the expressions. As an extreme example of an expression in which the absence of guards leads to a multiplicity of solutions, consider rec $x.x$. This should represent a system satisfying the equation $x = x$ up to strong bisimilarity. But, of course, *any* system would satisfy this equation.

3.4.2 Operational Semantics for CCS

Since Milner's introduction of CCS, a wide variety of methods for supplying semantics for CCS expressions has been proposed. Milner himself [32] let an expression denote an equivalence class of synchronization trees. Winskel [48] suggested using event structures; Goltz and Mycroft [20] and Olderog [42] gave Petri net semantics; and Degano and Montanari [10] devised distributed transition systems to specify the meaning of a CCS expression. We shall follow the lead of Plotkin [44] in letting a CCS expression specify a transition system whose states are themselves certain CCS expressions. (Plotkin actually used his technique to give an operational semantics for CSP, but the method transfers readily to CCS.)

A *term* in CCS is an expression having no free occurrences of variables. For any term, we will describe a transition system whose behavior is that which the term is intended to express. Let B be a term over the alphabet Σ. Let the transition system T_B be a system on the same alphabet whose states are the CCS terms over Σ and whose initial state is the term B. The events and labelling function of T_B are to be determined by the axioms and inference rules given below. The system has only those transitions that can be inferred using these rules.

In each rule, existence of the labelled transition above the line allows one to infer the existence of the labelled transition below the line. The statement associated with action expressions is an axiom of this inference system; the transitions exist unconditionally. Except in the recursion rule, B, B', B'', C, and C' are any CCS terms, $\alpha \in \Sigma - \{\tau\}$, and $\sigma \in \Sigma$. In the recursion rule, B can be any expression having no free variables other than x.

We also introduce a notation for substitution of one expression in another. For any expression E, let

$$E\{E_1/x_1, \ldots, E_n/x_n\}$$

designate the expression obtained from E by replacing each free occurrence of x_i in E by the expression E_i. (In the process, it may be necessary to rename some bound variables of E to avoid clashes with variables appearing in the E_i.) A shorthand notation for this substitution is $E\{\tilde{E}/\tilde{x}\}$, where $\tilde{E}$ and $\tilde{x}$ represent parallel vectors of expressions and variables, respectively.

Action.

$$\frac{}{\sigma.B \xrightarrow{\sigma} B}$$

Summation.

$$\frac{B \xrightarrow{\sigma} B'}{B + B'' \xrightarrow{\sigma} B'} \qquad \frac{B \xrightarrow{\sigma} B'}{B'' + B \xrightarrow{\sigma} B'}$$

Composition.

$$\frac{B \xrightarrow{\sigma} B'}{B \mid B'' \xrightarrow{\sigma} B' \mid B''} \qquad \frac{B \xrightarrow{\sigma} B'}{B'' \mid B \xrightarrow{\sigma} B'' \mid B'}$$

$$\frac{B \xrightarrow{\sigma} B', C \xrightarrow{\bar{\sigma}} C'}{B \mid C \xrightarrow{\tau} B' \mid C'}$$

Restriction.

$$\frac{B \xrightarrow{\sigma} B'}{B \setminus \alpha \xrightarrow{\sigma} B' \setminus \alpha} \text{ if } \sigma \notin \{\alpha, \bar{\alpha}\}$$

Relabelling.

$$\frac{B \xrightarrow{\sigma} B'}{B[S] \xrightarrow{S(\sigma)} B'[S]}$$

Recursion.

$$\frac{B\{\text{rec } x.B/x\} \xrightarrow{\sigma} B'}{\text{rec } x.B \xrightarrow{\sigma} B'}$$

Some of these rules deserve further comment. The rules for the composition of two processes indicate the concurrent operation of those processes by granting to the composition all of the actions of the components. Either component can develop within the composite just as it would on its own. Actions of the components may be interleaved in any fashion. Any nondeterminism of the components is, of course, present in the composite, but additional nondeterminism can be introduced if both components are capable of taking a particular action.

Composition also allows the components to communicate with one another. If one component can take action σ and the other can take the complementary action $\bar{\sigma}$, then the composite can, in effect, take the two actions simultaneously and invisibly. The components communicate and synchronize by taking complementary actions, but these acts of communication are not visible to external observers.

The rule for recursion uses substitution into an expression B, but since we assume that x is the only free variable in B, the result of the substitution is a CCS term. The requirement that occurrences of x in B be guarded allows us to know actions of $B\{\text{rec } x.B/x\}$ without already knowing actions of rec $x.B$. For example, if B is $a.nil + b.x$, then $B\{\text{rec } x.B/x\}$ is

$$a.nil + b.\text{rec } x.(a.nil + b.x).$$

The action axioms show us that the latter process has actions a, leading to process nil, and b, leading to process rec $x.(a.nil + b.x)$. Hence, by the inference rule for recursion, rec $x.(a.nil + b.x)$ has the same actions.

To illustrate the operational semantics described here, Figure 3.9 gives a transition system for the term

$$(a.b.nil + b.nil) \mid \bar{a}.b.nil.$$

The given transition system differs from the one specified in this section only by the omission of states not reachable from the initial state. The two systems are in the same strong bisimilarity class.

3.4.3 Equivalences of CCS Expressions

While the operational semantics presented in the preceding section clearly describes how a given process will evolve as it takes actions, the model is unsatisfactory in that it ascribes semantic significance to any syntactic differences between terms. For example, in Figure 3.9, the terms $b.nil \mid nil$ and $nil \mid b.nil$ are distinguished even though they can take the same actions leading to the same new state. Likewise, the term $nil \mid nil$ is syntactically and, hence, semantically distinct from nil although neither term has any associated actions. More generally, we do not expect the meaning of a summation or composition to change simply because the subterms are written in a different order; the same nondeterministic choice or concurrent behavior is anticipated in any case. But the operational semantics just presented does associate different transition systems to summations or compositions differing only in the ordering of their components.

The solution to this problem is to develop an equivalence relation on CCS terms that will make only the desired distinctions between terms. Several such equivalence relations have been proposed, but we will consider just two, namely, *strong equivalence* and *observation equivalence.* These are the two most commonly used equivalences and are well suited to our needs.

We have already seen that each term B in CCS has an associated transition system T_B, and that the relations of strong and weak bisimilarity can be defined on transition systems. Connecting these two facts gives us the equivalence relations on terms that we need.

So, we will say that terms B and B' are ***strongly equivalent*** if the associated transition systems T_B and $T_{B'}$ are strongly bisimilar. Likewise, B and B' are ***observation equivalent*** if T_B and $T_{B'}$ are weakly bisimilar.

Either form of equivalence is sufficient to prove the following laws, among others, for any CCS expressions B, B', and B'', and any action σ and visible action α:

1. $B + B' \equiv B' + B$
2. $B + (B' + B'') \equiv (B + B') + B''$
3. $B + \mathit{nil} \equiv B$
4. $B + B \equiv B$
5. $\mathit{nil} \setminus \alpha \equiv \mathit{nil}$
6. $(B + B') \setminus \alpha \equiv B \setminus \alpha + B' \setminus \alpha$
7. $(\sigma.B) \setminus \alpha \equiv \mathit{nil}$ if $\sigma \in \{\alpha, \bar{\alpha}\}$
8. $(\sigma.B) \setminus \alpha \equiv \sigma.(B \setminus \alpha)$ if $\sigma \notin \{\alpha, \bar{\alpha}\}$
9. $\mathit{nil}[S] \equiv \mathit{nil}$
10. $(B + B')[S] \equiv B[S] + B'[S]$
11. $(\sigma.B)[S] \equiv S(\sigma).B[S]$

Composition is less amenable to proofs of strong equivalence because more of the syntactic form of a composite expression is carried over to the result of a transition than is the case with an expression such as a summation. It is, however, clear that $\mathit{nil} \mid b.\mathit{nil} \equiv b.\mathit{nil} \mid \mathit{nil}$ and $\mathit{nil} \mid \mathit{nil} \equiv \mathit{nil}$. It is also possible to prove the following equivalence expressing the composition of certain sums as the sum of guarded compositions. Let $\sum\{B_i : i \in I\}$ designate the summation of expressions B_i for all i in some finite index set I. (That this notation is unambiguous modulo strong equivalence follows from the commutative and associative laws for summation stated above.) Each B_i is a *summand*. The sum is a *sum of guards* if each summand has the form $\sigma_i.B_i'$. Let B and C be sums of guards. Then

$$\begin{aligned} B \mid C \quad \equiv \quad & \textstyle\sum\{\sigma.(B' \mid C) : \sigma.B' \text{ is a summand of } B\} \\ & + \textstyle\sum\{\sigma.(B \mid C') : \sigma.C' \text{ is a summand of } C\} \\ & + \textstyle\sum\{\tau.(B' \mid C') : \sigma.B' \text{ is a summand of } B \text{ and } \bar{\sigma}.C' \text{ is a summand of } C\} \\ & + \textstyle\sum\{\tau.(B' \mid C') : \bar{\sigma}.B' \text{ is a summand of } B \text{ and } \sigma.C' \text{ is a summand of } C\} \end{aligned} \tag{3.1}$$

Equations such as these are the beginnings of an algebra for communicating systems. An algebra suitable for object-based programming would address issues of software reuse and substitutability by demonstrating that two modules exhibited the same external behavior or by showing that a proposed module meets a specification for desired behavior. But CCS provides the most limited possibility for defining program modules and no capacity for creating new agents dynamically. We need a more flexible scheme for representing interacting agents. The actor and Abacus models described in the next two chapters attempt to address these shortcomings of CCS, thereby creating more useful models of a concurrent, object-based environment.

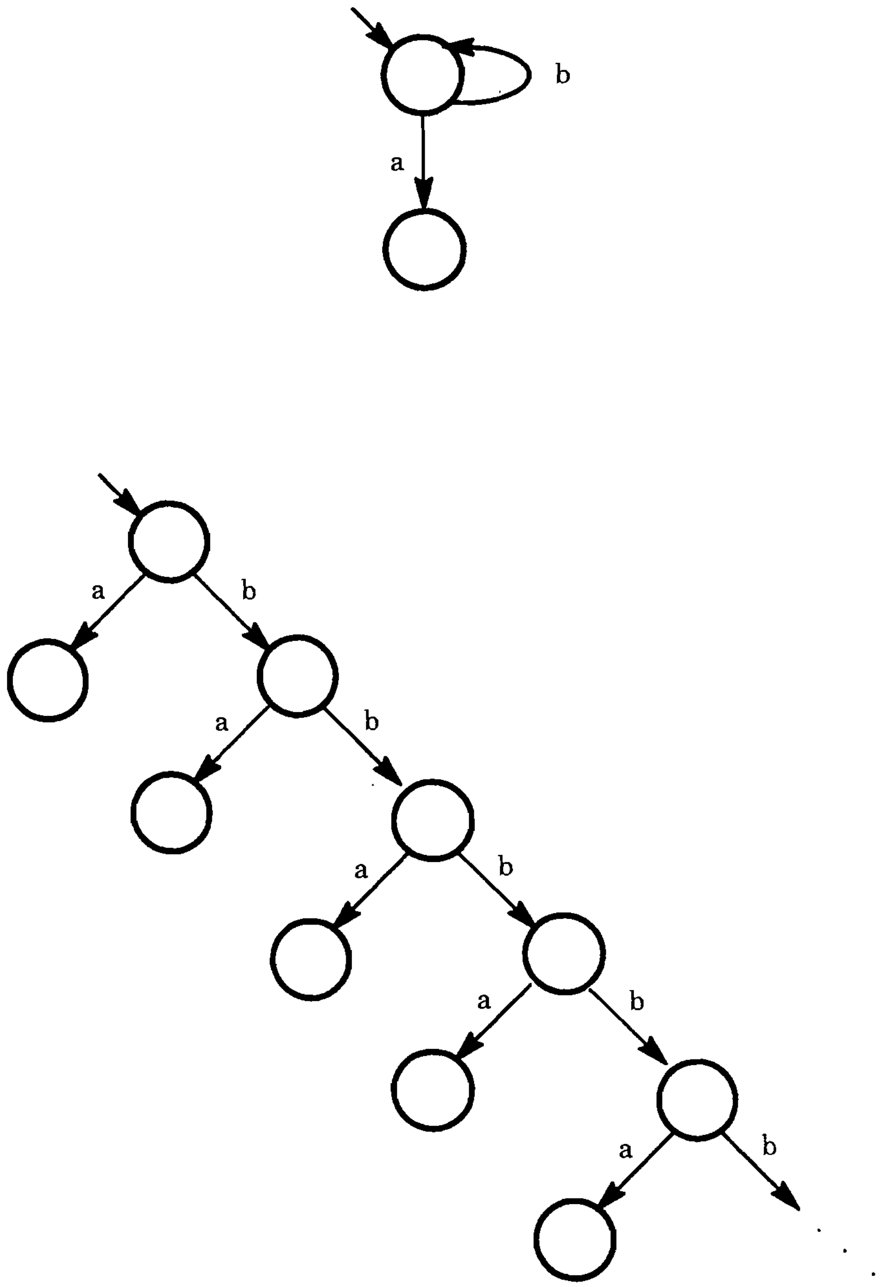

Figure 3.8: Transition systems for a recursively defined system

$$\begin{tikzcd}
 & & (a.b.nil + b.nil)|\bar{a}.b.nil \arrow[dll, "b"'] \arrow[dl, "a"'] \arrow[d, "\tau"] \arrow[drr, "\bar{a}"] & & \\
nil|\bar{a}.b.nil \arrow[drr, "\bar{a}"'] & b.nil|\bar{a}.b.nil \arrow[l, "b"'] \arrow[r, "\bar{a}"] & b.nil|b.nil \arrow[d, "b"'] \arrow[dr, "b"] & & (a.b.nil + b.nil)|b.nil \arrow[ll, "a"'] \arrow[d, "b"] \\
 & & nil|b.nil \arrow[dr, "b"'] & b.nil|nil \arrow[d, "b"] & (a.b.nil + b.nil)|nil \arrow[l, "a"'] \arrow[dl, "b"] \\
 & & & nil|nil &
\end{tikzcd}$$

Figure 3.9: Transition system for the CCS expression $(a.b.nil + b.nil)|\bar{a}.b.nil$

Chapter 4

Actors

The actor model of computation was originally developed as a means of expressing concurrency [25,24], but the current incarnation of the model, described in Agha's work [1,2,3] shares the defining characteristic of an object-based system, namely, that of encapsulating parts of a system's state to enhance data integrity and modularity.

4.1 Basic Concepts

An *actor* is a structure with a fixed mail address at which it can receive messages from other actors and with a current *behavior*. It can respond to the messages that it receives by sending further messages. creating new actors, or specifying a new behavior for itself (its *replacement behavior*). Messages sent by an actor can only go to its acquaintances, that finite group of actors whose mail addresses are known to the sender. When an actor adopts its replacement behavior, it continues to receive mail at its original mail address. Only the actor's response to those messages may change.

The system in which the actors are embedded is expected to provide support for the transmission and receipt of mail, and, indeed, to guarantee the delivery of all mail that is sent. In order to assure delivery, even when an actor is occupied with other tasks, incoming messages for the actor may be buffered in a mail queue.

The combination of an actor and its mail queue is reminiscent of an automaton reading a tape of input symbols (see Figure 4.1), but there are important differences. First, while the replacement behavior of an actor is analogous to the next state of the automaton, there is no limit on the number of distinct behaviors that an actor may exhibit in its lifetime, so that the actor is more like an infinite state automaton.

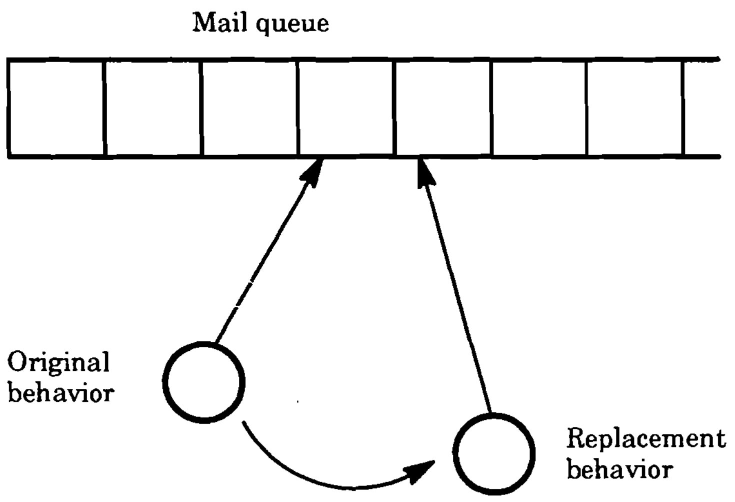

Figure 4.1: An actor and its mail queue

```
Define Factorial with acquaintances self
    let communication be integer n and customer u
    become Factorial
    if n = 0
        then send 1 to customer u
        else let c = Customer with acquaintances n and u
             send n-1 and mail address of c to self
Define Customer with acquaintances integer n and customer u
    let communication be integer k
        send n*k to u
```

Figure 4.2: Actors to compute factorials

In addition, there is no named location where an actor's state is stored. The creation of an actor may be parameterized (as we shall see in the example below), but the actor's state is merely implicit in the pattern of behaviors that it computes. This provides the ultimate in encapsulation of objects.

Finally, while an actor can "write" on its own "input tape," it can do so only through the mechanism of the mail system. It can send itself messages just as any other actor acquainted with its mail address could.

Actors also bear some similarity to Milner's Calculus of Communicating Systems (CCS) [33], where, as we have seen, a process can evolve in any of a number of directions depending on communications received. Actor systems, however, are more dynamic in that new actors can be created at run time, whereas CCS uses a fixed set of processes. Communication protocols also differ in that actors communicate asynchronously, in contrast to the synchronous message passing in CCS.

4.2 An Example

The example of an actor system given in Figure 4.2 (adapted from Agha's book [2]) is designed to compute factorials, and illustrates many of the ideas of the actor model. This declaration describes two sorts of actors, a Factorial actor which manages the computation of an integer factorial and a Customer actor on whose behalf some factorial is computed. Computation begins when an actor u sends a message containing an integer n to a Factorial actor. The Factorial actor specifies that its replacement behavior will also be a Factorial actor in the statement *become Factorial.* It then sends one of two messages, depending on the value of n. If n is zero, then the value of $n!$ is 1, which is thus sent to the actor u. If n is non-zero, then a new Customer actor is created to receive the results of the computation of $(n-1)!$. The latter computation is initiated by sending an appropriate message to *self*, which is bound to the mail address of the original factorial actor. Since the replacement behavior for that actor is also a Factorial, the correct outcome can be expected.

The Factorial actor is capable of managing several computations at once. At any time, its mail queue might contain messages associated with many strands of computation. The creation of Customer actors ensures that the results of each computation are ultimately delivered to the right place.

A Customer actor is created with two acquaintances (aside from itself), an integer n and another actor u which needs the value of $n!$. Upon receiving a message containing the integer k (which should be the value of $(n-1)!$), the Customer actor passes on the value of the product $n * k$ to the actor u. No replacement behavior for a Customer is given explicitly, so the implicit behavior is the *undefined behavior* that generates no new messages and creates no new actors in response to all incoming messages. An implementation of the actor model could reclaim the memory occupied by a Customer actor as soon as it sends its message.

Although the syntax by which integer operations are expressed here is different from that used for operations with the Factorial and Customer actors, there is no reason not to regard all constructs in the actor model as actors. That is, use of the traditional infix notation for integer operations is just a syntactic convenience; we might just as well suppose that $n - 1$ is computed by sending the actor n a message to subtract 1 from its integer value and return the result. This sort of homogeneity in a model is often a great convenience in developing the mathematics of the model. Of course, an implementation of the model will have to regard some actors (such as the natural numbers) as *primitive actors* whose computations are completed without further resort to message passing, other than to communicate the result.

Notice that the actor u which initiates the factorial computation might be any sort of actor, just so long as it has the address of a Factorial actor. Also, the actors involved in one of these computations are completely encapsulated from the rest of the environment. The Customer actors created for the computation are known only to the Factorial actors that create them and the other Customer actors from which they will get a partial result. Their mail addresses are unknown elsewhere, so there is no possibility of any other actors sending them messages. Only the Factorial actor interacts directly with the rest of the system, thus acting as a *receptionist* for the Factorial component of the system.

4.3 Actor Semantics

One can construct a transition system to represent the computation undertaken by a collection of actors. The basic components of the transition system are not difficult to construct, though there are some technical hurdles to overcome.

First, we must develop an adequate description of a state or configuration of an actor system. At any time, there is an existing set of actors, each with some number of unprocessed messages in its mail queue. Each actor is given by a mail address and a behavior, while each message is associated with a specific mail address and has contents consisting of a tuple of values (actors' addresses). Of course, it is possible for an actor to receive more than one message with the same contents, so we must provide each message with a unique *tag* to distinguish it.

Without being specific yet about the nature of some of these constructs, let us use the following symbols to represent the indicated sets:

$\mathcal{B}$	Behaviors
$\mathcal{M}$	Mail addresses
$\mathcal{K}$	Communications (message contents)
$\mathcal{I}$	Tags for messages
$\mathcal{A} = \mathcal{M} \times \mathcal{B}$	Actors
$\mathcal{T} = \mathcal{I} \times \mathcal{M} \times \mathcal{K}$	Tasks (unprocessed messages)

Recall now the informal description of an actor, which, in response to a message, may send messages and create new actors, and must specify a replacement behavior. More formally, then, the behavior of an actor with address m is a mapping from messages with address m to tuples consisting of a finite set of tasks, a finite set of actors, and a replacement behavior at address m. We write this as

$$\mathcal{B} = (\mathcal{I} \times \mathcal{K} \rightarrow F_s(\mathcal{T}) \times F_s(\mathcal{A}) \times \mathcal{A})$$

While this equation conveys symbolically the basic idea of what will constitute a behavior in the actor semantics, it is an apparently circular definition because the actors on the right side of the equation are themselves defined in terms of behaviors.

4.3.1 Behavior Semantics

Construction of a rigorous semantics for behaviors can follow the pattern exemplified by the description of CCS in section 3.4. That is, we can recursively describe a syntax for behavior specifications that builds complex specifications from primitive behaviors using a small collection of syntactic operators. The semantics is then obtained by giving the semantics of the primitive behaviors, and showing how the semantics of a complex behavior specification is derived from the semantics of the syntactic components of the complex specification.

The choice of primitive components and syntactic operators can vary somewhat while still retaining the spirit of the actor model of computation. Agha [2] builds a model whose primitives include integer and Boolean expressions, using the usual range of integer and Boolean operators, and mail expressions, which consist either of a mail identifier or an expression *new* $E(e_1, \ldots, e_i)$, which describes the creation of a mail address for a new actor with behavior E and acquaintances $e_1, \ldots, e_i$.

A behavior description has the following form:

$$\text{def } E(p_1, \ldots, p_i)[p'_1, \ldots, p'_j]S \text{ enddef}$$

The parameters $p_1, \ldots, p_i$ are *acquaintance parameters* that will be bound at the time an actor is created or replaces its behavior to the mail addresses of other actors in the environment. The parameters $p'_1, \ldots, p'_j$ are *communication parameters* that will be bound to values at the time an actor begins to process a message. We shall not specify a syntax for associating the communication parameters with specific components of a communication, but several mechanisms are possible, including pattern matching or the use of keywords. The identifier S in the behavior definition indicates a *command* of a form described below.

In giving the syntax of a command, we will assume that S, S_1, and S_2 are commands; b is a Boolean expression; $e_1, e_2, \ldots$ are arbitrary expressions (integer, Boolean, or mail); $a, a_1, \ldots, a_j$ are mail address identifiers; and $E, E_1, \ldots, E_j$ are behavior identifiers. Then a command has one of the following forms:

concurrent	$S_1 // S_2$
conditional	if b then S_1 else S_2 fi
communication	send $[e_1, \ldots, e_i]$ to a
replacement	become $E(e_1, \ldots, e_i)$
creation	let $a_1 =$ new $E_1(e_{1,1}, \ldots, e_{1,i_1})$ and ... and $a_j =$ new $E_j(e_{j,1}, \ldots, e_{j,i_j})\{S\}$

In this syntax, the factorial example given above would be expressed as in Figure 4.3.

Recall now that we are trying to describe a behavior as a mapping from tagged communications to 3-tuples consisting of a finite set of new tasks, a finite set of new actors, and a single replacement behavior. The tagged communication, through its binding of a behavior's communication parameters, establishes the environment in which the behavior's commands can be interpreted. The environment determines, for example, the value of the boolean expressions in any conditional commands, or the values of the acquaintance arguments in a replacement command. Also part of the local environment at run time are the mail address of the actor that is processing the communication

```
def Factorial (self) [n, u]
    become Factorial (self) //
    if n = 0
        then send [1] to u
        else let c = new Customer (n,u)
             {send [n-1,c] to self}
enddef
def Customer (n,u) [k]
    send [n*k] to u
enddef
```

Figure 4.3: Another description of actors to compute factorials

(customarily denoted by *self*, as in the factorial example) and the current tag *curr* (initially, the tag of the task being processed), which is used to generate the mail addresses of new actors and tags of new tasks.

We will now describe informally the 3-tuples associated with each form of command:

Concurrent: The 3-tuple associated with a concurrent command is constructed by forming the unions of the sets of new tasks and new actors generated by the concurrent components S_1 and S_2. The replacement behavior is that of S_1 or S_2, if either specifies a replacement. If both do, then the concurrent command is semantically (and syntactically) invalid.

Conditional: If the Boolean expression b evaluates to *true* in the local environment, then the 3-tuple associated with the conditional is precisely that associated with S_1. Otherwise, it is the 3-tuple associated with S_2.

Communication: This command generates a new task for the actor at the mail address denoted by a. The task's contents are obtained by evaluating $e_1, \ldots, e_i$ in the local environment, while the task's tag is generated from the current value of *curr*. No new actors and no replacement behavior arises from this command.

Replacement: The replacement command generates no new tasks or actors. As its name suggests, it specifies a replacement behavior, identified by E, whose acquaintance parameters are obtained by evaluating $e_1, \ldots, e_i$ in the local environment.

Creation: This command dictates the creation of new actors with the specified behaviors, whose mail addresses are bound to the identifiers $a_1, \ldots, a_j$. The 3-tuple associated with this command consists of these new actors, together with the new tasks, new actors, and any replacement behavior associated with S, the command S being considered in an environment that consists of the environment of the creation command, as amended by the bindings of $a_1, \ldots, a_j$.

This completes our description of a semantics for an actor's behavior, except for saying a few words about the creation of new tags and mail addresses. If behavior is to be a well-defined mapping of tasks and actors are to be identified by their mail addresses, then care must be taken that distinct tasks are indeed distinguished by their tags and that no two actors have the same address. Furthermore, if an actor system is to exploit its potential concurrency fully, then we must allow each actor to generate new tags and addresses independently, yet with the assurance that they are distinct from all other tags and addresses.

Agha's scheme is to represent tags and addresses by sequences of natural numbers. When an actor is dealing with a task with tag t, new tags and addresses are created by appending another natural number to the sequence represented by t. This is done in such a way that at no time is any tag or address in the system a prefix of another tag or address (assuming that this condition is satisfied initially), so that subsequent evolution of the system cannot lead to duplication of tags or addresses. For our purposes, it is enough to know that the semantics is well-defined.

4.3.2 Transition Systems for Actor Programs

Now that we have pinned down the nature of behavior in an actor system, we can complete our description of a transition system for an actor program. Define a *local states function*

$$l : \mathcal{M} \to \mathcal{B}$$

such that, for each $m \in \mathcal{M}$, $l(m)$ is the current behavior of the actor with mail address m. We can regard l as a partial function defined only on the mail addresses corresponding to existing actors, or let its value on unused mail addresses be the *undefined behavior* $\beta_\perp$. The local states function has an equivalent representation as the set of actors currently in the system, each actor being an ordered pair (m, β) of mail address and current behavior.

A *configuration* of an actor system can then be given by a pair (l, T), where l is a local states function and T is a finite set of tasks (i.e., those yet to be processed). These configurations will be the states of our transition system for an actor program.

To determine the initial state of the transition system, consider a typical actor program, whose syntax is given by

$$D_1 \ldots D_n S$$

Here, each D_i is a behavior definition and S is a command. For example, in a program to compute

15!, D_1 and D_2 could be the behavior definitions for Factorial and Customer given above, while S could be

```
    let f = new Factorial (f) {send [15] to f}
```

We can analyze S just as we did commands in behavior definitions, obtaining an initial set T_0 of tasks to be processed and an initial set $\{(m_1, \beta_1), \ldots, (m_k, \beta_k)\}$ of actors, which implicitly defines an initial local states function l_0. There is no need or sense in S specifying any replacement behavior, since S is not executed by an actor.

In order to describe transitions more easily, define the functions *states* and *tasks* on configurations as follows: For any configuration (l, T), let states$(l, T) = l$ and tasks$(l, T) = T$. For any configuration c_1 of an actor system, one can show that the possible transitions from c_1 correspond to processing any one of the tasks in c_1. So, if $\sigma = (t, m, k) \in$ tasks(c_1) and states$(c_1)(m) = \beta$, where $\beta(t, k) = (T, A, \gamma)$, then there is a configuration c_2 such that

$$\text{tasks}(c_2) = (\text{tasks}(c_1) - \{\sigma\}) \cup T$$

and

$$\text{states}(c_2) = (\text{states}(c_1) - \{(m, \beta)\}) \cup A \cup \{\gamma\}$$

As usual, we write $c_1 \xrightarrow{\sigma} c_2$ to denote this transition. This completes the construction of an operational semantics for actor systems.

4.4 Other Aspects of the Actor Model

We have seen that the actor model of computation provides for strong encapsulation of objects with maximal exploitation of concurrency both between objects and within them. Computation is carried out almost entirely through asynchronous message passing. We have, however, not yet considered the prospects for sharing or reusing code in an actor system.

4.4.1 Actor Modules

The very effective encapsulation of individual actors can be extended to collections of cooperating actors through the designation of *receptionists* for the collection. The receptionists for a program module are actors whose addresses are made known outside the module, and through which all messages requesting the module's services are sent. For example, the Factorial actor of section 4.2

could serve as the receptionist for a factorial-computing module consisting of the Factorial actor and the Customer actors that it creates.

From the viewpoint of an external observer, an actor module is as thoroughly encapsulated as an individual actor. Indeed, there is no evidence available to an actor outside the module that the module consists of anything more than its receptionists. Transitions of the module corresponding to intra-module communication are invisible actions of the module, as far as an external observer is concerned.

The rudimentary actor language described in section 4.3 provides no support for the definition and reuse of modules of actors, but one could build on that language to create mechanisms for module definition. For example, if, in the factorial example, the scope of the behavior identifier *Customer* were restricted so that only Factorial actors could meaningfully create Customer actors, then a greater degree of modularity would be achieved.

Of course, the dynamic nature of actor systems—specifically, the possibility of communicating mail addresses among actors—poses some risk to the intended encapsulation of an actor module. As the system runs, the address of an actor that had not previously served as a receptionist may become known outside a module. Then messages can be sent to the module through this new acquaintance of actors outside the module.

4.4.2 Sharing Among Actors

The basic actor model makes no provision for any form of sharing among actors, though its flexibility creates the means for implementing different forms of sharing by building on the foundation of the basic model. Static type checking of behavior definitions is a possibility, as is dynamic checking of newly created actors for conformance to the parameterization of their initial behaviors.

There has been an effort to implement a system of classification and inheritance within the actor model, but that dates back to 1981 and the description system Omega [5]. No more recent developments are referred to in the literature. The very flexibility of actors may inhibit attempts to enforce uniformity of behavior across a large collection of actors. The actors in the factorial example were very conservative in their choice of replacement behaviors, opting either to retain the current behavior or, in effect, to shut down. But there is no *a priori* restriction on an actor's choice of replacement behavior.

Actors seem better suited to share information by means of *delegation*, in which one actor would be the repository for certain code and other actors needing to use the code would delegate the task of executing it on their behalf to that one actor. This would ensure that similar subcomputations needed by a variety of actors would be implemented uniformly throughout the system and could be modified readily if necessary.

Even delegation would require some system support beyond what is supplied in the basic actor model. To be specific, delegation differs from ordinary message passing in its use of references to *self*. The delegated computation is to be carried out just as if the code to be executed resided with the actor making the delegation. So the use of *self* in the delegated computation must be understood as a reference to the delegating actor, not to the recipient of the delegation.

In short, then, the actors model succeeds brilliantly in creating active objects that can operate independently and securely in a highly concurrent environment. But these very properties can complicate efforts to share or reuse code systematically.

Chapter 5

Abacus

In a series of papers [35,37,39,41], Nierstrasz and his colleagues address some of the deficiencies of both CCS and actors as models for object-oriented systems. They adopt much of the syntax and semantics of CCS, but extend it in two important ways. First, to endow CCS with some of the dynamic flexibility of actors, they provide for agents that can be parameterized and allow agent *patterns* to be passed in inter-agent messages. Second, to permit encapsulation of larger modules than a single object, they augment the syntax of CCS with prefixes for labels and filters based on those prefixes, which, together, constrain communication in ways not available within the basic CCS or actors models.

5.1 Syntax and Semantics

The development of the syntax and semantics of Nierstrasz's Abacus model follows the pattern that we have seen exemplified with both CCS and actors. The Abacus notation is built recursively from simpler patterns and identifiers using a small number of operators. Abacus *terms* are Abacus patterns with no free variables. Transition rules defined on terms are used to describe an operational semantics for Abacus in which each term is associated with a labelled transition system.

We shall describe this development of semantics for Abacus briefly, with emphasis on the ways in which Abacus departs from the other models that we have studied. It is important, however, to have a detailed description of the semantics in order to understand the differences among these models.

Behavior patterns in Abacus are built from a set L of *labels*, a set X of *label prefixes*, a set N of *simple agent names*, and a set V of *pattern variables*. Variables can be differentiated syntactically

Type	Form, y	Free variables, $f(y)$
Inaction	nil	$\emptyset$
Named agent	a_v	$v(a_v)$
Output	$e_v!p$	$v(e_v) \cup f(p)$
Input	$e_v?p$	$f(p) - (v(e_v) - \{\_\})$
Choice	$p + q$	$f(p) \cup f(q)$
Composition	$p\&q$	$f(p) \cup f(q)$
Restriction	$p \setminus e_v$	$f(p) \cup (v(e_v) - \{\_\})$
Relabelling	$p/[f_v]$	$f(p) \cup v(f_v)$
Prefixing	$x_v : p$	$f(p) \cup v(x_v)$
Filtering	$p\backslash : x_v$	$f(p) \cup v(x_v)$

Figure 5.1: Syntax for Abacus expressions

from the other elements (so that they can be recognized as variables in any context) and include a distinguished element "_" called the *anonymous variable.* Use of the anonymous variable provides some additional flexibility in describing patterns, at the cost of somewhat more complicated rules for determining the free variables in an expression.

These atomic elements of syntax can be combined to form the *event patterns* E_v, *relabelling patterns* F_v, and *pattern names* A_v, as follows:

- $E_v ::= L|V|X_v : E_v|[R_v]$
- $X_v ::= X|V$
- $R_v ::= E_v|E_v, R_v$
- $F_v ::= E_v/E_v|E_v/E_v, F_v$
- $A_v ::= N|N(R_v)$

Each type of pattern has an associated *closed form* in which all variables are bound to values. Descriptions of the closed forms (E, X, R, F, and A) can be obtained by systematically removing the subscripts from the definitions given above.

To help complete our description of behavior patterns, let v and f be functions giving, respectively, the variables and the free variables in an expression. For any expression y of a sort described above, $v(y)$ and $f(y)$ both consist of all elements of V appearing in y.

Now let e_v, r_v, x_v, f_v, a_v, and n represent arbitrary elements of E_v, R_v, X_v, F_v, A_v, and N, respectively, and let p and q represent any two pattern expressions. Then the table in Figure Abacustab shows the different forms that a pattern expression may take, along with the free variables in each such expression.

Recursion in Abacus is provided by *pattern declarations* of the form

$$a_v := p$$

The free variables in this declaration are $f(p) - (v(a_v) - \{\_\})$. An object environment is described in Abacus by a set D of pattern declarations, which may be mutually recursive.

As we did with CCS, we can construct a labelled transition system for Abacus whose states are Abacus terms and whose label alphabet consists of

$$\{e?|e \in E\} \cup \{e!|e \in E\} \cup \{\tau\}$$

The symbol τ represents an invisible transition, as before.

Note that, unlike CCS, the elements of the label alphabet for Abacus are endowed with structure because of the way that E is defined. We will use that structure to be much more specific with Abacus than we were with actors about how values in incoming messages are bound to an agent's variable parameters. Briefly, expressions match only if they are structurally similar. In that case, the matching implicitly defines a binding of values to variables, provided that each variable occurs at most once. The presence of the anonymous variable may facilitate matching of two expressions, but has no effect on the resulting binding.

To lend some precision to this description of matching and binding, let

$$M ::= E|X|R|A \quad \text{and} \quad M_v ::= E_v|X_v|R_v|A_v$$

Matching and substitution in a behavior expression is indicated by

$$p\{m/m_v\}$$

where p is a behavior pattern and the expressions m and m_v are in M and M_v, respectively. The matching of m with m_v and the substitution into p based on the binding determined by the matching is defined only if the rules given below can be applied to complete the substitution. The rules define the substitution by structural induction on the matched expressions and on the behavior expression.

- $p\{x : e/x_v : e_v\} \equiv p\{x/x_v\}\{e/e_v\}$
- $p\{n(r)/n(r_v)\} \equiv p\{r/r_v\}$
- $p\{[r]/[r_v]\} \equiv p\{r/r_v\}$
- $p\{e, r/e_v, r_v\} \equiv p\{e/e_v\}\{r/r_v\}$

- $p\{m/m\} \equiv p$ for any $m \in M$
- $p\{e/\_\} \equiv p$
- $(p \text{ op } q)\{\sigma\} \equiv p\{\sigma\} \text{ op } q\{\sigma\}$, where *op* could be any member of {!, ?, +, &, \ , :, \:}, provided that p and q are constrained appropriately so that p op q is a valid behavior expression, and σ is any substitution m/m_v.
- $p/[f_v]\{\sigma\} \equiv p\{\sigma\}/[f_v\{\sigma\}]$
- A substitution is applied separately to the components of event or relabelling patterns, just as it is applied to components of behavior patterns. For example, $(x_v : e_v)\{\sigma\} \equiv x_v\{\sigma\} : e_v\{\sigma\}$.
- $v\{e/v'\} \equiv \begin{cases} e & \text{if } v = v' \text{ and } v \neq \_ \\ v & \text{otherwise} \end{cases}$
- $m\{\sigma\} \equiv m$ for any $m \in M$ and any substitution σ
- $\text{nil}\{\sigma\} \equiv \text{nil}$

Finally, we are in a position to give the transition rules on Abacus terms. Many are like those for CCS, but there are some significant differences.

Output.

$$\frac{}{e!p \xrightarrow{e!} p}$$

Input.

$$e_v?p \xrightarrow{e?} p\{e/e_v\}$$

Choice.

$$\frac{p \xrightarrow{\alpha} p', \alpha \neq \tau}{p + q \xrightarrow{\alpha} p'} \qquad \frac{p \xrightarrow{\alpha} p', \alpha \neq \tau}{q + p \xrightarrow{\alpha} p'}$$

$$\frac{p \xrightarrow{\tau} p'}{p + q \xrightarrow{\tau} p' + q} \qquad \frac{p \xrightarrow{\tau} p'}{q + p \xrightarrow{\tau} q + p'}$$

Composition.

$$\frac{p \xrightarrow{\alpha} p'}{p\&q \xrightarrow{\alpha} p'\&q} \qquad \frac{p \xrightarrow{\alpha} p'}{q\&p \xrightarrow{\alpha} q\&p'}$$

$$\frac{p \xrightarrow{e?} p', q \xrightarrow{e!} q'}{p\&q \xrightarrow{\tau} p'\&q'} \qquad \frac{p \xrightarrow{e?} p', q \xrightarrow{e!} q'}{q\&p \xrightarrow{\tau} q'\&p}$$

Restriction.

$$\frac{p \xrightarrow{\alpha} p'}{p \setminus e_v \xrightarrow{\alpha} p' \setminus e_v} \text{ if } \alpha = \tau, \text{ or } \alpha \in \{e?, e!\} \text{ and } e/e_v \text{ is not a valid substitution}$$

Relabelling.

$$\frac{p \xrightarrow{\alpha} p'}{p[f] \xrightarrow{f(\alpha)} p'[f]}$$

Prefixing.

$$\frac{p \xrightarrow{\alpha} p', \alpha \neq \tau}{x : p \xrightarrow{x:\alpha} x : p'} \qquad \frac{p \xrightarrow{\tau} p'}{x : p \xrightarrow{\tau} x : p'}$$

Filtering.

$$\frac{p \xrightarrow{x:\alpha} p'}{p\setminus : x \xrightarrow{\alpha} p'\setminus : x} \qquad \frac{p \xrightarrow{x:\alpha} p', x \neq y}{p\setminus : y \xrightarrow{x:\alpha} p'\setminus : y}$$

Recursion.

$$\frac{a_v := p, p\{a/a_v\} \xrightarrow{\alpha} p'}{a \xrightarrow{\alpha} p'}$$

5.2 Abacus Semantics vs. CCS Semantics

With the semantic rules for Abacus in hand, we can now compare the syntax and semantics of Abacus to that of CCS and actors. The action rule for CCS has been replaced by the input and output rules for Abacus, with a minor change in syntax ($e?$ and $e!$ instead of a and $\bar{a}$ for the actions). The input rule shows an incoming message e matching with an agent's input offer e_v, thereby binding the variables in e_v to values in the message. If the message and offer do not match, then the substitution is undefined and no transition on e exists. Notice too, in this context, that the anonymous variable can occur in Abacus terms only in input offers, for an occurrence anywhere else would be free.

One of the more significant departures of Abacus from CCS occurs in the choice operation. Comparing the transition rules for Abacus to those for CCS summation presented in section 3.4.2, we see that an inference such as $p + q \xrightarrow{\tau} p'$ from $p \xrightarrow{\tau} p'$ can be made in CCS but not in Abacus. That is, Abacus will not allow the choice between two offered agents to be made internally or invisibly. Either agent can itself evolve through invisible transitions, but the choice must remain until one agent is selected through acceptance of a visible action. It turns out that weakening the choice operator in this way does not lessen the expressive power of the language, and even offers the theoretical advantage that the strong and observational equivalences are congruences in Abacus, though they are not in CCS.

Rules for composition and relabelling are the same in CCS and Abacus. The rules for recursion differ only in reflecting the different syntax used for recursive definitions in the two languages. Likewise, the rule for restriction is different in Abacus principally in recognition of the more complex syntactic structure of events.

Notice, though, the role that variables can play in recursion and restriction. The only unbound variable that can appear in a restriction is the anonymous variable. Thus, we can specify that certain event patterns do not allow an agent to evolve. Recursion is, with input, one of only two pattern types in which ordinary (i.e., non-anonymous) variables can occur in Abacus terms. Furthermore, non-anonymous variables occurring in the agent name a_v are never free variables of the recursion expression, so (ironically) they may be used freely. This is the mechanism by which agents can be parameterized. As a very simple example, consider the following class of agents which accept a single input:

$$in1(X) := X?nil$$

where $X \in V$. According to the semantics, for any event e, there is an agent $in1(e)$ having as its sole transition

$$in1(e) \xrightarrow{e?} nil$$

The expanded possibilities for encapsulation in Abacus are evident in the rules for prefixing and filtering. The prefixing rule indicates that prefixed agents can evolve only through similarly prefixed events, or through invisible actions. In doing so, they retain their prefix. (It is worth emphasizing here the distinction between a prefix on a behavior expression, such as $x : p$, and a prefix on an event expression which forms an input or output offer, as in $x : e!q$. In the former case, the behavior prefix implicitly applies to all offers subsequently made by the agent p, as the semantic rules indicate, while in the latter case only the first offer need be prefixed.)

All the agents sharing a given prefix thus constitute a module through which communication can be controlled. An output offer in Abacus is effectively broadcast in that it can be accepted by any agent in the environment that is prepared with a matching input offer. But the use of prefixes provides a means of restricting the targets of output offers, ensuring that they can only be accepted by agents bearing the same prefix.

Filtering plays a role that is complementary, in different ways, to both restriction and prefixing. Where restriction allows one to specify the transitions to be hidden, a filter provides a description of transitions to be revealed. To be specific, there are three relevant categories of transition. A filtered agent will not expose any unprefixed transitions; it will reveal unchanged any transitions bearing a prefix that is not named in the filter; and if a transition's prefix is the one named in the filter, then the transition will be revealed after stripping the prefix.

As an example of the interaction between filtering and prefixing, consider the following agent:

$$(x : a!p + y : b!q + c!r)\backslash : x \ \&\ y : (a?p' + b?q' + c?r')$$

Either of the prefixed output offers in the first concurrent component can pass the filter, with $x : a!$ passing simply as $a!$ or $y : b!$ passing unchanged. But only the latter can be accepted by the input agents in the second component because of that component's prefix. So the composite agent can only evolve to

$$q\backslash : x \ \&\ y : q'$$

via an invisible transition. Wegner [47] describes the ***abstraction boundary*** of a module as the interface that it presents to its clients, and the ***distribution boundary*** as the "boundary of accessible names visible from within an object." In these terms, we see that we can control the abstraction boundary of a module by using prefixing and the distribution boundary by filtering.

5.3 Modeling CCS and Actors with Abacus

Using the operational semantics that has been defined for Abacus, we can define strong and observation equivalence for Abacus terms just as we did for CCS terms in section 3.4.3. Given the close connection between CCS and Abacus, it is then not surprising to learn that there is an embedding tr from CCS agents to Abacus agents that preserves strong bisimilarity of agents, so that for any CCS agents p and q, $p \sim q$ if and only if $tr(p) \sim tr(q)$. The converse is not true, so Abacus does have greater expressive power than CCS.

The embedding from CCS to Abacus is mostly just a rewriting of CCS notation into Abacus notation, but care must be taken to accommodate the changed semantics of summation. It is instructive to take a closer look at the mapping as an example of judicious prefixing and filtering.

Let CCS be defined, as in section 3.4.1 on a set X of variables and an alphabet Σ. Then we include X in the set of Abacus variables, and embed Σ in the Abacus label set so that, for any $a \in \Sigma$, a maps to $a?$, $\bar{a}$ to $a!$, and τ to τ. Except for summations, tr is defined as follows:

$$\begin{array}{ll}
tr(\text{nil}) \equiv \text{nil} & tr(p \mid q) \equiv tr(p) \& tr(q) \\
tr(x) \equiv x & tr(p \setminus a) \equiv tr(p) \setminus a \\
tr(a.p) \equiv a?tr(p) & tr(p[f]) \equiv tr(p)/[f] \\
tr(\bar{a}.p) \equiv a!tr(p) & tr(\text{rec } x.p) \equiv x := tr(p)
\end{array}$$

The difficulty with mapping a CCS summation is, of course, that it interacts differently with invisible transitions than does the corresponding operation in Abacus. We first isolate the invisible transitions by rewriting any CCS summation $p + q$ as

$$\sum_i \tau.p_i + \sum_j a_j.p_j$$

where $a_j \neq \tau$. This can be done using the rules of section 3.4.3, including especially equation 3.1 if p or q is itself a composition of terms. Then application of the mapping tr to this expression produces

$$((\sum_i k?x : tr(p_i) + x : \sum_j tr(a_j.p_j))\&k!nil)\backslash : x \tag{5.1}$$

If this Abacus agent takes any visible action, its offer will be prefixed internally by x. But the filter strips this prefix, so that externally the offers will appear as expected. Alternatively, any one of the first summands can accept the offer of the agent $k!nil$, reducing the expression to $(x : tr(p_i))\backslash : x$ for some i. In this way, the Abacus choice in expression 5.1 is accomplished by the visible offer and acceptance of k, but produces an agent whose behavior corresponds to that of a CCS summand that would have been selected invisibly in CCS. The filtering ensures that the offer of $k!$ can only be accepted internally to the agent of expression 5.1.

It is possible to extend the model of CCS presented in section 3.4.1 to allow CCS agents to pass values [33]. While the embedding is trickier to define, the relationship between CCS and Abacus is the same: Value-passing CCS can be mapped to Abacus so that strong equivalence of agents is preserved.

The roots of Abacus in CCS made it almost inevitable that Abacus could be used to model CCS. A more interesting and surprising result is that Abacus can be used to model Actors [39]. Since Abacus agents communicate synchronously, while actors communicate asynchronously, one must use several Abacus agents to model each actor and ensure that whenever mail is sent to an actor there is an agent prepared to accept it. Each actor is realized as a pair of agents, one to manage the actor's mail queue and one to represent the actor's behavior. That behavior, in turn, is achieved through the interaction of an environment agent to record the bindings of an actor's acquaintance and communication parameters, a handler agent that accepts and responds to the next message provided by the mail queue, and a cleanup agent that activates the replacement behavior at the appropriate time.

The Actor system as a whole is modelled by the actor agents, as described above, together with a factory agent that creates new actors, and a command agent to execute the top-level Actor program. Extensive use is made of prefixing and filtering to control the scope of name bindings during the actors' operations so that the agents that constitute a given actor interact properly with each other, without interfering with the agents constituting other actors.

Chapter 6

Maude

The two models for object-based environments that we have examined so far take as their foundation a representation of concurrency and add to it facilities for encapsulating collections of related objects and for sharing certain information among selected objects in the environment. Now we turn to a model whose roots are in a carefully constructed type theory, which is then extended to deal with concurrent operations. Specifically, we shall examine Meseguer's object-oriented system Maude and its semantics of concurrent rewriting [18,29,30]. A related but less fully developed theory can be found in the work of Mosses [34].

Despite the complementary approach taken by Meseguer, we will see much that is reminiscent of the development of the CCS, Actors, and Abacus models, and will have a chance to look at those models from another perspective. One of the selling points of Meseguer's theory is that it is quite general and subsumes, among other models of concurrency, Actors, CCS, and labelled transition systems.

The price of this generality is that the algebraic machinery developed along the way can be quite formidable. We'll briefly sketch the approach here, then get a taste of it by looking at the construction of initial algebra semantics for type specifications. Finally, we will tackle concurrent rewriting itself and look at its relationship to the other models that we have studied.

6.1 An Overview of Algebraic Semantics

Viewed from a sufficient remove, the development of Maude's semantics follows the pattern that we have seen before. A simple syntax consisting of selected constants and operation symbols is created, from which complex terms can be constructed. An operational semantics for the terms can be derived from fundamental operations associated with each syntactic construct by using carefully tailored rules of inference. Though we can then compute with the terms, they are generally unsuitable as semantic objects because they suggest unwanted distinctions between expressions that are meant to be semantically identical. To eliminate such distinctions, we equate certain terms through the creation of equivalence or congruence classes of terms. These classes, then, constitute the semantics of the original syntactic expressions. If the equivalence classes are properly constructed, the original operational semantics can still be applied in a well-defined fashion by using terms to represent the classes. (That is, from a given initial class, an operation will always lead us to the same destination class, even though the terms representing those classes may vary.)

In the cases of CCS, Abacus, and, to a lesser extent, Actors, the outcome of this process (the collection of equivalence classes) has been algebraic in nature, although the operational interpretation in the form of the transition systems that we have constructed has received more attention. In these models, the algebra emerges from our ability to represent an object-based environment using expressions built from constants, variables, and operators, and then to manipulate and simplify those expressions using algebraic laws arising from the equivalence relation on terms. The construction of a semantics for Maude is, by contrast, algebraic from the outset, relying on basic ideas from universal algebra and category theory.

This use of algebra permits the representation of "true" concurrency in the form of simultaneous rewriting or simplification of different parts of a given algebraic expression. By comparison, a labelled transition system only hints at the possibility of concurrency, when there is a nondeterministic choice of operations to be made from one of a number of concurrently operating processes. Ultimately, some one choice must be made, so that our retrospective view of a computation is always of a *sequence* of operations.

A Maude program is built from two types of modules, functional modules and system modules. Functional modules present the signature of an abstract data type by naming other types from which it is constructed, new operations available for that type, and properties of those operations (such as associativity, commutativity, or the satisfaction of certain equations). This signature determines a category of algebras all of which realize the syntactic features of the signature. Within that category, the preferred semantics is represented by the so-called initial algebra, which is, roughly speaking, the most abstract algebra meeting the specifications of the signature.

Functional modules appeared in the OBJ languages that were precursors to Maude [13]. System modules are an innovation introduced in Maude. They can contain the sort of information found in functional modules, as well as additional rules for rewriting expressions—rules that cannot safely be interpreted as equations. Such unidirectional rules make time and state factors in Maude programs, just as they play a role in reactive systems generally. This contrasts with the static type descriptions of the functional modules. The challenge presented by the system modules is to find a semantics for Maude that captures the rewriting suggested by these rules while subsuming the initial algebra semantics of the functional modules. Ultimately, the answer is still to use initial algebras, but in rather different categories from those associated with functional modules.

Because the unidirectional rules of system modules must be viewed as rewriting rules rather than equations, we shall treat equations as rewriting rules also in order to develop a unified approach. Both, then, become mappings in a suitably defined algebra, and, as mappings, generate a separate algebraic structure of their own on which an equivalence relation can be defined that captures the desired semantics. (When there are only equations, the algebraic structure of the mappings is isomorphic to the usual algebra of terms.) It turns out that this algebra of equivalence classes of mappings is initial in a certain category. That the objects of this category are themselves categories is perhaps an indication of the level of abstraction involved here.

Now that we have laid out the general framework for the development of the algebraic semantics we seek, we can delve into some of the details of the construction to get a clearer picture of the semantics.

6.2 The Semantics of Many-Sorted Signatures

We will begin our discussion of algebraic models of types by developing the terminology and machinery for discussing *many-sorted algebras.*

6.2.1 Syntax

Let S be a set, whose elements we will refer to as *sorts.* An *S-sorted set* is a collection of sets indexed by S, that is, in one-to-one correspondence with the elements of S. Typically, S will be finite, and the members of an S-sorted set will be sets such as the integers or the real numbers. If $\mathbf{A} = \{A_s | s \in S\}$ and $\mathbf{B} = \{B_s | s \in S\}$ are two S-sorted sets, then an *S-sorted mapping* between them is a collection of functions

$$\{f_s : A_s \rightarrow B_s | s \in S\}$$

indexed by S.

Now, let S^* denote the set of all (finite) words formed from an alphabet S of symbols. Then an *S-sorted signature* Σ is an $S^* \times S$-sorted set whose elements are denoted $\Sigma_{w,s}$ for w in S^* and s in

S. An element of $\Sigma_{w,s}$ is an *operation symbol*, or *function symbol*, of *arity* w and *sort* s. The *rank* of such a symbol is the pair (w, s). We will denote the empty string by λ. Operation symbols with arity λ are constant symbols. Other strings will be indicated by enclosing the constituent symbols within angled brackets.

As examples, we can construct signatures for the natural numbers and for a stack data type. These examples will be embellished as new concepts are introduced in this paper. More elaborate examples can be found in Ehrig and Mahr [12]. For the natural numbers, let the sort set consist of the symbol *nat*, and define a constant symbol 0 and unary function symbol *inc* as the sole members of $\Sigma_{w,nat}$ when w is λ and $< nat >$, respectively. For all other w, the set $\Sigma_{w,nat}$ is empty. A signature for a stack of (unspecified) elements can be built from sorts *elt*, *stack*, and *bool*, and the operation symbols *true*, *false*, *nil*, *push*, *pop*, *top*, and *empty*. *True* and *false* are constants of sort *bool*; *nil* is a constant of sort *stack*; *push* has rank $(< elt, stack >, stack)$; *pop* has rank $(< stack >, stack)$; *top* has rank $(< stack >, elt)$; and *empty* has rank $(< stack >, bool)$.

6.2.2 Many-Sorted Algebras

It is worth remembering that, while the names of the sorts and function symbols suggest a particular interpretation, this is just a suggestion, and that the meanings of the symbols used are not determined by the signature. Rather, the interpretation of the symbols is given when a Σ-algebra is defined. A *Σ-algebra* $\mathbf{A}$ for an S-sorted signature Σ is an S-sorted set $\{A_s | s \in S\}$ of *carrier sets* together with an actual function $\sigma_A : A_w \rightarrow A_s$ for each function symbol σ in $\Sigma_{w,s}$. Here, for any non-empty word $w =< s_1, ..., s_n >$ in S^*, the symbol A_w denotes the Cartesian product $A_{s_1} \times \ldots \times A_{s_n}$. When w is empty, A_w is a one-element set.

One interpretation of the first example signature given above is, of course, the natural numbers. That is, we can define a Σ-algebra $\mathbf{N}$, where N_{nat} is {0,1,...}, the function symbol 0 is interpreted as the numeral 0, and the function inc_N associated with the function symbol *inc* is defined by $inc_N(n) = n + 1$. But other interpretations are possible. For example, any directed graph G in which each vertex has outdegree 1 can be used to define a Σ-algebra $\mathbf{B}$ whose carrier set is the set of vertices of G. Let 0_B be any vertex, and let $inc_B(v) = w$ if (v, w) is a directed edge of G. Our assumption regarding the vertices' outdegrees ensures that inc_B is well defined.

These two algebras are related in ways other than simply being different interpretations of the same signature. Define a *Σ-homomorphism* from a Σ-algebra $\mathbf{A}$ to a Σ-algebra $\mathbf{B}$ to be an S-sorted mapping $f : \mathbf{A} \rightarrow \mathbf{B}$ such that, for each word $w =< s_1, \ldots, s_n >$ in $S*$, for each element $(a_1, \ldots, a_n)$

in A_w, for each sort s in S, and for each function symbol σ in $\Sigma_{w,s}$, the following equation is satisfied:

$$f_s(\sigma_A(a_1, \ldots, a_n)) = \sigma_B(f_{s_1}(a_1), \ldots, f_{s_n}(a_n)).$$

The concept of *category* will arise frequently in the discussion to follow. A category is a class of *objects* and *morphisms.* The morphisms are, in effect, labelled ordered pairs of objects, usually corresponding to the domain and co-domain of a mapping. An associative composition of morphisms is defined whenever the co-domain of one morphism matches the domain of another; the composition is itself a morphism. Each object has a distinguished morphism that acts as a left or right identity morphism whenever its composition with another morphism is defined.

It is easy to show that each Σ-algebra has an identity Σ-homomorphism whose components are the identity functions on the carrier sets of the algebra. Since the composition of Σ-homomorphisms is associative and always produces another Σ-homomorphism , the class of Σ-algebras for a given S-sorted signature Σ, together with their Σ-homomorphisms, forms a category of many-sorted algebras which we will denote $\mathbf{Alg}_\Sigma$.

Refer once again to our running examples. We can inductively define a Σ-homomorphism f from the natural numbers **N** to any of the graph-based algebras **B** by defining $f(0_N) = 0_B$, and, for any non-zero number $inc_N(n)$,

$$f(inc_N(n)) = inc_B(f(n))$$

The function f is a Σ-homomorphism by its very definition. Notice, too, that there is no other way to define a Σ-homomorphism from **N** to **B**. This is because each natural number can be represented uniquely by composing the functions 0_N and inc_N.

When dealing with data types, we want to maintain an abstract view of the type whenever possible. This means that we want to be able to use objects of that type without being concerned, or even aware, of how those objects are represented. This notion of abstraction is just the one captured algebraically by an *isomorphism.* A Σ-homomorphism $f : \mathbf{A} \rightarrow \mathbf{B}$ is a Σ-*isomorphism* if there is a Σ-homomorphism $g : \mathbf{B} \rightarrow \mathbf{A}$ such that the function compositions $f \circ g$ and $g \circ f$ are the identity homomorphisms on **B** and **A**, respectively. The mapping f is, in effect, just a renaming or change of representation of the elements of the algebra.

As with other algebras, one can define subalgebras and quotient algebras in the category of many-sorted algebras. Although these constructs underlie much of the theoretical development of algebraic type theory, we will not consider them in detail here, since our main concern is modeling object-based environments. Suffice it to say that abstract type theory benefits greatly because it can draw on the experience and, often, specific theorems developed over decades of research in abstract algebra.

6.2.3 Initial Algebras

Let $\mathcal{C}$ be a class of Σ-algebras for some S-sorted signature Σ. A Σ-algebra **A** is *initial* in $\mathcal{C}$ if **A** is a member of $\mathcal{C}$ and if for each Σ-algebra C in $\mathcal{C}$ there is exactly one Σ-homomorphism from **A** to C.

An easy proof shows that any two initial algebras in a class are isomorphic and that initial algebras have no proper subalgebras. What is a bit harder to do is to show that a given class of algebras actually has initial algebras. We will most often be concerned with the class of all Σ-algebras or the class of all Σ-algebras satisfying certain equations. In both of these cases, the existence of initial algebras can be shown using a term algebra construction that is reminiscent of the construction of Plotkin's approach to defining operational semantics for CSP. [44]

Before taking up the general construction, let us consider the case of the signature Σ given for natural numbers. That signature has one constant symbol 0 and one unary function symbol *inc*. The main idea in constructing a term algebra is to include elements in the algebra only as they are necessary to keep the algebra closed under the functions required by the signature. In this case, a Σ-algebra $\mathbf{T}_\Sigma$ must have an element corresponding to the constant symbol 0. We will call that element simply 0. To keep $\mathbf{T}_\Sigma$ closed under a function corresponding to the function symbol *inc*, we must have an element to serve as the image of 0 under inc_T. Call that element $inc(0)$. Again for closure, we now need an element $inc(inc(0))$, then $inc(inc(inc(0)))$, and so on. We take each new element to be distinct from all other elements of the algebra. Then $\mathbf{T}_\Sigma$ is a Σ-algebra with carrier set

$$\{0, inc(0), inc(inc(0)), ...\}$$

with constant $0_T = 0$, and with function inc_T defined by

$$inc_T(n) = inc(n)$$

for any element n in the carrier set. The equation defining inc_T is interpreted to mean that the result of applying inc_T to n can be found by replacing n in the right hand side of the equation by the term corresponding to n in the carrier set of $\mathbf{T}_\Sigma$. We recognize the resulting algebra as embodying a unary representation of the natural numbers in which a natural number n is given by the term having n occurrences of the function symbol *inc*. Binary or decimal notations or representations using trees are, of course, also possible, but all represent isomorphic Σ-algebras, and, as we will see, all are initial in the class of all Σ-algebras for this signature.

The general construction of a term algebra $\mathbf{T}_\Sigma$ for an arbitrary many-sorted signature Σ is complicated by the existence of more than one sort, but the process of beginning with constants and by stages including terms using other function symbols is the same as in the example above. To be precise, given an S-sorted signature Σ, define a term algebra $\mathbf{T}_\Sigma$ with carrier sets

$$\{T_{\Sigma,s} | s \in S\}$$

inductively as follows.

1. Let $\mathbf{T}_{\Sigma,s}\ (0) = \Sigma_{\lambda,s}$ for each s in S.
2. For $n > 0$, let $\mathbf{T}_{\Sigma,s}\ (n)$ be the union of $\mathbf{T}_{\Sigma,s}\ (n-1)$ and the set of all terms $\sigma(a_1, \ldots, a_r)$, where σ is in $\Sigma_{w,s}$ for some $w =< s_1, \ldots, s_r >$ and a_i is in $\mathbf{T}_{\Sigma,s}\ (n-1)$ for $i = 1, \ldots, r$.

Then $\mathbf{T}_{\Sigma,s}$ is the union of $\mathbf{T}_{\Sigma,s}\ (n)$ for $n = 0, 1, \ldots$.

Having said what the carrier sets are, we must now describe the functions corresponding to the function symbols in Σ . If σ is in $\Sigma_{\lambda,s}$, then σ_T is just the term σ . If σ is in $\Sigma_{w,s}$, then σ_T is defined so that, for any $(a_1, \ldots, a_n)$ in $\mathbf{T}_{\Sigma,w}$, the value of $\sigma_T(a_1, \ldots, a_n)$ is the term $\sigma(a_1, \ldots, a_n)$.

The algebra $\mathbf{T}_\Sigma$ has been constructed in such a way that there is only one way to define a Σ-homomorphism from $\mathbf{T}_\Sigma$ to any other Σ-algebra $\mathbf{A}$. The constants of $\mathbf{T}_\Sigma$ must be mapped to the corresponding constants of $\mathbf{A}$, while the effect of the homomorphism on more complicated terms is similarly determined by the defining equations for Σ-homomorphisms, just as the homomorphisms defined earlier on the natural numbers were determined by the definitions of zero and the successor function in the domain and target algebras. Thus, the term algebra $\mathbf{T}_\Sigma$ is initial in the category $\mathbf{Alg}_\Sigma$ of all Σ-algebras.

We are now in a position to make an explicit connection between many-sorted algebras and data types in programming languages. A many-sorted signature Σ amounts to the specification of an abstract interface for a data type. A *data representation* for that signature would be any Σ-algebra , while a *data type* is an isomorphism class of Σ-algebras. Since a Σ-algebra is initial if and only if it is isomorphic to an initial algebra, the initial algebras in $\mathbf{Alg}_\Sigma$ form an isomorphism class, and, hence, a data type. In fact, this is the type most often used in connection with a given signature. The initial algebras provide *initial algebra semantics* to go along with the syntactic information contained in the signature. Although we want to treat data types abstractly, as a practical matter, some specific data representation must be selected in order to compute with the type. The term algebra provides a convenient representation in that it is initial and can be generated mechanically from the signature (assuming, of course, that the signature itself is in some way computable). A type other than that

This technique of coercing one type of algebra into another (to borrow some programming language terminology) appears repeatedly in the development of algebraic type theory. $\mathbf{T}_{\Sigma(X)}$ also has the property that for any Σ-algebra $\mathbf{A}$ and any mapping f from X to $\mathbf{A}$, there is a unique Σ-homomorphism $f* : \mathbf{T}_{\Sigma(\mathbf{X})} \rightarrow \mathbf{A}$ agreeing with f on X.

We define a Σ-*equation* of sort s to be a triple $< X, t_1, t_2 >$ where X is a finite S-indexed set and t_1 and t_2 are terms of the same sort s in $\mathbf{T}_{\Sigma(X)}$. Notice that X must include all of the variable symbols in t1 and t2, but it may include additional variable symbols. A Σ-algebra $\mathbf{A}$ is said to *satisfy* the Σ-equation $< X, t_1, t_2 >$ if for every mapping $f : X \rightarrow \mathbf{A}$ the homomorphism $f*$ satisfies $f*(t_1) = f*(t_2)$. When X is the empty set, f is the empty mapping and the homomorphism $f*$ is understood to be the unique Σ-homomorphism from $\mathbf{T}_\Sigma$ to $\mathbf{A}$. An algebra satisfies a set E of equations if it satisfies each equation in the set. In that case, the algebra is said to be a (Σ, E)-*algebra*.

There are some technical difficulties associated with using equations to derive a congruence of terms in many-sorted algebras, but the following rules are sound and complete [15]:

Reflexivity.

$$\frac{}{< X, t, t >}$$

Symmetry.

$$\frac{< X, t1, t2 >}{< X, t2, t1 >}$$

Transitivity.

$$\frac{< X, t1, t2 >, < X, t2, t3 >}{< X, t1, t3 >}$$

Substitutivity.

$$\frac{< X, t1, t2 >, < Y, u1, u2 >}{< Z, t1\{u1/x\}, t2\{u2/x\} >}$$

where $< X, t1, t2 >$ is of sort s, the variable x in X is of sort s', the equation $< Y, u1, u2 >$ is of sort s', and $Z = (X - \{x\}) \cup Y$.

Abstraction.

$$\frac{< X, t1, t2 >}{< X \cup \{y\}, t1, t2 >}$$

if y is a variable of sort s that is not in X.

Concretion.

$$\frac{< X, t1, t2 >}{< X - \{x\}, t1, t2 >}$$

where x is a variable of sort s that does not appear in $t1$ or $t2$, and s is *void* in the signature Σ (meaning that $\mathbf{T}_{\Sigma,s}$ is empty).

The first four rules here are derived from those ordinarily used for one-sorted equational logic, while the last two allow for adding and removing variable declarations. The variable declarations are needed to ensure soundness of the logic, while the abstraction and concretion rules are needed for completeness.

Using these rules, one can confirm the existence of an algebra $\mathbf{T}_{\Sigma,E}$ that is initial in the category of all (Σ, E)-algebras. It can be represented as a quotient of $\mathbf{T}_{\Sigma}$ by a congruence generated from the equations E using the rules of inference given above. The same construction can be generalized to create quotients of $\mathbf{T}_{\Sigma(X)}$ for any set X of variables. These quotient algebras, denoted $\mathbf{T}_{\Sigma,E}(X)$, play a role in the semantics of concurrent rewriting presented below.

Let us return to our recurring example of the natural numbers. Suppose that the signature that we have been using is augmented with a binary function symbol $+$ to create a new signature, which we shall also call Σ . Then, as pointed out before, the natural numbers are not initial in the class of all Σ-algebras. But if we add the set of equations E consisting of

$$< \{x\}, +(0, x), x > \text{ and } < \{x, y\}, +(inc(x), y), inc(+(x, y)) >$$

then the natural numbers are initial in the class of all (Σ, E)-algebras.

While many-sorted algebras do a nice job of providing semantics for some types, they are inadequate in contexts where reference to subtypes is needed. Various solutions to this problem have been proposed, but the most successful to date is to generalize the model of many-sorted algebras to so-called order-sorted algebras in which a partial ordering may be imposed on the sorts [16,17]. We will not consider order-sorted algebras here, but it is worth observing that the approach to developing semantics for many-sorted signatures and for many-sorted signatures with equations is mimicked for order-sorted signatures. That is, the signature is augmented syntactically to provide a vehicle for subtyping information, then the signature is interpreted in a class of algebras for which the initial algebra provides a satisfactory semantics. Furthermore, just as the case of many-sorted signatures with equations is reduced to that of unadorned many-sorted signatures, so the order-sorted case can be reduced to those we have already considered. This kind of bootstrapping will also be evident as we turn to the semantics of concurrent rewriting.

6.3 The Semantics of Rewrite Theories

The syntactic construct on which concurrent rewriting is based is a *labelled rewrite theory*

$$\mathcal{R} = (\Sigma, X, E, L, R)$$

consisting of a signature Σ, a countable X set of variables, a set E of equations, a set L of labels, and a set R of labelled rewrite rules, where

$$R \subset L \times T_{\Sigma,E}(X) \times T_{\Sigma,E}(X).$$

An element of R can be written in the form $r : [t] \rightarrow [t']$, the notation $[t]$ indicating the equivalence class of the term $t \in \Sigma(X)$ modulo the equations E. To simplify the exposition, we shall assume that there is just one sort in the signature Σ, so that the arities of function symbols differ only in the number of arguments that they take.

Just as the equations constituted the base case for an inductive definition of equivalence of terms, so the rewrite rules will be the basis for defining *sequents* $\alpha : [w] \rightarrow [w']$, where w and w' are in $\mathbf{T}_{\Sigma,E}(X)$ and α is a *proof term* built as described in the following inference rules:

Identity.

$$\frac{}{[t] : [t] \rightarrow [t]}$$

Σ-structure. For each function symbol f of arity n,

$$\frac{\alpha_1 : [t_1] \rightarrow [t_1'], \ldots, \alpha_n : [t_n] \rightarrow [t_n']}{f(\alpha_1, \ldots, \alpha_n) : [f(t_1, \ldots, t_n)] \rightarrow [f(t_1', \ldots, t_n')]}$$

Replacement. For each rewrite rule $r : [t(x_1, \ldots, x_n)] \rightarrow [t'(x_1, \ldots, x_n)]$,

$$\frac{\alpha_1 : [w_1] \rightarrow [w_1'], \ldots, \alpha_n : [w_n] \rightarrow [w_n']}{r(\alpha_1, \ldots, \alpha_n) : [t(\tilde{w}/\tilde{x})] \rightarrow [t'(\tilde{w}'/\tilde{x})]}$$

Composition.

$$\frac{\alpha : [t_1] \rightarrow [t_2], \beta : [t_2] \rightarrow [t_3]}{\alpha; \beta : [t_1] \rightarrow [t_3]}$$

We can get some intuitive grasp of the proof terms before stating some additional equations that make the intuitions algebraically precise: First, the identity and Σ-structure rules ensure that, for each class $[t]$ in $\mathbf{T}_{\Sigma,E}(X)$ and each term w in $[t]$, there is a sequent $\alpha : [t] \rightarrow [t]$ whose proof term α is "syntactically similar" to w. In fact, w and α would be identical if all square brackets were stripped from α. So the structure of $\mathbf{T}_{\Sigma,E}(X)$ has a counterpart in the collection of proof terms. This is the starting point for ensuring that the semantics of concurrent rewriting subsumes the semantics of many-sorted signatures with equations. Second, in the identity and composition rules, we see a nascent category structure, in which the objects are the equivalence classes and the morphisms are the proof terms. Third, the replacement rule incorporates the rewrite rules into the semantics and represents concurrency in the simultaneous substitution for $\tilde{x}$ and rewriting of t. The concurrency of the system is limited, however, in that two rewrite rules cannot act concurrently. Finally, notice that any sequent generated solely from the identity, Σ-structure, and composition rules will have the same source and target (i.e., it will be of the form $\alpha : [t] \rightarrow [t]$). Thus, only the rewrite rules allow one term to be transformed to an unequal term.

The following equations on proof terms determine a set of equivalence classes of proof terms that is endowed with all of the desired structure:

Categorical structure.

Associativity. For all proof terms α, β, and γ,

$$(\alpha;\beta);\gamma = \alpha;(\beta;\gamma)$$

whenever all of the compositions are defined.

Identities. For all sequents $\alpha : [t] \rightarrow [t']$, we have $\alpha;[t'] = \alpha$ and $[t];\alpha = \alpha$.

Σ-algebraic structure. The following equations hold for each function symbol f:

Composition. $f(\alpha_1;\beta_1,\ldots,\alpha_n;\beta_n) = f(\alpha_1,\ldots,\alpha_n);f(\beta_1,\ldots,\beta_n)$, whenever the compositions are defined.

Identities. $f([t_1],\ldots,[t_n]) = [f(t_1,\ldots,t_n)]$

Satisfaction of E. If $t(x_1,\ldots,x_n) = t'(x_1,\ldots,t_n)$ is in E, then, for all $\alpha_1,\ldots,\alpha_n$,

$$t(\alpha_1,\ldots,\alpha_n) = t'(\alpha_1,\ldots,\alpha_n).$$

Exchange. For each rewrite rule $r : [t(x_1,\ldots,x_n)] \rightarrow [t'(x_1,\ldots,x_n)]$, if $\alpha_i : [w_i] \rightarrow [w_i']$ for $i = 1,\ldots,n$, then

$$r(\alpha_1,\ldots.\alpha_n) = r([w_1],\ldots,[w_n]);t'(\alpha_1,\ldots,\alpha_n)$$

and

$$r(\alpha_1,\ldots.\alpha_n) = t(\alpha_1,\ldots,\alpha_n);r([w_1'],\ldots,[w_n'])$$

```
fmod NATNUMS is
    sort Nat .
    op 0 : -> Nat .
    op inc_ : Nat -> Nat .
    op _+_ : Nat Nat -> Nat [comm assoc id: 0] .
    vars X Y : Nat .
    eq : 0 + X == X .
    eq : (inc X) + Y == inc (X + Y)
endfm

mod NAT-CHOICE is
    extending NATNUMS .
    op _?_ : Nat Nat -> Nat .
    vars N M : Nat .
    rl N ? M => N .
    rl N ? M => M .
endm
```

Figure 6.1: An example of Maude syntax.

Let $\mathcal{T}_{\mathcal{R}}(X)$ denote the set of equivalence classes of proof terms modulo the equations just given. When X is the empty set, write $\mathcal{T}_{\mathcal{R}}(X)$ simply as $\mathcal{T}_{\mathcal{R}}$. The set $\mathcal{T}_{\mathcal{R}}(X)$ can be viewed either as a (Σ, E)-algebra or as the class of morphisms for a category, and the two structures interact appropriately. We can view the exchange rule in different ways, too. If $r(\alpha_1, \ldots, \alpha_n)$ represents a concurrent rewriting, then the exchange rule gives us two ways to look at it as a sequence of operations, with either substitution of subterms or rewriting of the term's structure occurring first. On the other hand, in the terminology of category theory, the exchange rule indicates that r is a *natural transformation.* $\mathcal{T}_{\mathcal{R}}$ is initial in the class of all objects having this combination of structures. So once again, we have an initial object semantics for our syntactic constructs.

6.4 Object-Oriented Programming in Maude

Maude is a language offering a convenient syntax for specifying labelled rewrite theories and implementing the semantics that we have just described. In so doing, it takes advantage of a wealth of research on term rewriting, including especially the Knuth-Bendix algorithm [11,27,28].

Figure 6.1 presents a sample of Maude's syntax, based in part on one of our earlier examples and in part on an example given by Meseguer [29]. The **NATNUMS** module is a functional module containing all of the elements of a many-sorted signature (except that in this case there is just one sort, namely **Nat**). The lines beginning with the keyword op define function symbols and indicate their arities. In addition, the function symbol + is declared to represent a commutative and associative operation

```
sorts Object Attribute Attributes Msg Configuration
  Value OId CId AId .
subsorts OId CId AId < Value .
subsorts Attribute < Attributes .
subsorts Object Msg < Configuration .
op <_:_|_> : OId CId Attributes -> Object .
op (_:_) : AId Value -> Attribute .
op _,_ : Attributes Attributes -> Attributes
  [comm assoc id: nil] .
op _ _ : Configuration Configuration -> Configuration
  [comm assoc id: ]
```

Figure 6.2: Specification of an object-oriented environment

having an identity of 0. Such properties are indicated in this way rather than by equations because the properties play a special role in the standard term rewriting algorithms. Two variables of a specified sort are declared and used to state equations.

The second module, NAT-CHOICE, is a "system module" in that it forces us to use the semantics of rewrite systems rather than many-sorted signatures to interpret the program. The module is declared to be extending NATNUMS, meaning that the semantics of NATNUMS are effectively embedded in that of NAT-CHOICE and the syntax of NATNUMS can be used in declaring NAT-CHOICE. In this case, only the sort Nat is reused. The NAT-CHOICE module gives two rewrite rules, indicated by the keyword rl. One could regard them as representing a non-deterministic choice between two natural numbers. Interpreted as equations, they would imply a semantics in which all elements of sort Nat are equal. This is certainly not intended, nor is it consistent with the notion of "extending" NATNUMS. The semantics of the preceding section solves this problem if we take these two Maude modules together to specify a labelled rewrite theory (unique labels for the rules being supplied automatically by the Maude interpreter).

An approach to object-oriented programming through concurrent rewriting is suggested by the semantics of Actors, where object-based computation consisted of transforming configurations of messages and objects into new configurations. Meseguer's representation of the top level of an object-oriented environment is given in Figure 6.2. The intended interpretation is that an object is a term of the form

$$< O : C | a_1 : v_1, \ldots, a_n : v_n >$$

where O and C identify the object and its class, respectively, and each $a_i : v_i$ gives the name and value of an attribute or instance variable of the object. A configuration of an object-oriented system consists of a collection of objects and messages (whose form is not specified here and, indeed, would most likely vary considerably).

The syntax of order-sorted signatures is used to clarify the relationships among the sorts and simplify the description of attributes and configurations. The sort `Value` might subsume all of the data types defined in the environment. Rules associated with specific modules would show how a configuration that included messages and objects having particular forms can be rewritten into a new configuration, much as was done with Actors. To ease the task of object-oriented programming, Maude includes special syntax to specify messages, classes, and attributes, and to hide and thereby encapsulate aspects of a module's definition, but the syntax of these *object-oriented modules* can be translated readily into that of functional and system modules.

Notice how the semantics of concurrent rewriting reduces the programmer's workload by allowing rewrite rules for different types of objects to be written without regard for the rules obeyed by other types of objects. Although a configuration consists of a particular sequence of messages and objects, the programmer does not need to worry about where certain messages or objects occur in the sequence. In fact, the nature of a configuration need not be known to the programmer. It is enough that the programmer's rules for processing messages interact with the concatenation operator for configurations as described in section 6.3. Then, since concatenation is commutative and rules act on equivalence classes of terms, the order of messages and objects is irrelevant.

The semantics also allow rules to operate independently because no interaction among them is specified. This does, however, leave the task of avoiding deadlock or starvation to the programmer since nothing in the semantics ensures that the "right" rule will be chosen in situations when more than one rule could be applied.

Chapter 7

Conclusion

We have just examined a succession of increasingly powerful models for object-based systems–CCS, Actors, Abacus, and Maude. From the point of view of object-based systems, CCS is really a proto-model. It is the basis of other models, but many of the features that one wants when modelling encapsulated objects. Actors provides that encapsulation, and introduces the possibility of creating new objects dynamically. Abacus improves on Actors' encapsulation by allowing it to extend to constellations of inter-related agents. Maude provides a fully-developed system of type declaration that interacts harmoniously with facilities for concurrent operations.

While these models grow increasingly powerful, they also present ever more complicated problems of implementation in a parallel processing environment. Actors might be regarded as the assembly language of object-based systems in this regard. The mapping of objects to processors in Actors is relatively straightforward, just as an assembly language facilitates the mapping of variables to registers.

But as facilities for sharing grow more sophisticated, the task of managing resources shared among objects operating on different processors becomes more complex. This should not be too great a surprise. By allowing objects to share resources or code, we reduce their independence. Increased interdependence, in turn, places a greater premium on matching objects to processors in an efficient manner, both so that objects that communicate frequently can do so easily and so that objects with special computational needs are assigned to processors that meet those needs.

The contrast between Actors and Maude illuminates the challenges presented by increased sharing. The semantics of Actors is expressed in terms of messages and behaviors, each with an associated mail address. Once that mail address is assigned to a processor, it is easy to see how to manage the flow of computation because the actors are so self-contained. In Maude, on the other hand, the semantics is expressed using terms constructed from syntactic elements introduced by a variety of different objects. While concurrent rewriting provides for separate simplification of different parts of a term, the "ownership" of those parts is not always clear because of inheritance and because the focus of attention shifts rapidly to different subterms as simplification proceeds. This may be an argument for basing a model of object-oriented programming on a model for concurrency, since the latter is more closely related to the underlying hardware on which execution of the program will occur.

We are just beginning to develop rigorous models of object-based systems. Each of the models that we have considered represents the first few steps in an effort to develop a comprehensive model of object-based systems that addresses the many desirable aspects of object-oriented programming, allowing them to co-exist without clashing. The authors of these models acknowledge that the models are incomplete. Certainly no model has achieved the sort of influence that CCS has for concurrent computation or the Turing machine has for computability.

It may be that we are asking too much of object-based programming languages. They are to be suitable for rapid prototyping of systems and long-term software development efforts. They are to have a tractable mathematical semantics but be capable of modelling the complexities of the real world. They are to give the programmer great flexibility, but carefully guard the encapsulation of objects and other modules.

Nonetheless, researchers are optimistic that object-oriented programming will eventually meet the high expectations that many have for it. A carefully formulated model will be an important part of achieving the goals of object-oriented programming.

Bibliography

[1] Gul Agha. Semantic considerations in the actor paradigm of concurrent computation. In *Seminar on Concurrency*, Lecture Notes in Computer Science, pages 151–179, 1985.

[2] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* The MIT Press, Cambridge, Massachusetts, 1986.

[3] Gul Agha and Carl Hewitt. Actors: A conceptual foundation for concurrent object-oriented programming. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 49–74. The MIT Press, Cambridge, Massachusetts, 1987.

[4] K. R. Apt. Ten years of Hoare's logic, part I. *ACM Transactions on Programming Languages and Systems*, 3:431–484, 1981.

[5] G. Attardi and M. Simi. Semantics of inheritance and attributions in the description system Omega. In *Proceedings of IJCAI 81*, 1981.

[6] Stephen Bear, Phillip Allen, Derek Coleman, and Fiona Hayes. Graphical specification of object oriented systems. In *OOPSLA ECOOP '90 Conference Proceedings*, pages 28–37. The Association for Computing Machinery, October 1990.

[7] W. D. Clinger. Foundations of actor semantics. Technical Report AI-TR-633, MIT Artificial Intelligence Laboratory, May 1981.

[8] O. J. Dahl, B. Myrhaag, and K. Nygaard. *Simula 67 Common Base Language.* Norwegian Computing Center, 1970.

[9] Scott Danforth and Chris Tomlinson. Type theories and object-oriented programming. *Computing Surveys*, 20(1):29–72, March 1988.

[10] Pierpaolo Degano and Ugo Montanari. Concurrent histories: A basis for observing distributed systems. *Journal of Computer and System Sciences*, 34:422–461, 1987.

[11] N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17:279–301, 1982.

[12] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics.* Springer-Verlag, Berlin, 1985.

[13] K. Futatsugi et al. Principles of OBJ2. In *Proceedings, Symposium on Principles of Programming Languages*, pages 52–66. Association for Computing Machinery, 1985.

[14] Robert W. Floyd. Assigning meanings to programs. *Proceedings of Symposia in Applied Mathematics*, 19:19–31, 1967.

[15] J. A. Goguen. Completeness of many-sorted equational logic. *Houston Journal of Mathematics*, 11:307–334, 1985.

[16] J. A. Goguen. Remarks on remarks on many-sorted equational logic. *SIGPLAN Notices*, 22:41–48, April 1987.

[17] J. A. Goguen, J.-P. Jouannaud, and J. Meseguer. Operational semantics for order-sorted algebra. In *Proceedings of the 12th ICALP*, Lecture Notes in Computer Science, pages 221–231, Berlin, 1985. Springer-Verlag.

[18] Joseph Goguen, Claude Kirchner, and José Meseguer. Concurrent term rewriting as a model of computation. In R. Keller and J. Fasel, editors, *Proceedings of a Workshop on Graph Reduction*, Lecture Notes in Computer Science, pages 53–93, 1987.

[19] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[20] Ursula Goltz and Alan Mycroft. On the relationship of CCS and Petri nets. In *11th ICALP*, Lecture Notes in Computer Science, pages 196–208, 1984.

[21] D. Harel, A. Pnueli, J. P. Schmidt, and R. Sherman. On the formal semantics of statecharts. In *Proceedings of the Second IEEE Symposium on Logic in Computer Science*, pages 54–64. IEEE, 1987.

[22] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[23] David Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

[24] C. Hewitt and H. Baker. Laws for communicating parallel processes. In *1977 IFIP Congress Proceedings*, pages 987–992. IFIP, August 1977.

[25] C. E. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, June 1977.

[26] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, New York, 1985.

[27] G. Huet and D. Oppen. Equations and rewrite rules: A survey. In R. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 349–405. Academic Press, 1980.

[28] D. Knuth and P. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

[29] José Meseguer. A logical theory of concurrent objects. In *OOPSLA ECOOP '90 Conference Proceedings*, pages 101–115. The Association for Computing Machinery, October 1990.

[30] José Meseguer. Rewriting as a unified model of concurrency. In *Proceedings of the Concur '90 Conference*, Lecture Notes in Computer Science, pages 561–576, 1990.

[31] Josephine Micallef. Encapsulation, reusability and extensibility in object-oriented programming languages. *Journal of Object-Oriented Programming*, 1(1):12–38, April/May 1988.

[32] Robin Milner. *A Calculus of Communicating Systems.* Lecture Notes in Computer Science. Springer-Verlag, New York, 1980.

[33] Robin Milner. *Communication and Concurrency.* Prentice Hall, New York, 1989.

[34] Peter D. Mosses. Unified algebras and modules. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 329–343. The Association for Computing Machinery, January 1989.

[35] O. M. Nierstrasz. Active objects in Hybrid. In *OOPSLA '87 Conference Proceedings*, pages 243–253. The Association for Computing Machinery, October 1987.

[36] Oscar Nierstrasz. The next 700 concurrent object-oriented languages. Unpublished draft of February 5, 1991.

[37] Oscar Nierstrasz. A pattern-based notation for modelling active objects. Unpublished draft of October 3, 1990.

[38] Oscar Nierstrasz. A survey of object-oriented concepts. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 3–21. ACM Press, New York, 1989.

[39] Oscar Nierstrasz. A guide to specifying concurrent behavior with Abacus. In D. C. Tsichritzis, editor, *Object Management*, pages 267–293. Centre Universitaire d'Informatique, University of Geneva, July 1990.

[40] Oscar Nierstrasz and Michael Papathomas. Towards a type theory for active objects. In D. C. Tsichritzis, editor, *Object Management*, pages 295–304. Centre Universitaire d'Informatique, University of Geneva, July 1990.

[41] Oscar Nierstrasz and Michael Papathomas. Viewing objects as patterns of communicating agents. In *OOPSLA ECOOP '90 Conference Proceedings*, pages 38–43. The Association for Computing Machinery, October 1990.

[42] Ernst-Rüdiger Olderog. Operational Petri net semantics for CCSP. In Grzegorz Rozenberg, editor, *Advances in Petri Nets*, Lecture Notes in Computer Science, pages 196–223. Springer-Verlag, New York, 1987.

[43] Michael Papathomas. Working group on models for concurrent object-based programming. In *Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 9–11. The Association for Computing Machinery, September 1988.

[44] G. D. Plotkin. An operational semantics for CSP. In D. Bjørner, editor, *Formal Description of Programming Concepts II*, pages 199–223. North Holland, Amsterdam, 1983.

[45] A. Pnueli. Applications of temporal logic to the specification and verification of reactive systems: A survey of current trends. In *Current Trends in Concurrency*, Lecture Notes in Computer Science, pages 510–584. Springer-Verlag, Berlin, 1986.

[46] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *OOPSLA '86 Conference Proceedings*, pages 38–45. The Association for Computing Machinery, September 1986.

[47] Peter Wegner. Concepts and paradigms of object-oriented programming. *OOPS Messenger*, 1(1):7–87, August 1990.

[48] Glynn Winskel. Event structure semantics for CCS and related languages. In *9th ICALP*, Lecture Notes in Computer Science, pages 561–576, 1982.