ORDER AND CONTAINMENT IN CONCURRENT SYSTEM DESIGN

by

John Sidney Davis II

Memorandum No. UCB/ERL M00/47

8 September 2000

ORDER AND CONTAINMENT IN CONCURRENT SYSTEM DESIGN

by

John Sidney Davis II

Memorandum No. UCB/ERL M00/47

8 September 2000

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Order and Containment in Concurrent System Design

by

John Sidney Davis II

Abstract

Order and Containment in Concurrent System Design

by

John Sidney Davis II

Doctor of Philosophy in Engineering-Electrical Engineering and Computer Science

University of California at Berkeley

Professor Edward A. Lee, Chair

This dissertation considers the difficulty of modeling and designing complex, concurrent systems. The term *concurrent* is used here to mean a system consisting of a network of communicating components. The term *complex* is used here to mean a system consisting of components with different models of computation such that the communication between different components has different semantics according to the respective interacting models of computation.

Modeling and designing a concurrent system requires a clear understanding of the types of relationships that exist between the components found within a concurrent system. Two particularly important types of relationships found in concurrent systems are the *order* relation and the *containment* relation. The order relation represents the relative timing of component actions within a concurrent system. The containment relation facilitates human understanding of a system by abstracting a system's components into layers of visibility.

The consequence of improper management of the order and containment relationships in a complex, concurrent system is deadlock. *Deadlock* is an undesirable halting of a system's execution and is the most challenging type of concurrent system error to debug. The contents of this dissertation show that no methodology is currently available that can concisely, accurately and graphically

model both the order and containment relations found in complex, concurrent systems. The result of the absence of a method suitable for modeling both order and containment is that the prevention of deadlock is very difficult. This dissertation offers a solution to this problem with the introduction of the *diposet*.

Professor Edward A. Lee
Dissertation Committee Chair

This dissertation is dedicated to

O.C. Jones.[1]

O.C. Jones is the janitor who told me to "keep working hard, son." O.C. Jones is the faculty member, with few black colleagues, who encouraged me to push forward. O.C. Jones is the church deacon, the secretary, the grocery clerk, the barber who continually whispered encouragement and shouted for educational achievement. Without O.C. Jones I would not have completed this project.

[1] O.C. Jones is a fictional character who represents senior citizens of African descent in a fashion similar to Langston Hughes' Jesse B. Simple. I created O.C. Jones with the help of Marshell D. Jones.

Contents

List of Figures

List of Tables

Acknowledgements

I thank God for my completion of this dissertation and for all of the people who were placed in my life to help me as I made this journey. First and foremost, I thank my immediate family. I thank my father, **John S. Davis**, for teaching me the value of hard work and consistency and for the love of music that I inherited from him. I thank my mother, **Julia A. Davis**, for her curiosity and laughter and for teaching me how to have bold visions and then make them come true. I thank my brother, **Jeffrey C. Davis**, for teaching me compassion and sincerity and for reminding me to not forget to hit the ball.[2] I thank my sister, **Jennifer N.G. Davis**, for teaching me that if I'm ever losing an argument a sufficient level of silliness will distract my opponent and make me the victor. I also thank Jennifer for reminding me of the importance of teaching children.

I thank my good friend **Adrian J. Isles, Ph.D.** for being an excellent travel mate on this arduous doctoral journey. Our late night debates and housemate pontifications on random philosophical topics were fun and stimulating. I owe a debt of gratitude to my advisor **Edward A. Lee, Ph.D.** Edward's emphasis on academic rigor is at once intimidating and comforting. I must also mention that Edward's EECS 225A Digital Signal Processing course was one of the best organized classes I have ever taken. Beyond this I thank Edward for welcoming me into his research group after my first advisor moved to Europe. I truly appreciated the members of both my dissertation and qualifying exam committees. **Thomas A. Henzinger, Ph.D.** provided exhaustive rigor and **Shankar Sastry, Ph.D.** has a contagious academic enthusiasm that was exciting. I truly enjoyed working with **James A. Sethian, Ph.D.** both on my committees and with the Java™ level set algorithms.

There are several members of my research group that I wish to thank. **Geronico "Ron" Galicia** encouraged me to consider the electronic design automation research specialty and we developed a good friendship as we began our careers in the Ptolemy group. **Michael "Cameron" Williamson, Ph.D.** provided calm wisdom when I was a new member in my research group. I learned C++ under the patient tutelage of **Allan Lao**. **Johnathan Reason** was an excellent study mate and helped me stay focused. **Dr.ir. Bart Kienhuis**, a postdoctoral student, gave me a tremendous amount of help as I worked on the details of my dissertation. He helped me work out bugs with the LaTex word processing commands and he gave very detailed feedback on style. Furthermore, he repeatedly

[2] Jeffrey was my biggest fan during my little league baseball years; he would often shout "Don't forget to hit the ball" when I was up to bat.

reinvorced my confidence in the subject matter by recognizing the significance of my contributions. Of all of my Dutch friends named "Bart Kienhuis," Bart is my favorite. **Stephen Edwards, Ph.D.** provided "tough academic love" as I prepared for my qualifying examination and he reminded me and many others to put things in perspective and graduate.[3] **Mudit Goel** and I shared adjacent desks in the Ptolemy Group that facilitated stimulating conversation and a great friendship. **Bilung Lee, Ph.D.** helped me clear my dissertation thoughts as we both neared the finish line together and **Nick Zamora** provided useful comments on a draft of my first chapter.

My extended family and friends provided immeasurable support. **Marshell D. Jones** reminded me to take graduate school and life by the reins and ride! **Peggye Brown** kept me well fed; oh how wonderful it was to receive her abundant gifts of home cooked food after a late night of software debugging. **Sheila Humphreys, Ph.D.** had the ability to boost my confidence and get me smiling on even the rainiest of days. Peggye and Sheila were my mothers away from home. My cousin **Carla Chambers** and her son **Chréstien** were distant relatives upon my arrival to California, but they immediately took me in and reminded me of the strength of a loving family bond. I thank my aunt **Sarah "Tina" McIntosh, M.D.** who left her husband and two sons at home and braved a cross country trip from Ohio with her two year old daughter to attend my graduation. I was blessed to have a wonderful church family at **Allen Temple Baptist Church** and in particular the **Young Adult Sunday School Class** taught by **Carl Gill** helped keep me spiritually centered throughout this process.

The members of **BGESS (Black Graduate Engineering and Science Students)**, **BESSA (Black Engineering and Science Students Assocation)** and **GSAD (Graduate Students of African Descent)** were a source of many lifetime friendships. These organizations were critical to my success at UC Berkeley. **MTFO**, a small dissertation support group of friends, provided a wonderful, weekly push. **MTFO** included **Andre Lundkvist, David Campt, Ph.D., Adrian J. Isles, Ph.D., Jeffrey R.N. Forbes, Ricks Brooks, Ph.D., Kamau Bobb, John Harkless, Ray Gilstrap, Tajh Taylor** and **Tameka Carter**. The war stories, accolades and laughter shared in MTFO were mental nourishment and I highly recommend a dissertation support group for others working on graduate degrees.

It is a pleasure to thank everyone who (at one time or another) I shared living space with in

[3] Planning to begin graduate school? Heed Stephen's advice and read the book "Getting What You Came For" by Robert L. Peters.

the penthouse at **474 Merritt Avenue, Apt. 8** in Oakland, California. **Adrian J. Isles, Ph.D., Jeffrey R.N. Forbes, Robert Stanard, Damian I. Rouson, Ph.D., Kysseline Jean-Marie, Carrletta M. Price** and **Christine Hoang** were all great roommates. I can't wait until the movie comes out!

I conclude these acknowledgements by thanking the host of government and corporate sponsors that supported my graduate work through the Ptolemy project. These supporting organizations included the **Defense Advanced Research Projects Agency (DARPA)**, the **State of California MICRO Program**, and the following companies: **The Alta Group of Cadence Design Systems, Hewlett Packard, Hitachi, Hughes Space and Communications, NEC, Philips**, and **Rockwell.**

Chapter 1

Managing Inconsistency And Complexity In Concurrent Systems

The first rule of using threads is this: avoid them if you can.
- The online Java™Tutorial[1]

A concurrent system is a set of interacting components. A concurrent, computational system consists of components that cooperate in computing data [Andrews, 1991; Milner, 1989; Hoare, 1985]. We find concurrent, computational systems all around us. For example, the operating systems found on personal computers concurrently run browsers, word processors and database programs. The embedded systems found in automobiles concurrently respond to the driver's foot and the road being travelled upon to properly operate an anti-lock braking system. State-of-the art cell phones transfer the user's voice into bits and bytes while simultaneously responding to call waiting requests and input from the phone touch pad.

In some cases, concurrency is *apparent*, meaning that only one action happens at any given time although a human would perceive different actions happening simultaneously. In other cases, concurrency is *actual*, meaning that two or more actions occur simultaneously. In this latter category, concurrent computation is supported by multiple processing units while in the former case a single processing unit alternates between the actions of each component. Hence, an enterprise compute

[1]The online Java™Tutorial can be found at `http://java.sun.com/docs/books/tutorial/index.html`. This quote can be found on the "Creating a GUI with JFC/Swing" trail in the "Using Other Swing Features" lesson entitled "How to Use Threads."

server with multiple CPUs might perform actual concurrency while a desktop computer with a single CPU performs apparent concurrency. Perhaps the Internet is the largest example of a computational system that performs actual concurrency.

Concurrent, computational systems are very complex, and this has placed a great burden on those who design such systems. I believe that much of the complexity associated with concurrent systems is based on the sequential style of thinking engaged in by humans. Human sequential thought patterns are poorly matched for describing, comprehending and building concurrent systems, especially given the magnitude of many of the concurrent computational systems being built.[2]

Perhaps the more practical difficulties with concurrent programming are related to the problem of guaranteeing that disparate components in a concurrent system have knowledge that is consistent with the knowledge of other components. Consider a real world "soccer mom" example: mom, her teenagers, the pets and other related persons are all components in a concurrent system. Inconsistent knowledge in such a system can lead to unwanted behavior: the soccer mom dropping the dog off at the veterinarian on the wrong day because of a missed answering machine message from the vet. Inconsistent knowledge can also lead to unnecessary wait periods: the teenager waiting for mom to pick him up after soccer practice and not realizing that mom expects him to bike home.

Easing the burden of the designers by managing the complexity of concurrent system design has been, in part, the goal of the *electronic design automation* (EDA) community. One high level technique for managing design complexity is through *component-based design.* Component-based design leverages the natural partitioning of a concurrent system into components. A component-based design approach presumes a mechanism for information transfer between components and a mechanism for computation of the transferred information. The mechanisms for information communication and computation vary across different types of systems that might be described by a component-based design approach. For example, the communication style of components in a cellular phone may be quite different from the communication style in a medical device control unit. Informally, the characterization of information transfer and computation are jointly referred to as a *model of computation* (MoC).

[2] I am confident that there are many who disagree with my assertion that humans think sequentially. If anyone in society thinks concurrently, surely it is the soccer mom dealing with a long shopping list, rowdy teenagers, dinner time and dirty pets; I contend that soccer moms are at best engaged in apparent concurrent thinking. Nevertheless, I will not vigorously argue this point and instead will let the copious literature on the difficulties of concurrent programming serve as my evidence.

In a typical computational system, the MoC associated with components of one part of the system can be quite distinct from that of components in a different part of the system. For this reason, a heterogenous set of MoCs is often necessary for specifying a complete system. While a model of computation specifies how components of a particular MoC interact, it says little about how interaction occurs across the boundary of two MoC's. In this dissertation, I present a technique for dealing with communication between components of different MoC's. My approach applies to a specific class of models of computation that are well suited for systems in which components have autonomous control. In the remainder of this chapter, I will consider component-based design and in so doing I will establish the background for heterogeneous MoCs and several other problems for which my dissertation work provides a solution.

1.1 Component-Based Design

Component-based design is applied in many disparate fields including object oriented programming, software engineering, formal semantics, and system level EDA. The various communities that use a component-based design approach tackle different problems with unique solutions. Many of these solutions can be leveraged by multiple communities. Considering multiple communities and their varied techniques can provide the breadth upon which new solutions will arise.

1.1.1 Object-Oriented Programming

Object-oriented programming places components at the core by equating components with software objects. An *object* is a set of variables with a set of methods that may operate on those variables and/or parameter data. The application of object-oriented techniques to a software system results in a system of interacting objects. Each object maintains *state* based on the values of the variables it contains. The *behavior* of an object represents how its functions can be invoked and whether such invocations impact the object's state. Note that variables may be objects themselves.

The decomposition of a system into objects is fundamentally about managing complexity by dividing and conquering. Grady Booch, a pioneer in object-oriented design, cites two basic approaches for dividing and conquering through decomposition: algorithmic decomposition and object-oriented decomposition. *Algorithmic decomposition* breaks a system into modules where each

module is a step in a logically sequential process. In *object-oriented decomposition*, objects each have independent behavior and state and hence need not operate in any logically sequential manner. Booch argues that while object-oriented models may not be superior to algorithmic decomposition models for all systems, they are superior more often then not [Booch, 1994].

Booch defines object-oriented techniques as having four major elements and three minor elements. I will present five elements that are most relevant to this work.

1. Abstraction

   An abstraction focuses on a set of essential characteristics of an object relative to a given perspective. Abstraction results in a particular interface for an object where the interface is amenable to a relevant perspective. The notion of abstraction is used in everyday life whenever people agree to focus on certain similarities and ignore certain differences for comparing a set of entities. Hence, we apply abstraction when, for example, we define the notion of *household pets*. There are clearly many differences between dogs and cats, but from the perspective of domestication and animal companionship, dogs and cats both can be defined as household pets.

   Abstraction plays a central role in the object-oriented design process. Abstraction impacts the particular details that need to be implemented for a given system being designed. Consider how abstraction might impact the design of a database for storing music. One perspective might place emphasis on the song artists. Another perspective might emphasize the genre of the songs in the database. Still another perspective might focus on the title of the songs. Different perspectives impact the database's interface design.

2. Encapsulation

   While abstraction determines an interface, encapsulation determines the implementation of an interface. Through encapsulation, an interface is separated from its implementation. One technique for accomplishing encapsulation is information hiding. In essence, the details of an interface's implementation are hidden from view. Semantically, encapsulation results in a has-a relationship. A typical object has-a variable.

   A real world example of encapsulation is realized whenever there is a spokesperson for a corporation or political body. When the press questions a large corporation about recent profits or

about its role in legal proceedings, the question is typically answered by a single spokesperson. Even though the single spokesperson gives a statement, we know that this statement is the result of numerous meetings, phone calls and board room debates. Nevertheless, these details are hidden from the view of the press to simplify the process.

3. Modularity

Modularity is where the notion of objects enter into object-oriented design. Modularity is where we decompose a system. In so doing, we partition an abstraction into discrete units: objects. An object serves as a boundary within which a single abstraction lives. Given the use of objects, it is often advantageous to maximize reuse. The *reuse* of an object is possible if it can be used by different applications. Efficient reuse can play a powerful role in object-oriented design because it allows the time spent on implementating an object to be amortized over several applications.

Closely related to the notion of an object is the notion of a *class*. An object is an implementation while its class is the blueprint. Presumably, the class for a human being is realized in that person's DNA. The class for a bicycle is realized in the design specs stating the characteristics of its tires, gears, pedals and handlebars. For a single class, several objects can be realized and each object has its own identity. Two bikes might be designed from the same blueprints but they are distinct bikes that can exist on different corners of the globe.

4. Hierarchy

Hierarchy prioritizes a set of abstractions. Such a prioritization is necessary because for most systems, a large number of abstractions are possible and it is necessary to organize the abstractions. Let's return to our household pets, the cat and dog. Aside from household pet, it is possible to abstract cats and dogs according to their mammalian characteristics, the number of limbs they have, and the color of their fur. Hierarchy orders these different abstractions. Each abstraction may result in a different set of classes. The household pet classes are cats, dogs and perhaps goldfish and turtles. The mammalian classes include cats, dogs, cows, whales and human beings among other mammals.

Semantically, hierarchy results in an *is-a* relationship. An is-a relationship is typically realized through inheritance. *Inheritance* prioritizes abstractions by hierarchically layering them

on top of one another. Basic or more fundamental abstractions exist towards the top of the hierarchy while more refined and detailed abstractions exist toward the bottom of the hierarchy. Note that in general, there is not necessarily one top or one bottom of the hierarchy. If a class inherits characteristics from another class due to their relative positions in a class hierarchy, then we say that the former is a *subclass* and the latter is a *superclass*. A possible hierarchy might be $mammal \rightarrow household\ pet \rightarrow dog \rightarrow light\ brown\ dog$. Thus, a household pet is-a mammal; likewise a dog is-a household pet; and so forth.

Hierarchy can also be viewed as a *has-a* relationship. A has-a relationship focuses on containment and in object-oriented programming considers how objects contain one another. If object A has object B as one of its variables than A contains B. The has-a hierarchy found in government systems serves as a good example: nations *have* states *have* counties *have* cities *have* neighborhoods. In this example hierarchy, nation is the superclass with its immediate subclass being state.

5. Type

   Type is closely related to the notion of class. Typing places constraints on how abstractions can be combined and allows the designer to enforce design decisions. Unlike the four *major* elements presented above, Booch describes type as a *minor* element of object-oriented design. He considers it important but non-essential. There are examples of object-oriented languages that are not typed (e.g., Smalltalk).

   There are two key concepts in typing. The first concept is related to rigor: how rigorously is typing enforced. A *strongly typed* language detects at compile time whether typing constraints are violated. A *weakly typed* or *untyped* language loosens (to a small or large degree, respectively) this detection. The second concept determines how the names of variables are bound to (or associated with) types. *Static binding* means that variables are bound at compile time. *Dynamic binding* (or late binding) means that the types of some variables are not known until a program is actually run. The interaction of typing and inheritance is polymorphism. In *polymorphism*, a single variable name may represent objects of many different classes that all have a common superclass.

Object-oriented techniques are generally applied within the context of software. Nevertheless, the

term "object" was originally used within a hardware context and was first associated with descriptor-based architectures and later capability-based architectures in the early 1970's. These architectures served to close the gap between high level languages and the low level hardware that was being controlled. Many fundamental ideas of object oriented programming first appeared in Simula 67 [see Booch, 1994, pg. 37]. Smalltalk evolved the concepts in Simula by requiring all objects to instantiate a class. Dijkstra was the first researcher to formally speak of composing systems as layers of abstractions [Dijkstra, 1968a]. Parnas introduced the idea of information hiding [Yourdon, 1979]. Hoare contributed with his theory of types [Nygaard and Dahl, 1981, pg. 460]. Object-oriented programming is experiencing a high point of sorts as we exit the 20^{th} century through the Java programming language which enforces object-oriented programming in a manner that has not widely been seen prior.

1.1.2 Software Engineering

Software engineering is the organized production of software using a collection of predefined techniques and notational conventions [Rumbaugh *et al.*, 1991]. Although software engineering as a community is very diverse, a great deal of effort has been expended on extending and refining object-oriented techniques. In particular, there has been emphasis on specifying object models and in formalizing their reuse.

The *Unified Modeling Language* (UML) is an attempt at facilitating the specification of an object model [Booch, Rumbaugh, and Jacobson, 1999]. The UML is a graphical language for specifying an object-oriented model. The building blocks of UML are *things*, *relationships* between things and *diagrams*. Things are the objects that make up a model. Examples of relationships between things are the is-a and has-a relationships between objects. Diagrams serve as graphical tools for representing a set of things. Different diagrams are employed depending on the types of things being represented. Example diagrams include class diagrams, object diagrams and statechart diagrams.

Design patterns are object-oriented solution templates; they are methodologies for reusing tried and true design approaches. In a sense, design patterns extend the fruits of UML by canonizing them. Design patterns have evolved from years of object-oriented design. It became apparent over time that certain common designs were being applied over and over to different problems that shared

essential qualities. Design patterns encourage design reuse. Objects that are good implementations of a particular design can potentially be reused as well. A design pattern has four essential elements [Gamma *et al.*, 1995].

1. Pattern Name

   The pattern name allows us to refer to a particular design and serves as a member of a vocabulary of patterns. Ideally, the name should be succinct but meaningful.

2. Problem

   The problem provides context and determines when a pattern should be used.

3. Solution

   The solution describes the elements of the particular pattern, their relationships, and collaborations.

4. Consequences

   When choosing any design pattern there are always trade-offs. The designer is made explicitly aware of the trade-offs by a listing of pattern consequences.

Software engineering as a field subsumes the field of object-oriented techniques. Boehm wrote a classic survey paper that brings to light the problematic trends of software design [Boehm, 1976]. The UML evolved from Booch and Jacobson's Object-Oriented Software Engineering (OOSE) approach and Rumbaugh's Object Modeling Technique (OMT) in the mid-1990's. The notion of software design patterns was borrowed from the field of architecture. Christopher Alexander recognized the existence of design patterns in building houses and towns [Alexander *et al.*, 1977]. The "gang of four" (Gamma, Helm, Johnson and Vlissides) adapted patterns to software by codifying 23 commonly used patterns [Gamma *et al.*, 1995]. There have been several extensions of their initial set.

A picture is worth a thousand words and one of the key benefits of the UML community as well as the gang of four is their emphasis on graphical representations. Unfortunately, none of the 23 patterns offered by the gang of four are explicitly intended for concurrent systems. Although it is true that each pattern offers structure that can be used to specify relationships between components within a concurrent system they include no description of how execution should occur. Concurrent systems are often dealt with independent of the structure of a system and often books on concurrent

systems emphasize logic [Andrews, 1991; Schneider, 1997]. Graphical representations of concurrent systems can help to clarify meaning and serve to convey ideas between designers. There are very few well accepted graphical techniques in the concurrent programming community and certainly there are no canonical techniques. In Chapter 2, I offer a graphical technique that is easy to understand and suitable for richly representing concurrent programs.

1.1.3 Formal Semantics

In the formal semantics (or formal methods) community, what would be considered a component is referred to as a process, agent or actor. The formal semantics community typically reverses the efforts of the other communities spoken of thus far. Rather than emphasizing the decomposition of a system, formal semanticists study the systems that result from compositions of processes. For this reason, a formal semantics system is often referred to as a *process algebra* - the elements of a process algebra are processes and the algebra consists of operations for composing processes. The desired goal of a process algebra is to show that properties about a composition are guaranteed given that the processes being composed satisfy certain criteria. The attainment of this goal means that process algebra can be used to mathematically verify characteristics about a system being designed. Unfortunately, the degree to which process algebra are successful at attaining this goal is limited.

Formal semantics comes in many different flavors with each flavor represented by a particular modeling system. Examples include Tony Hoare's *Communicating Sequential Processes* (CSP), Robin Milner's *Calculus of Communicating Systems* (CCS) and Gul Agha's *Actor's Model.* A concept that is shared by most systems of formal methods is the separation of communication from concurrent computation. This separation is made clear by the words of Robin Milner:

> "Each action of an agent is either an interaction with its neighbouring agents, and then it is a communication, or it occurs independently of them and then it may occur concurrently with their actions [Milner, 1989]."

Communication is an action that is shared between a set of processes. Computation is an internal action that is independent of other processes. The separation of communication from computation is a very powerful abstraction and can be thought of as extending an object by partitioning its functions into an external/internal dichotomy. Formal semantics give meaning to objects.

At the highest level there are two mechanisms by which processes or components communicate. *Shared variable* communication involves a globally accessible repository of data that processes write to and read from. A process makes a portion of its state available to other processes through this shared variable. The critical concern in this mechanism is in how to make sure that the state of the shared variable is consistent with the intentions of the processes. A real world example of this is realized in a joint bank account shared by a husband and wife. If the husband and wife simultaneously attempt to retrieve money from the single account from separate branch locations, the bank must make sure that it does not give out more money then is available. The fundamental solution to this problem is based on mutual exclusion. Only one patron can gain retrieval access from a bank account at a time.

Message passing is the second fundamental style of communication. In message passing, components communicate through channels. A component must have a channel for each other component it wants to communicate with. Communication through a channel falls into two categories: asynchronous and synchronous. *Synchronous message passing* requires both the sender and receiver connected by a channel to be synchronized when a communication occurs. In synchronous message passing, the notion of communication is atomic. Both the sender and receiver must be simultaneously engaged during the duration of the communication. An example of this style of communication occurs with the passing of the baton during a relay race. *Asynchronous message passing* does not require sender and receiver to be simultaneously engaged. As long as room is available in the channel, a sender may place a message in the channel and then continue with other activities independent of whether the receiver reads the message. CCS and CSP are both examples of synchronous message passing systems. Gilles Kahn's *Process Networks* model is an example of asynchronous message passing.

Computation deals with two questions: *when?* and *how?* The when question addresses how tightly coupled the concurrent activities of processes are with one another. At one extreme, processes execute their computations in lock-step. This approach is referred to as the *synchrony* hypothesis and assumes that processes alternate between phases of simultaneously computing and then simulateously communicating with one another. The opposite extreme assumes that the timing of the computation of one process does not necessarily overlap at all with the computation of other components. Some systems fall in the middle between the synchronous and asynchronous extremes. The

notions of synchrony and asynchrony mentioned in this paragraph should not be confused with those associated with communication. Here we are simply considering whether processes jointly enter their computation or communication phases. A system of processes could adhere to the synchrony assumption yet communicate through asynchronous message passing.

Formal methods have roots in the study of concurrent systems and programming language semantics. Dijkstra can be credited as one of the founding pioneers of both of these communities. Dijkstra [1965] introduced the notion of a critical section. A *critical section* is a region of a program that accesses a shared variable and requires an entry/exit protocol. The entry/exit protocols typically require some sort of mutual exclusion. Dijkstra also introduced guarded commands and non-deterministic control, both of which are instrumental in many process algebras.

Kahn's Process Networks model was first introduced in 1974. A key feature of process networks is the guarantee of determinacy given that certain reasonable constraints are obeyed. Hoare's CSP and Milner's CSP independently offered very similar semantics to one another and were presented in the late 70's. A host of derivatives of both CSP and CCS sprouted in response. Recent activity in formal verification has been valuable within the formal semantics community. As an example, Alur and Henzinger [1996] proposed the *Reactive Modules* model as a system using the synchrony hypothesis but with the possibility of modeling a variety of systems with different communication and computation schemes.

1.1.4 System Level EDA

The system level EDA community brings interoperability and heterogeniety to the table. While process algebras generally incorporate a single model of computation (MoC), system level designers make no such assumptions. System level designers focus at the highest level and therefore require meta-models for describing systems. From the point of view of the system level designer, a complete system requires a variety of communication and computation styles. For this reason, the toolset of the system level designer typically consists of a framework for incorporating heterogeneous semantics. A *framework* is a language for describing languages. An example framework is the *Tagged Signal Model* [Lee and Sangiovanni-Vincentelli, 1997]. The tagged signal model uses a set theoretic approach for describing communication and computation of components.

The use of multiple MoCs increases the richness of a system level design tool, but at the

expense of certain costs: *multiple MoCs require MoC interaction and this interaction must be well defined.* The question of heterogeneous semantics is one of the central concerns of system level EDA and I address this issue heavily in Chapter 3.

System level design is arguably the least well understood design community of those discussed. System level designers borrow techniques from each of the other communities and integrate the fruits of each community's harvest. For this reason, the boundaries of system level EDA are particularly malleable. The indefiniteness of system level design offers both a great challenge as well as a great opportunity.

1.2 Abstracting Component-Based Design

At this point we can digest a broad set of information associated with each of the previous design communities. Indeed, it is worthwhile to apply some of the techniques we've learned to manage the material just presented. One way to organize the information is to consider how reuse ability evolves with the progression of the four communities presented.

- Object-Oriented Programming
  Reuse of objects is enabled.
- Software Engineering
  Reuse of object specifications is enabled.
- Formal Semantics
  Reuse of communication and computation primitives is enabled.
- System Level EDA
  Reuse of models of computation is enabled.

An equally insightful way to organize the communities is to consider their results with respect to syntax and semantics:

- Object-Oriented Programming
  Syntax: Structure over space.

- Software Engineering

  Syntax: Structure over time.

- Formal Semantics

  Semantics: The meaning of object to object interaction.

- System Level EDA

  Semantics: The meaning of MoC to MoC interaction.

Component-based design as an umbrella term can leverage results from each of the design communities above. Based on the layered abstractions according to reuse or syntax/semantics, the component-based design community can select the appropriate level at which to focus. I will be doing precisely this throughout my dissertation.

1.3 Dissertation Outline

This dissertation describes three research accomplishments. The first contribution is presented in Chapter 2 and addresses the difficulty of modelling concurrent systems. Modelling concurrent, computational systems is difficult and there are no graphical tools that sufficiently characterize even the simplest concurrent systems. In Chapter 2, I introduce the diposet. A *diposet* is a formal, mathematical structure that is similar in nature to a partially ordered set. The rigorous characterization of a diposet facilitates mathematical proofs and allows the diposet to serve as a foundation for precise description of semantics. In particular, a diposet is suitable for describing concurrent, computational systems. Using a diposet to represent concurrent systems is distinct from traditional concurrency methods that instead focus on logic. Diposets use an *order-centric* approach that offers insight into the relative timing of events in a concurrent system. A key advantage of the diposet is that it is amenable to simple and intuitive graphical depiction.

An example of a sequential, nested diposet can be found in Figure 1.1. Each node in a diposet represents an event in a concurrent, computational system. The arrows with black arrowheads represent an order relationship between events. In Figure 1.1, the arrow between nodes d and f indicates that event d precedes event f. The arrows with white arrowheads represent a containment relationship between events. The arrow between nodes c and d in Figure 1.1 indicates that event d

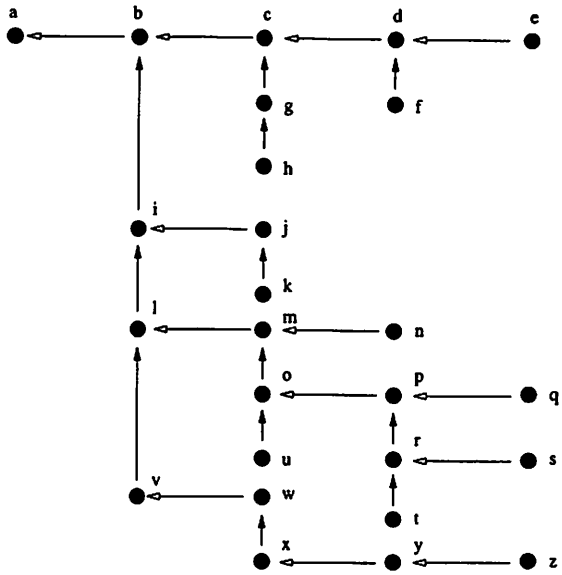

Figure 1.1. A Sequential Nested Diposet

is contained in event *c*.

While the contribution of Chapter 2 is oriented towards the formal semantics community, Chapter 3 presents a system level EDA contribution related to the interaction of heterogeneous models of computation. A difficulty in the execution of a network of components with heterogeneous models of computation is how much the order of execution impacts the computed results. More specifically, how much does the order of data consumption on input channels by message passing components impact the execution of a network of such components. It is known, for example, that Gilles Kahn's Process Networks (PN) model of computation is such that the order of execution of components has no impact on the resulting stream of output data [Kahn, 1974]. It was not clear whether the order of data consumption on input channels would alter the execution output in a network of PN components.

Unfortunately, unlike PN, most other models of computation offer very little insight into the relation between execution order and execution output. My contribution in Chapter 3 is the development of a way for characterizing this relation. I refer to the characterization as reorder invariance. As discussed in Section 3.3.2, a model of computation is *reorder invariant* if the process of reordering a component's communications with neighboring components will not impact the safety or liveness

of the network of components. Chapter 3 leverages the work of Chapter 2 by using diposets.

Chapter 4 presents my third and final contribution by describing my implementation of the work found in Chapter 3. The results of this chapter involve extensive use of software engineering and object-oriented programming techniques. My implementation is part of the UC Berkeley Ptolemy II project under the leadership of Professor Edward A. Lee. Ptolemy II is a modelling and design tool written in the Java™ Programming Language. Chapter 5 concludes the dissertation with references following.

Chapter 2

The Semantics of Concurrent Systems

Semantics is a strange kind of applied mathematics; it seeks profound definition rather than difficult theorems.
- J.C. Reynolds, 1980[1]

Ensuring proper execution of complex, concurrent, computational systems requires great care. Such care can be realized through formal semantics. In this chapter, I present an approach to formal semantics that focuses on the types of relationships that occur between components in a complex, concurrent system. The relationships I am concerned with are the order relationship and the containment relationship. The primary contribution of this chapter is the diposet. I created the diposet to facilitate modeling order and containment in a single mathematical entity. The diposet is compact, precise and amenable to graphical representation.

My emphasis on order and containment is distinct from other expositions on concurrent systems and programming language semantics that instead choose to focus on logic [Andrews, 1991; Magee and Kramer, 1999]. For convenience, I provide an overview of traditional approaches to semantics in Appendix A.

2.1 The Semantics of Concurrent Programs

A concurrent program specifies a set of two or more processes that are coordinated to perform a task and a set of resources that are shared by the processes [Milner, 1989; Andrews, 1991;

[1] R. D. Tennent, *Semantics of Programming Languages*, (Prentice Hall: London, 1991), p. 3.

Magee and Kramer, 1999; Schneider, 1997]. Each process consists of a sequential program made up of a sequence of instructions and this sequence is often referred to as a *thread of control* or *thread* for short. Because each thread is a sequence, the instructions contained within a thread are totally ordered; i.e., given two distinct instructions, a and b, either a is before b or b is before a.

The coordination of threads requires communication between them so that when appropriate, threads may modify their activites based on information from other threads. Communication is accomplished by the shared resources and is realized through *communication instructions* or *synchronization*. In some cases a shared resource might be a conduit through which communication messages are transferred. In other cases a shared resource might be a memory location that multiple threads have read/write access to. While communication is necessary to coordinate threads, undisciplined communication can lead to major problems. If two or more threads access the same shared resource, they can potentially interfere with one another. There are many different types of interference but at its core, *interference* occurs when two or more processes attempt to simultaneously change the state of a shared resource.

Interference is one of the fundamental problems faced in concurrent programming. The possibility of interference results in great emphasis being placed on the ordering of instructions in concurrent programming. If two instructions from different threads modify a common resource, it is essential that one instruction happen before the other so that interference is avoided. Adding order constraints can be effective in preventing interference; unfortunately, lavish use of ordering constraints can result in incomplete execution of a concurrent program. Consider for example two threads, A and B, such that thread A is instructed to wait on a particular instruction of thread B. If thread B decides to not invoke the instruction, perhaps in lieu of a more favorable option, then thread A will end up waiting forever - an undesirable result. For these reasons, concurrent programming can be viewed as the application of techniques and methodologies for enforcing an appropriate level of ordering on a set of multithreaded instructions.

The above discussion of ordering constraints in concurrent programming highlights two fundamental classes of problems: safety and liveness. *Safety* is the property that no *bad* thing happens during the execution of a program [Andrews, 1991; Schneider, 1997]. Interference is an example of a bad thing. *Liveness* is the property that something *good* eventually happens [Andrews, 1991; Schneider, 1997]. Liveness is violated if a program's execution terminates prematurely. All

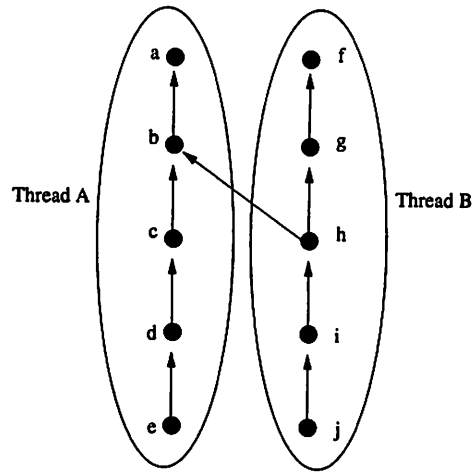

Figure 2.1. Two Communicating Threads

errors found in a concurrent program can be stated in terms of safety and liveness. These definitions of safety and liveness have a foundation in mathematical logic. I prefer to cast the definitions into a framework based on ordering. In the context of ordering, safety is violated in a concurrent program with too few ordering constraints; liveness is violated in a concurrent program with too many ordering constraints. In the following we will discuss methodologies for describing concurrent systems.

2.1.1 Concurrency and Order

Figure 2.1 can be thought of as a simple concurrent program in that it specifies the ordering of instructions in a concurrent program. Thread A consists of instructions a, b, c, d and e while thread B consists of instructions f, g, h, i and j. Note that the arrows indicate instruction ordering such that the arrowhead indicates the preceding instruction; e.g., in the figure, instruction a occurs before instruction b.

The angled arrow in Figure 2.1 indicates an ordering constraint imposed by communication. The arrow does not indicate polarity of the communication but rather serves to illustrate the ordering constraint that the communication imposes. As shown, instruction h must occur after instruction b. Implicitly, instructions i and j must also occur after instruction b. Such constraints between instructions in separate threads would not exist if not for the communication between the threads. Note that it is not possible to determine the relative ordering of all of the instructions in Figure 2.1.

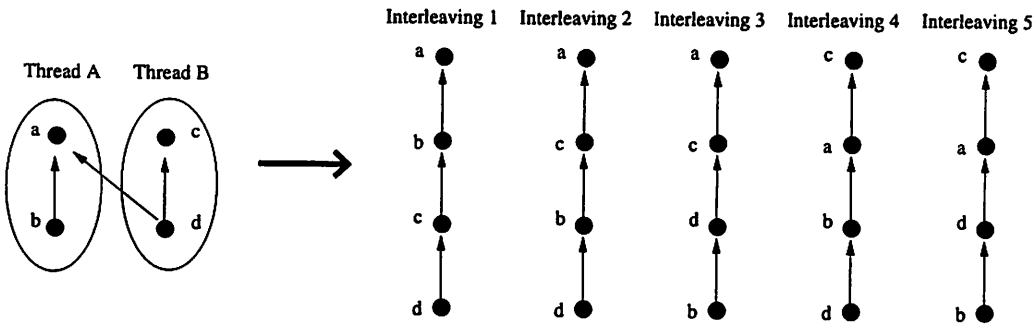

Figure 2.2. Sequential Interleavings

In particular, we can not determine whether instruction c occurs before or after instruction g. In general, a concurrent program will specify ordering constraints on only a subset of thread instructions. If all instructions between distinct threads were totally ordered, the result would be a single thread.

The absence of an ordering specification is usually taken to indicate that relative ordering is inconsequential. In other words, the specification in Figure 2.1 indicates that instructions c and g can be realized as c followed by g or g followed by c; either realization is allowed and the choice is arbitrary. The notion of arbitrary ordering of unordered instructions can be applied to all of the instructions of a set of threads and results in an interleaving. An *interleaving* is a sequential realization of a set of threads that does not violate any of the ordering constraints of the threads. Figure 2.2 is an example of an interleaving. Note that threads A and B can be interleaved in either of the five ways shown. What this means is that if the concurrent program specified by Figure 2.2 were executed, any of the five sequential orderings could represent the actual execution. In fact, each execution can randomly turn out to be any of the five orderings even without changing parameters! Multiple interleavings facilitate both apparent and actual concurrency. In both cases, the goal is to ensure that the sequential realization/model is correct; i.e., equivalent to what the designer wants.

Unfortunately the existence of multiple interleavings for a single concurrent program specification leads to a major difficulty with concurrent programming. The size of the set of interleavings for a given program is typically unmanageably large. In general, given N threads that each execute M distinct non-communication instructions, there are

$$\frac{(NM)!}{(M!)^N}$$

possible interleavings. Five threads with ten non-communication instructions result in over $4.83 \times$

10^{31} possible interleavings.

2.1.2 Representing Concurrent Systems

A key difficulty in designing and implementing concurrent systems is the absence of effective tools for specifying and representing such systems. Representation tools are extremely important in the design process. Representation tools aid designers in communicating with each other about a given design as well as in finding errors. Graphical representation tools are especially helpful in designing software. For example, graphical representation is the primary thrust of the UML movement [Booch, 1994; Rumbaugh *et al.*, 1991]. I will consider graphical representation tools for concurrent programming. In previous sections I have shown several figures (e.g., Figures 2.1 and 2.2) in an attempt to graphically represent concurrent programs. Unfortunately, these graphs have significant shortcomings.

In this section, I survey four approaches that are used to graphically model concurrent systems and discuss the pros and cons of each. The four approaches I survey are partially ordered sets, interval orders, graphs and Petri nets. I chose these four modeling techniques because of their widespread use and mathematical rigor [West, 1996; Neggers and Kim, 1998; Peterson, 1981]. My metric for measuring these four approaches will be their ability to represent both *containment* and *order* simultaneously. I will show that using this metric, each of these techniques falls short. I will then propose a new formalism for more effectively representing concurrent systems with containment and order; I refer to this formalism as a *diposet*.

Partially Ordered Sets

Definition 2.1. PARTIALLY ORDERED SET

Let X be a set. A **partial order**, R, on X is a binary relation that is reflexive, anti-symmetric and transitive. An ordered pair (X, R) is said to be a **partially ordered set** or a **poset** if R is a partial order on the set X. □

The three conditions on R hold for all $x, y, z \in X$ as follows

- Reflexive: $(x, x) \in R$
- Anti-Symmetric: $(x, y) \in R, (y, x) \in R$ implies $x = y$

- Transitive: $(x, y) \in R, (y, z) \in R$ implies $(x, z) \in R$

I will write $\leq$ for R such that $(x, y) \in R$ if and only if $x \leq y$; similarly $(y, x) \in R$ if and only if $y \leq x$.[2] Other common notations for R include $\sqsubseteq$ and $\preceq$. If $x \leq y$ or $y \leq x$ we say that x and y are *comparable*. If x and y are *incomparable* we write $x \parallel y$. We say that y *covers* x if $x \leq y$ and there is no element $z \in X$ such that $x \leq z \leq y$. The set X of a partially ordered set is called the *ground set*. If all elements of the ground set are comparable, then the set is called a *totally-ordered set* or a *chain*. If none of the elements of the ground set are comparable, then the set is called an *anti-chain*. The *up-set*, $Q \subseteq X$, of element y is defined such that $x \in Q \implies y \leq x$. We write the up-set of element y as y_{up-set}. The *down-set* is defined in a similar fashion.

Partially ordered sets can be graphically represented by Hasse diagrams. A *Hasse diagram* is a graph in which each vertex or point corresponds to one element of the ground set. An arrowed-line is drawn from point x to point y if y covers x.[3] If we interpret the partial order as representing precedence such that $x \leq y$ if y precedes x, then Figures 2.1 and 2.2 are examples of Hasse diagrams. For clarification, note that b is covered by a in Figure 2.1.

Program 2.1. EXAMPLE SEQUENTIAL CODE

```
public void start() {
    a = val;
}
public void compute() {
    do();
    undo();
}
public void finish() {
    a = 0;
}
```

It would seem that partially ordered sets are a natural way to express the ordering relationships in concurrent programming systems. If we let each element of a set represent a method or

[2] Note that I have chosen to use reflexive notation so that $\leq$ reads "less than or equal." Alternatively I could use irreflexive notation such as $<$, read "less than." Reflexive notation as given in the definition of partially ordered set defines the relation, R, as a *weak inclusion* while irreflexive notation defines the relation, R, as a *strong inclusion*. In some cases the relation associated with strong inclusion is called an *order* as opposed to a *partial order*.

[3] Alternatively, Hasse diagrams can be drawn with arrows from x to y if x covers y. Pay attention to the orientation when viewing a Hasse diagram.

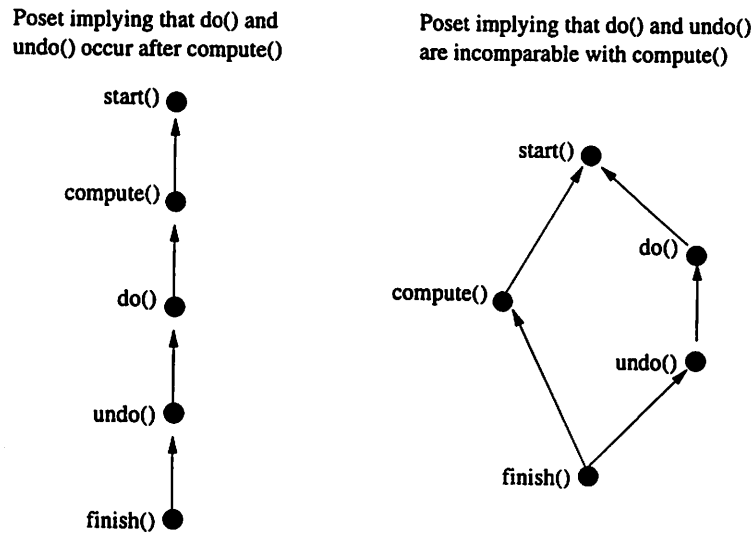

Figure 2.3. Insufficient Poset Representations of Program 2.1

function, then partially ordered sets can represent a program of method calls. Unfortunately, posets are not expressive enough to accurately represent even very simple programs. Consider the code fragment found in Program 2.1 (written in Java™ syntax) where we assume that the methods *do*() and *undo*() do not call any other methods.

Assume a thread that invokes *start*(), *compute*() and then *finish*(). A poset is not able to model the complete relationship between *start*(), *compute*(), *finish*(), *do*() and *undo*(). More specifically, how do we relate *do*() and *undo*() to *compute*(). Both of the Hasse diagrams in Figure 2.3 are less than accurate. The method *compute*() is neither before or after *do*() and *undo*(), yet to say that *compute*() is incomparable to *do*() and *undo*() is not quite right either. The method *compute*() is non-atomic in that it contains *do*() and *undo*(). *The problem illustrated by this example is that partially ordered sets can not represent both the notion of order and the notion of containment.* Order is necessary to relate *start*() and *compute*() while containment is necessary to show that *compute*() is non-atomic.

Interval Orders

An interval order is a special class of partially ordered sets. The name implies that interval orders are amenable to graphical representation, and on the surface an interval order seems suitable for de-

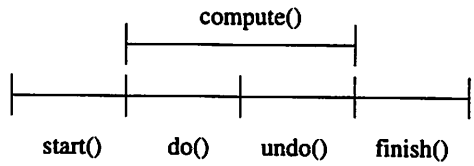

Figure 2.4. An Interval Order Representation of Program 2.1

scribing elements that are non-atomic. Nevertheless, interval orders can not describe containment and indeed they are less expressive then posets.

Definition 2.2. INTERVAL ORDER

A poset $(X, \leq)$ is an **interval order** if there is a function $I : X \rightarrow [i(x), t(x)]$ where $i(x), t(x) \in \Re$ so that $x < y$ in X iff $t(x) < i(y)$ in $\Re$. □

An interval order corresponding to Program 2.1 is shown in Figure 2.4. The primary problem with interval orders is that they can not represent certain posets. In particular, while interval orders can represent incomparable points, they can not represent incomparable chains. Figure 2.5 illustrates the inability of interval order to represent chains. Given that the intervals of a, b and c are as shown, where do we place the interval for d? Interval d must intersect a and b without intersecting c: an impossible constraint. Hence, an interval order must be free of the poset shown in Figure 2.5. This precludes a large set of posets and renders interval orders insufficient for our purposes.

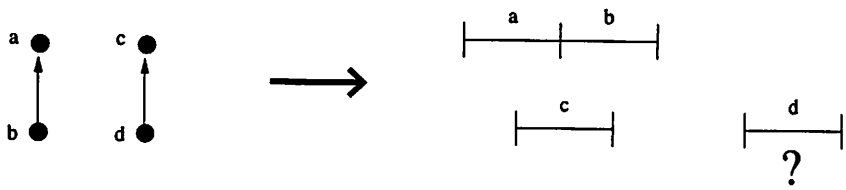

Figure 2.5. A Parallel Chain Poset With No Corresponding Interval Order

Graphs

A graph, as its name implies, is a mathematical structure that naturally lends itself to visual representation. Graphs are used extensively within the field of computer science. Examples include the representation of language grammars and network connectivity diagrams.

Figure 2.6. An Example Graph and Directed Graph

Definition 2.3. GRAPH

A **graph** G with n **vertices** and m **edges** consists of a **vertex set** $V(G) = \{v_1, ..., v_n\}$ and an **edge set** $E(G) = \{e_1, ..., e_m\}$. Each edge is a set of two (possibly equal) vertices called its **endpoints**. We write uv for an edge $e = \{u, v\}$. If $uv \in E(G)$, then u and v are **adjacent**. □

Graphs are illustrated by diagrams in which a point is assigned to each vertex and a curve is assigned to each edge such that the curve is drawn between the points of the edge's endpoints. An example graph is shown in Figure 2.6 (on the left). In some cases, it is useful to add directionality to the edges of a graph. A *directed graph* models such directionality and is defined in the following definition. An example directed graph can be found in Figure 2.6 (on the right) where arrowed curves indicate direction.

Definition 2.4. DIRECTED GRAPH

A **directed graph** is a graph in which each edge is an ordered pair of vertices. We write uv for the edge (u, v) with u being the **tail** and v being the **head**. □

The definitions above are consistent with that used in many texts on the subject [West, 1996; Chen, 1997]. Note that the edge set of a directed graph is simply a relation; e.g., $E(G) \subseteq V(G) \times V(G)$. Focusing on the fact that the edge set of a directed graph is a relation emphasizes the shared traits between directed graphs and many other mathemathical structures. In particular, a relation-oriented definition of directed graph makes it clear that a partially ordered set is a special case of a directed graph.

Graphs and directed graphs both have definitions for several useful characteristics. For our purposes, two particularly useful definitions are path and cycle. Informally, a *path* in a graph is an ordered list of distinct vertices $v_1, ..., v_n$ such that $v_{i-1}v_i$ is an edge for all $2 \leq i \leq n$. A path may

consist of a single vertex. A *cycle* is a path $v_1, ..., v_n$ in which $v_n v_1$ is an edge. The *length* of a path (cycle) $v_1, ..., v_n$ is n.

In their basic form, directed graphs and graphs are insufficient for modelling software systems for reasons similar to those cited for partially ordered sets. A directed graph only has a single relation on its set of vertices. A single relation will not sufficiently describe both the order and containment characteristics that are found in the typical object-oriented software program since order and containment are distinct qualities that require individual representation.

Petri Nets

Carl Adam Petri developed Petri theory with a concern for asynchronous communication between components and the causal relationships between events. The basic theory from which Petri nets developed can be found in the dissertation of Carl Petri [Petri, 1962]. The definition of a Petri net structure is found below.

Definition 2.5. PETRI NETS

A **Petri net structure**, C, is a four-tuple, $C = (P, T, I, O)$. $P = \{p_1, p_2, ...p_n\}$ is a finite set of **places**, $n \geq 0$. $T = \{t_1, t_2, ..., t_m\}$ is a finite set of **transitions**, $m \geq 0$. The set of places and the set of transitions are disjoint, $P \cap T = \emptyset$. $I : T \rightarrow P^\infty$ is the **input** function, a mapping from transitions to bags[4] of places. $O : T \rightarrow P^\infty$ is the **output** function, a mapping from transitions to bags of places. □

Tokens can reside in (or are assigned to) the places of a Petri net. A *marking* μ is an assignment of a nonnegative number of tokens to the places of a Petri net. The number of tokens that may be assigned is unbounded. Hence, there are an infinite number of markings for a Petri net.

A Petri net executes by firing its transitions. A transition *fires* by removing tokens from its input places and creating new tokens in its output places. A transition may fire if it is *enabled.* A transition is enabled if each of its input places contains at least as many tokens as connection arcs from the place to the transition. Tokens that cause a transition to be enabled are called *enabling tokens.* When a transition fires, it removes all of its enabling tokens from its input places and then deposits into its output places *one* token for each output arc.

[4] A bag is like a set except that it allows multiple occurrences of elements.

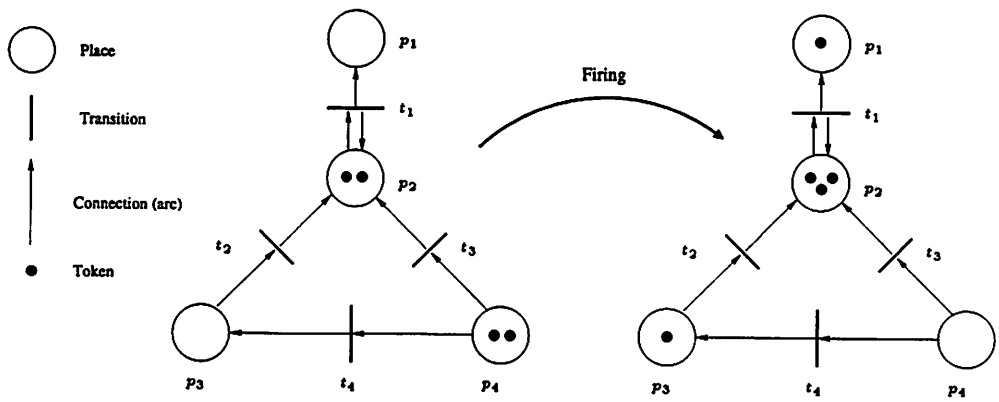

Figure 2.7. An Example Petri Net With Firing

A Petri net is often graphically displayed as shown in Figure 2.7. In fact, a Petri net is a directed, bipartite multigraph. A *bipartite graph* is a graph that consists of two classes of nodes such that each edge connects a node from one class to a node in the other class. In a Petri net, every arc (edge) connects a place to a transition. A *multigraph* is a graph that allows multiple edges from one node to another. As shown in Figure 2.7, several arcs may connect a place/transition pair.

A Petri net is not sufficient for representing order and containment. Even though it consists of two classes of nodes, its bipartite nature would constrain the order and containment relations to occur adjacently. It is not obvious how the containment relation could be graphically displayed using a Petri net, thus making it difficult to represent hierarchy. In addition, Petri nets assume an asynchronous style of communication. While it is true that asynchronous communication can serve as a foundation for synchronous communication [Brookes, 1999], asynchronous primitives can not represent synchronous communication in a succinct manner.

2.1.3 Diposets

In the previous sections I have summarized several mathematical formalisms and critiqued their usefulness in the context of describing object-oriented software systems. In each case, I showed that these formalisms were not sufficient for describing the richness of simple software systems. I have developed a new mathematical structure that I refer to as a *diposet*. In the remainder of this section I will define diposet and in subsequent sections I will make a case that diposets are suitable for robustly

describing software systems.

The key observation with each of the mathematical structures presented thus far is that a single relation is not satisfactory for describing software systems. One way to deal with this problem is to use a pair of structures for describing software systems. Consider a pair of directed graphs G_1 and G_2 such that $V(G_1) = V(G_2)$. For convenience I will refer to this *paired directed graph* as $\{G_1, G_2\}$. Associated with the pair of directed graphs are two relations, $E(G_1)$ and $E(G_2)$. Each relation spawns various characteristics. For example, $\{G_1, G_2\}$ may have two distinct paths, p_1 and p_2, such that p_1 is associated with $E(G_1)$ and p_2 is associated with $E(G_2)$.

A paired directed graph $\{G_1, G_2\}$ offers the beginnings of a tool equipped for describing a variety of systems that require two types of relations (e.g., order and containment) over a set of elements. In order to make a paired directed graph completely useful, more structure must be added. I created the diposet to fill the need for just such a structure.

Definition 2.6. DIPOSET

Let X be a set. A **diorder** on X is a pair of binary relations on X referred to, respectively, as the **order relation**, R_O, and the **containment relation**, R_C, such that R_C and R_O are both reflexive, anti-symmetric and transitive. For all $x, y \in X$, if $(x, y) \in R_O$ then $(x, y), (y, x) \notin R_C$. Similarly, for all $x, y \in X$, if $(x, y) \in R_C$ then $(x, y), (y, x) \notin R_O$. A set X that is equipped with a diorder is said to be a **diposet** and is denoted (X, R_O, R_C). □

It is immediately obvious that a diposet is a special case of a paired directed graph. It is also clear that (X, R_O) and (X, R_C) are both partially ordered sets with a common ground set. The ground set X of a diposet is equivalent to the set of vertices V $(= V(G_1) = V(G_2))$ of a paired directed graph. The containment and order relations of a diposet, $\{R_C, R_O\}$, are equivalent to the two sets of edges in a paired directed graph $\{E(G_1), E(G_2)\}$.

We say that the ground set, X, of a diposet consists of *events*. The order relation determines how events are ordered with respect to one another. Consider events $a, b \in X$. If $(a, b) \in R_O$ then we say that $a \leq_O b$. I.e., event a precedes event b. If $(a, b), (b, a) \notin R_O$ then we say that $a \|_O b$; e.g., a and b are *incomparable*. The containment relation facilitates non-atomic events and event containment. An event is non-atomic if it contains another event. If $(a, b) \in R_C$ then we say that $a \leq_C b$. I.e., event b is non-atomic and contains event a. If $(a, b), (b, a) \notin R_C$ then we say that $a \|_C$

b; e.g., a and b are *mutually non-inclusive*. Note the distinction between incomparable and mutually non-inclusive. In the context of diposets, incomparability refers to the order relation; mutual non-inclusiveness refers to the containment relation. Up-set is defined both for order and containment and is denoted as such; e.g., $up - set_O$ and $up - set_C$ (similar definitions exist for down-set). An *order (containment) path* in a diposet is a sequence of events $e_1, ..., e_n$ such that $e_1 \leq_O ... \leq_O e_n$ ($e_1 \leq_C ... \leq_C e_n$).

Note that a direct result of Definition 2.6 is that $R_O \cap R_C = \emptyset$. The fact that R_O and R_C of a diposet do not intersect leads to two results that hold for all diposets:

i) An event can not contain an event that it precedes or that it is preceded by.

ii) An event can not be contained by an event that it precedes or that it is preceded by.

The disjointness of R_O and R_C in a diposet serves as one of the key distinctions between a diposet and a paired directed graph. In a paired directed graph $\{G_1, G_2\}$ it is sufficient for G_1 and G_2 to share a common set of vertices but there is no constraint on the two sets of edges associated with a paired directed graph. For example, it is completely admissable for the edge sets of a paired directed graph to be identical; i.e., $E(G_1) = E(G_2)$. The intuition behind the disjointness of R_O and R_C is that each relation should provide orthogonal information. If the order and containment relations of a diposet provide redundant information, then the usefulness of distinct relations is undermined.

Partially ordered sets are graphically represented via Hasse diagrams. Hasse diagrams serve as a simple way to represent posets with directed graphs where an arrow is drawn from element a to element b if b covers a. Diposets utilize Hasse diagrams as well, with the notion of covering being extended to containment. I.e., b covers a if there does not exist q such that $a \leq_C q \leq_C b$. Given that b covers a according to a containment relation, we say that b is a *cover container* of a. To accomodate both relations in a diposet, diposet Hasse diagrams require two types of arrows. I will use a black arrow head to represent the the order relation and a white arrow head to represent the containment relation. Figure 2.8 displays an example diposet. From this figure we can see that event a is contained by event c and is incomparable to event d. Event b is preceded by event a and event f is preceded by event d.

In many systems, the kind of containment that can be modelled by a diposet is not sufficiently constrained. Most software systems require that containment be nested. I add this additional

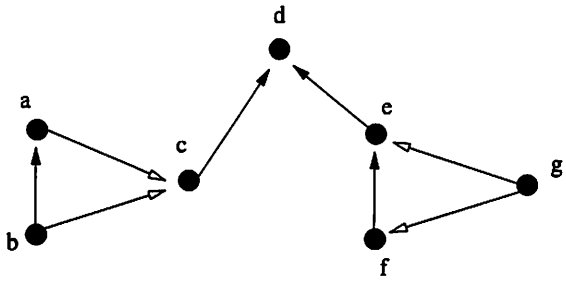

Figure 2.8. An Example Diposet

constraint with the following principle.

Definition 2.7. THE NESTED CONTAINMENT RULE

A diposet, (X, R_O, R_C), satisfies the **nested containment rule** if $\forall x, y, z \in X$, the following conditions are adhered to:

Condition I: If $x \parallel_C y$, then $(z \leq_C x \implies z \not\leq_C y)$ and $(z \leq_C y \implies z \not\leq_C x)$.

Condition II: If $x \leq_O y$, then $(z \leq_C x \implies z \leq_O y)$ and $(z \leq_C y \implies x \leq_O z)$.

A diposet that satisfies the nested containment rule is called a **nested diposet.** □

In plain English, Condition I says that an event can have at most one cover container. Condition II says that each event precedes (is preceded by) each event that its container events precede (are preceded by). An example nested diposet can be found in Figure 2.9.

A key distinction between the Hasse diagrams of diposets and nested diposets can be seen when comparing Figures 2.8 and 2.9. In Figure 2.9 it is implicit that $a \leq_O d$ by Condition II of Definition 2.7. In a similar fashion, we see that $g \leq_O e$. These assumptions can not be made in a general diposet, and hence in Figure 2.8 a and d are incomparable while in Figure 2.9 they are comparable. This distinction between the Hasse diagram for diposets versus nested diposets requires that one clearly state which type of diagram is being displayed, so that confusion can be avoided. Nested diposets are generally more useful than diposets. For example, most computer programs have a nested structure. For this reason, I will focus solely on nested diposets from this point on and I will use the term nested diposet and diposet interchangeably to mean nested diposet. Several interesting results

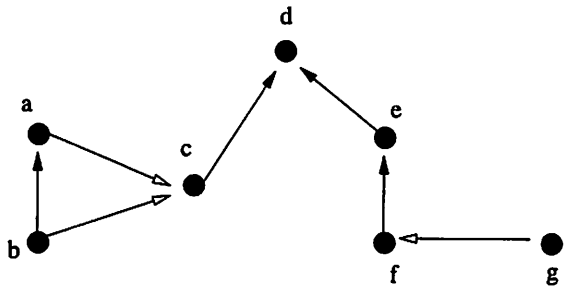

Figure 2.9. An Example Nested Diposet

can be derived based on the nested containment rule, as the Weighted Chain Theorem[5] illustrates.

Theorem 2.1. WEIGHTED CHAIN THEOREM

For nested diposet, (X, R_O, R_C)*, if there exists* $x_0 \in X$ *s.t.* $\forall x \in X, x_0 \leq_C x$ *then all events in* X *are incomparable.*

Proof by Contradiction Suppose that not all events in X are incomparable. Then there must exist two events $y, z \in X$ such that either $y \leq_O z$ or $z \leq_O y$. Consider the former case. We have $y \leq_O z$. Since $x_0 \leq_C y$ by the theorem statement, then we know from Condition II of Definition 2.7 that $x_0 \leq_O z$. Again referring to the theorem statement we have $x_0 \leq_C z$. This contradicts Definition 2.6 since an event can not be contained by an event that it precedes; e.g., the disjointness of R_O and R_C has been violated. Hence our supposition was false. The alternative cases follow in a similar manner. □

In considering the nested containment rule, it is important to be clear on what it does not imply. In particular, note that for a given nested diposet, (X, R_O, R_C), with $x, y, z \in X$

$$x \leq_O y \leq_C z \not\Longrightarrow x \leq_O z$$

The simplest counter example that satisfies the above statement is the following three event nested

[5]The intuition behind the name "Weighted Chain" is that if ever a subset of a diposet contains a minimum contained element (e.g., an element contained by all other members of the subset), then the minimum forces the elements in the subset to be pulled down like a hanging chain with a weight tied at the bottom.

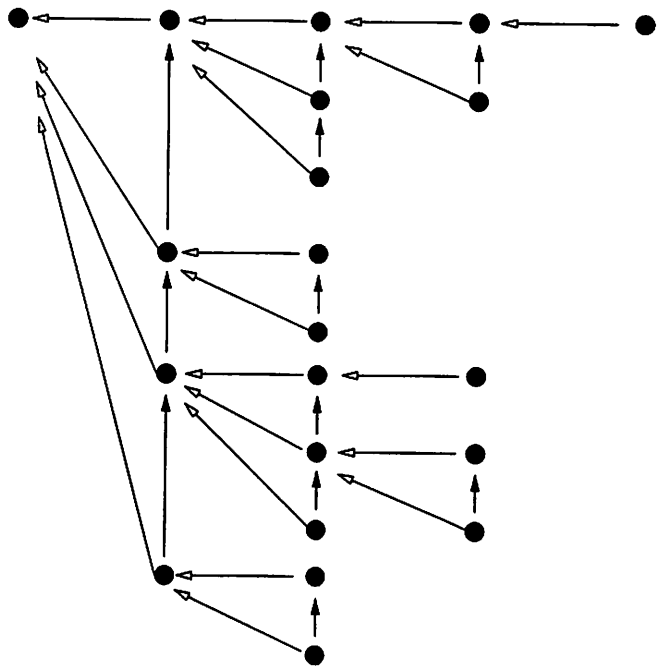

Figure 2.10. A Sequential Nested Diposet (Explicit Representation)

diposet, $x, y, z \in X$:

$$x \leq_C z$$

$$y \leq_C z$$

$$x \leq_O y$$

Note that if $x \leq_O y \leq_C z \Longrightarrow x \leq_O z$ then Def. 2.6 would be violated; i.e., $R_O \cap R_C \neq \emptyset$.

Definition 2.8. SEQUENTIAL NESTED DIPOSET (THREAD)

A **sequential nested diposet** or **thread** is a nested diposet, $X_{ND} = \{X, R_O, R_C\}$, for which $\exists x_0 \in X$, called the **maximum container** of X, such that $x \leq_C x_0, \forall x \in X$ and such that $\forall x, y \in X$, if x and y have a common cover container, then $x \leq_O y$ or $y \leq_O x$. □

An example thread is shown in Figure 2.10. It is drawn in an explicit graphical format. Explicit graphical format will be explained in Section 2.1.4.

Given that each event in a thread has at most one cover container,[6] it is useful to develop

[6] A characteristic that is true of all nested diposets.

a notion of depth. We define depth recursively. The *depth* of the maximum container in a thread is 0. For any event x contained within a thread other than the maximum container, the *depth* of x is the depth of its cover container plus 1.

Theorem 2.2. CONNECTED THREAD THEOREM

Any two events in a sequential nested diposet (thread) are either related by the order relation or the containment relation but never both.

Direct Proof Consider any two events x, y contained in a sequential nested diposet with ground set X. We know that x and y can not be related by both the order and containment relations by Definition 2.6. In terms of the rest of the proof there are three possible cases as listed below.

i) If x and y are not mutually non-inclusive then x and y must be related by the containment relation and we are done.

ii) If x and y are mutually non-inclusive and have a common cover container then by Definition 2.8 x and y must be related by the order relation and we are done.

iii) If x and y are mutually non-inclusive and do not have a common cover container then apply the following step. Select the event (either x or y) that has the greatest depth.[7] Without loss of generality assume that x has a greater depth then y. If the cover container of x is comparable to y than we are done by virtue of Condition II of Definition 2.7. Otherwise, repeat this step.

□

Theorem 2.3. ACYCLIC DIPOSET THEOREM

A diposet can not contain order or containment cycles of length 2 or more.

Proof by Contradiction Suppose that a diposet (X, R_O, R_C) contains an order cycle of length 2 or more. Then there must exist a path $e_1, ..., e_n$ with $e_1 \neq ... \neq e_n$ such that $e_1 \leq_O ... \leq_O e_n \leq_O e_1$. By the anti-symmetry property of partially ordered sets, this implies that $e_1 = ... = e_n$. Hence, our supposition must be false and the diposet does not contain a cycle of length 2 or more. Similar reasoning applies to containment cycles of length 2 or more. This completes our proof. □

Note that in general a paired directed graph can contain both order and containment cycles

[7] If x and y have the same depth then arbitrarily choose one or the other.

of any length. As will be shown in subsequent sections, the existence of a cycle indicates that a system can not be modelled by a diposet but perhaps can be modelled by a paired directed graph.

In many situations it is useful to label the events of a diposet. For example, multiple events in a diposet's ground set may each share a common label indicating that they represent a common entity or labels may serve as a basis for relating a class of events. A labelling function facilitates this process.

Definition 2.9. LABELLED DIPOSETS

A **diposet labelling function**, $f : X \rightarrow L$, maps the ground set of a diposet to a **label set**, L. A diposet that is associated with a labelling function and label set is referred to as a **labelled diposet**.

□

Many of the example diposets that have been previously shown were labelled. For example, in Figure 2.9 the label set is $L = \{a, b, c, d, e, f, g\}$ and the labelling function is bijective.

2.1.4 Types of Order

Thus far I have presented three partially ordered structures: diposet, nested diposet and sequential nested diposet (thread). Nested diposets and sequential nested diposets are especially important for our purposes because of the abundance of nested structures in the field of computer science. When considering a nested diposet, it is always the case that the order relation of a nested diposet can be separated into two subsets: $C_O \cup T_O \subseteq R_O$. C_O is referred to as the set of *communication order relations* and T_O is referred to as the set of *threaded order relations*. For any two events, $x, y \in X$, $C_O(x, y)$ represents a subset of order relations that are associated with x and y; i.e., $C_O(x, y) \subseteq C_O$. In a similar fashion $T_O(x, y) \subseteq T_O$.

The threaded order relation T_O relates events that are in the same thread. For any nested diposet (X, R_O, R_C), we have $T_O(x, y) = \emptyset$ if $x, y \in X$ are not part of the same sequential nested diposet. The communication order relation C_O relates events that are not in the same thread. We have $C_O(x, y) = \emptyset$ if $x, y \in X$ are part of the same sequential nested diposet.

A simple communication order relation is $C_O(x, y) = (x, y)$. As will be shown in Section 2.2.4, this order relation is equivalent to asynchronous communication between two threads in which the thread containing event y receives from the thread containing event x. A more elaborate

communication order relation is

$$C_O(x,y) = \begin{array}{l} \{(x,y')|y' \in y_{up-set_O}\} \cup \{(y',x)|y' \in y_{down-set_O}\} \\ \cup\{(x',y)|x' \in x_{down-set_O}\} \cup \{(y,x')|x' \in x_{up-set_O}\} \end{array} \quad (2.1)$$

I will show that the communication order relation of Equation 2.1 is a precise characterization of synchronous message passing communication in which x and y represent the sending/receiving events of two communicating threads (See Section 2.2.4).

Given nested diposet (X, R_O, R_C) we can write

$$R_O = \left[\bigcup_{(x,y)\in X\times X} C_O(x,y)\right] \bigcup \left[\bigcup_{(x,y)\in X\times X} T_O(x,y)\right] \quad (2.2)$$

We can leverage the dichotomy found in Equation 2.2 to simplify our nested diposet Hasse diagrams. Recall that Figure 2.10 is drawn in an explicit graphical format. By *explicit* I mean that all cover container relationships are explicitly shown. Alternatively a sequential nested diposet can be represented in an implicit graphical format. The implicit graphical representation of a nested diposet relies on the following three rules.

i) Order relations associated with T_O are drawn with a vertical line.

ii) Order relations associated with C_O are drawn with a non-vertical curve.

iii) Containment relations are drawn with a horizontal line.

An example sequential nested diposet that is drawn in an implicit format is shown in Figure 2.11. The thread drawn in Figure 2.10 is identical to that of Figure 2.11. The only difference is that the former is represented explicitly while the latter is represented implicitly. In the remainder of this dissertation, I will use implicit representation of sequential nested diposets.

2.2 Diposets and Concurrent Programming

Diposets are amenable to modeling a wide variety of systems including manufacturing schedules, distributed transactions and hardware systems. Given our interests, we will use nested diposets to model concurrent software systems. In a concurrent system, the ground set of our nested diposet consists of method invocations or code blocks. The label of an invocation is simply the

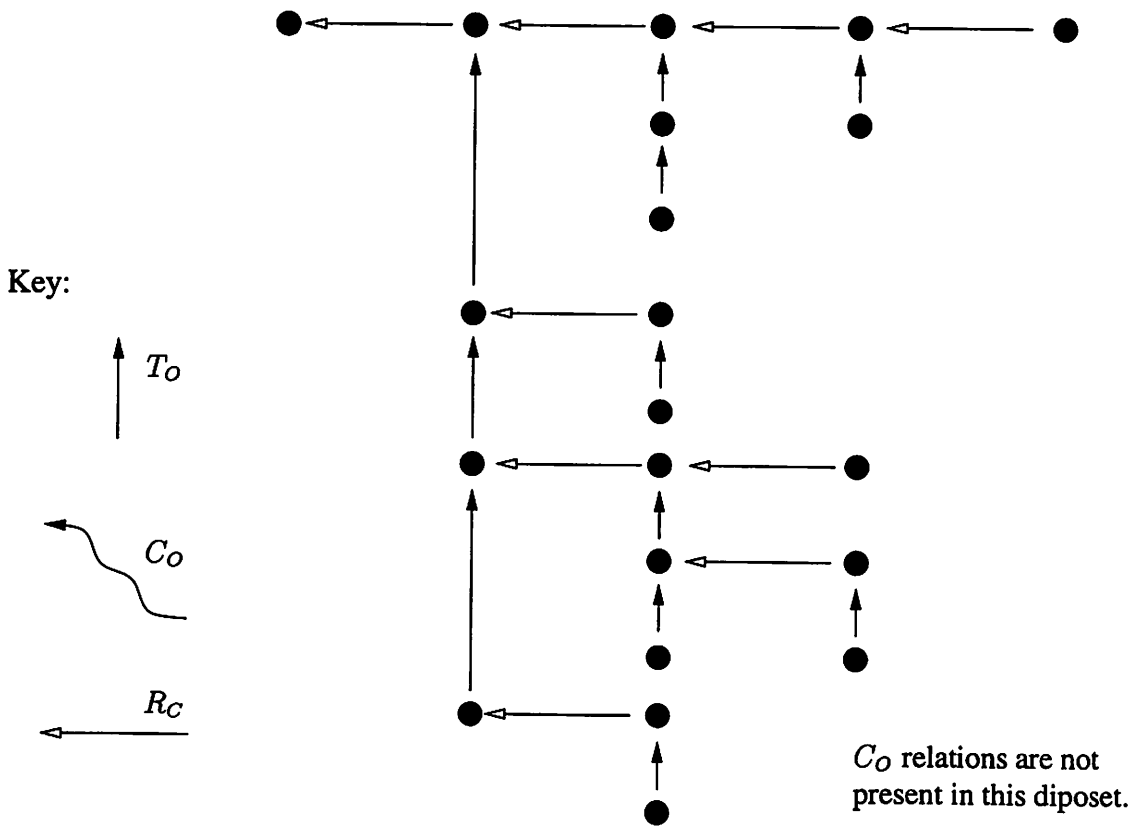

Figure 2.11. A Sequential Nested Diposet (Implicit Representation)

method's name. Hence, multiple invocations of a method share a common label. Note that declaring a nested diposet's ground set as consisting of method invocations can accomodate a rich class of programming constructs including recursion and software objects.

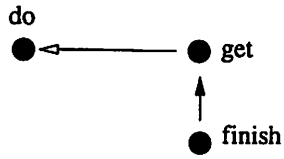

Figure 2.12. A Nested Diposet Corresponding To Program 2.2

If the body of method a contains an invocation of method b, then we say that $b \leq_C a$. If the body of method a precedes method b (as in method a returns prior to the invocation of method b) then we say that $b \leq_O a$. Consider the code fragment shown in Program 2.2. Here we see both the notion of containment and order. The methods $get()$ and $finish()$ are contained within the method $do()$. E.g., $get \leq_C do$ and $finish \leq_C do$. In addition, the method $finish()$ is ordered to occur after the method $get()$. E.g., $finish \leq_O get$. A single invocation of the method $do()$ would result in the thread displayed in Figure 2.12.

Program 2.2. SAMPLE METHOD CALLS

```
public void do() {
    get();
    finish();
}
public void get() {
    z1 = x + y;
}
public void finish() {
    z2 = z1++;
}
```

In some cases a nested diposet or a diposet will not be sufficient for describing a software system. In particular, as Theorem 2.3 (the Acyclic Diposet Theorem) declares, a diposet can not contain non-trivial cycles. In cases where inclusion of a cycle is crucial, the structure of a diposet can be relaxed and transformed into a paired directed graph. Paired directed graphs are amenable to describing cycles because they are not beholden to anti-symmetry.

2.2.1 Safety and Synchronization

Recall that the key problems of concurrent programs fall into two classes: safety and liveness problems. Let us consider how nested diposets can model these constucts. Safety is solved by applying mutual exclusion to the critical section of code that should not be simultaneously accessed by multiple threads. A common way to guarantee safety in a concurrent program is to require lock synchronization to code blocks. Only one process can synchronize with a given lock and thus access to the block of code will necessarily be mutually exclusive.

Safety via lock synchronization can be represented with containment and order relationships. Locks apply to blocks of code, thus we can think of a realized lock as an invoked method. In nested diposet terms, the code that a realized lock synchronizes is contained by the lock. We must make sure that multiple realizations of the same lock are not invoked simultaneously. This is accomplished by ordering the lock invocations. This process is illustrated in Program 2.3 (written in psuedo Java™code) and Figure 2.13. Note that the `synchronized` keyword means that the lock for the corresponding method is an instatiation of the `Obj` class. A nested diposet showing a possible interleaving of calls to methods *do* and *undo* that satisfies the synchronization lock constraints is given in Figure 2.13.

Program 2.3. SYNCHRONIZED METHOD CALLS

```
public class Obj {
    public synchronized void do() {
        modify();
        change();
    }
    public synchronized void undo() {
        change();
        modify();
    }
    private void change() {
        // Atomic; contains no methods
    }
    private void modify() {
        // Atomic; contains no methods
    }
}
```

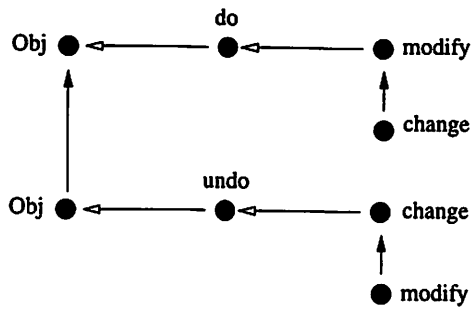

Figure 2.13. A Possible Interleaving Of Calls To `do()` And `undo()` In Program 2.3

2.2.2 Liveness and Deadlock

The result of liveness problems within concurrent, computational systems are perhaps the most recognizable difficulties that the typical computer user must face. Liveness is closely associated with the inter-dependencies and relative speeds of autonomous threads. Relative thread speeds are tied to the thread scheduling algorithms of operating systems and such algorithms are typically beyond the control of software developers. For this reason, liveness problems have an inherently non-deterministic nature from the perspective of the software developer. Although the problems associated with the absence of safety can be just as devastating as those associated with the absence of liveness, safety is much easier to maintain than liveness and thus for most computational systems safety is not a major issue.[8] Doug Lea categorizes liveness into four groups [Lea, 1997].

I) *Contention* occurs when several processes wait on resources but only a subset of the processes gain the resources. Contention is fundamentally related to fairness and is generally a deterministic problem in that it is based on the thread/process scheduling algorithm being used.

II) *Dormancy* occurs when a waiting thread is not notified that the condition it is waiting on becomes true. This problem is relatively easy to solve with well placed "wake up" mechanisms. For example, in the Java™programming language a `notify()` or `notifyAll()` method would be used. Dormancy is typically deterministic in that the wake up mechanisms are usually not dependent upon a particular interleaving of threads.

[8] While corrupt data (the result of safety problems) are farely rare, who among us has not witnessed the *blue screen of death*?

III) *Deadlock* occurs when a cycle of processes are mutually dependent upon each other at the same time. More precisely, N processes each wait on exclusive access to one of N resources while simultaneously holding exclusive access to another one of the N resources such that each process is awaiting access to a distinct resource. Deadlock is typically non-deterministic in that it is dependent upon the relative speeds of the processes acquiring the resources.

IV) *Premature Termination* occurs when a process ceases operation unexpectedly without properly notifying the other processes in the concurrent system. Such termination can result in both safety and liveness problems for the remaining processes. Premature termination is akin to a reversal of dormancy and is relatively easy to solve given appropriate exception handling.

Each of the types of liveness problems can cause a concurrent program to halt in an undesirable manner. While they are all challenging to deal with, in my experience deadlock stands out in a class of its own. In the best case scenario, deadlock is tied to the interleaving of the threads involved. This means that deadlock will non-deterministically occur based on the relative speeds of the threads and how the relative speeds impact thread interleaving. In the worst case scenario deadlock is not dependent upon relative thread speeds. In this case deadlock is intrinsic in the semantics of the communicating threads and there is no hope of evasion. Hence, in the worst case scenario there is nothing one can do while in the best case scenario one's view of the situation is blurred by randomness. Given the heightened difficulty of deadlock, I will focus on its representation.

Definition 2.10. DEADLOCK

A paired directed graph exhibits **deadlock** if and only if it contains a cycle. □

Nested diposets can not exhibit deadlock as per the Acyclic Diposet Theorem (Theorem 2.3). What Definition 2.10 tells us is that a software system that can be modelled by a nested diposet can not exhibit deadlock. To determine if a system exhibits deadlock, apply the relavent order and containment relationships and attempt to construct a diposet model of the system. *If it is possible to apply order and containment relationships without violating the nested containment rule and arrive at a cycle, then deadlock can occur. In such instances the model is not a diposet but rather a paired directed graph that is not anti-symmetric. Otherwise, the model is a diposet and by definition it is deadlock free.*

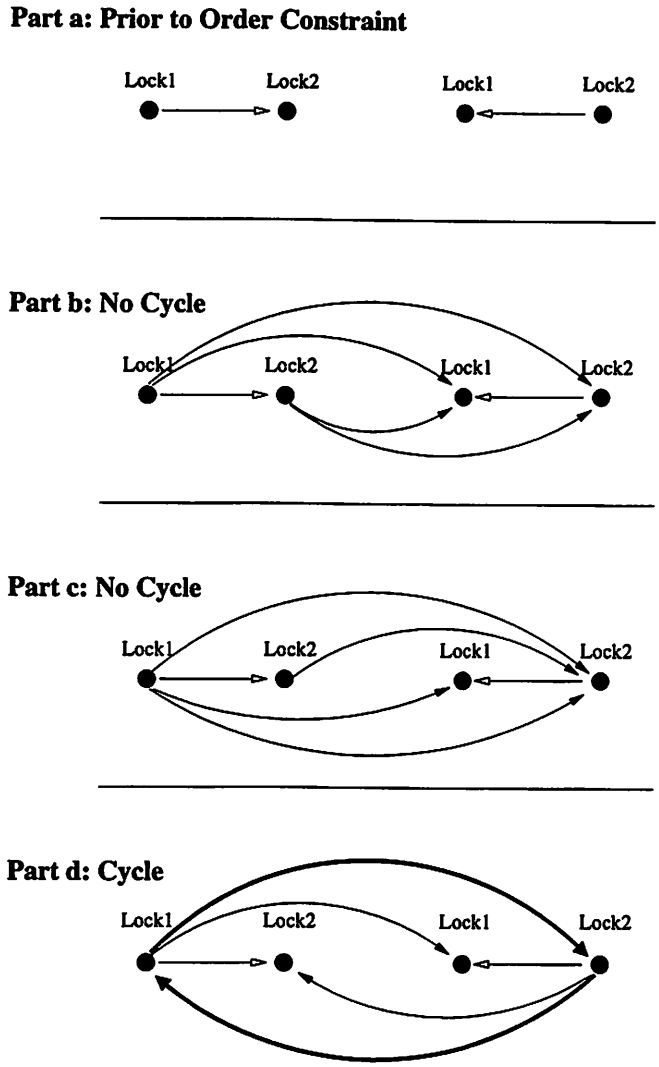

Figure 2.14. Order/Containment Constraints Leading To Deadlock

Deadlock often comes about through the use of multiple synchronization locks. As stated in the previous section on safety, synchronization locks that have a common label typically have an ordering constraint that requires that they be comparable. In conjunction with the order constraint on synchronization locks, deadlock-prone code often implements such locks so that they are contained by one another. This containment constraint can often contradict the ordering constraint and lead to deadlock. To illustrate this phenomenon see Figure 2.14. The first section (part a) of the figure shows a diposet consisting of two distinct threads each involving two events with the displayed labels. If

we treat these events as the holding of synchronization locks, then we know that an ordering relation must be applied between the separate threads so that the locks are not concurrent. The next three sections of Figure 2.14 show distinct application of order constraints to the two threads. In each case, the applied order constraints do not violate the nested containment rules. In part d of Figure 2.14 the thick lines indicate that a cycle exists - *deadlock!* Given the order constraints imposed by the synchronization locks, it is possible for this system to experience deadlock and in fact the model in part d of Figure 2.14 is not a diposet.

Figure 2.15 consists of an alternative configuration such that the containment constraint of the left thread is reversed. Again, order constraints are applied to the nested diposet, however, because of the reversed containment constraint, order constraints must be applied in a manner different from Figure 2.14. In no case can order constraints be applied without violating the nested containment rule and lead to cycles. Thus, the configuration of this software system is not deadlock-prone. Note that there are only two ways to apply order constraints without violating the nested containment rules in Figure 2.15.

In considering Figures 2.14 and 2.15 note how the order and containment constraints come about in concurrent programs. Containment constraints are typically determined at compile time. How the source code of a program is written determines what the containment constraints will be. Order constraints between threads are typically determined at run-time and are a function of relative thread speeds. This is why we show the containment constraints first followed by the order constraints.

2.2.3 Conservative Compile Time Deadlock Detection

One of the key advantages of diposets is their potential for compile time detection of deadlock. Detection of deadlock at *compile time* means that the determination of the possibility of deadlock in a system will occur prior to the execution of the system. Compile time detection takes place during a system's design process and thus offers the opportunity for correction of the problem by the system designers. Compile time deadlock determination is in contrast to determination of deadlock at run-time. *Run-time* deadlock detection occurs while a system is actually executing and being used. There are many approaches for detecting deadlock at run-time [Mattern, 1989; Chase and Garg, 1998; Lynch, 1996]. Unfortunately, detection of deadlock at run-time suffers from two prob-

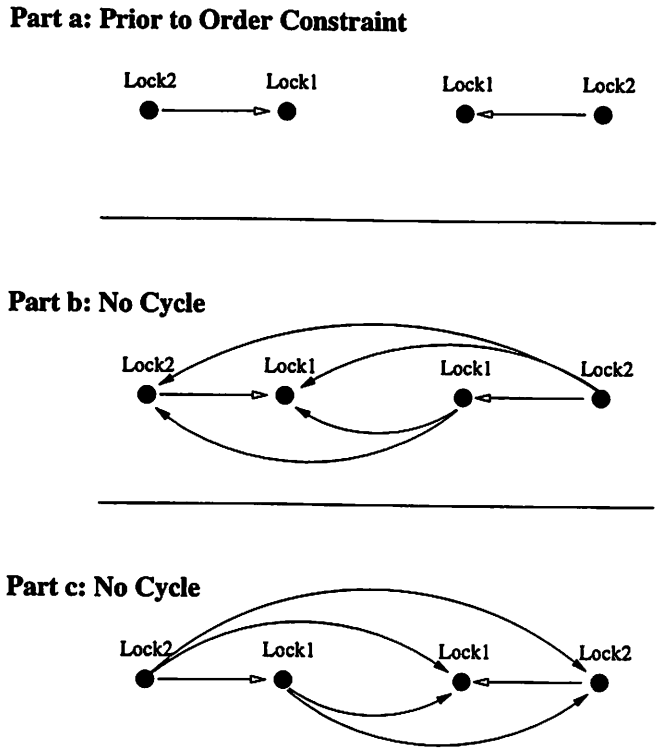

Figure 2.15. Order/Containment Constraints That Do Not Lead To Deadlock

lems. First, it is better to prevent deadlock before using a system than to simply detect deadlock during a system's operation. Second, a solution for deadlock while the system is being used is typically insufficient. This is especially so with the advent of widely available embedded systems. Many embedded systems are of a safety critical nature such as an embedded system controlling an automobile's antilock braking system. Obviously deadlock detection *during* the operation of an antilock braking system will place the lives of the automotive passengers in jeopardy.

Diposets offer the opportunity for conservative detection of deadlock at compile time. By *conservative* I mean that one can determine the *possibility* of deadlock, not the *certainty* of deadlock. Conservative deadlock detection determines whether deadlock *can* occur not whether deadlock *will* occur. This is a key distinction. In general software systems, i.e., software systems with infinite-valued variables, determining if deadlock will occur is undecidable. *Undecidability* stems from the fact that the model of computation of general software systems is Turing complete.[9] Determining if

[9] A Turing complete model of computation can implement a Turing machine. All of the process models of computation in this dissertation are Turing complete.

deadlock will occur for a Turing complete system would require checking a search space consisting of an infinite set of possibilities. In contrast, using a conservative deadlock detection mechanism requires the consideration of a finite set of possibilities.

The basic approach for using diposets to determine the possibility of deadlock at compile time is simple: *if and only if a software system can be represented by a diposet, then deadlock will not be possible.* If a software system can be represented by a diposet then that implies that the system does not contain cycles which further implies that deadlock is not possible. The general algorithm for this process is as follows.

1) **Create the System Specification**

   This step simply involves the system programmer(s) writing the software program.

2) **Automatically Recognize Order and Containment**

   Determining order and containment in the system specification can be automated by an appropriate tool.

3) **Store the Order and Containment in an Appropriate Data Structure**

   Storage of the order and containment relations will be similar to the storage techniques used for binary trees and other common data structures.

4) **Search for Cycles**

   The absence of cycles indicates that the structure is a diposet.

The difficulty with a realization of the above algorithm is that it will be extremely computationally complex. In fact, it will likely be NP-Complete (see Garey and Johnson [1979]). Nevertheless, there may be opportunities for developing heuristics that simplify the deadlock detection process considerably. Such heuristics are beyond the scope of this dissertation but will be part of any future work on this topic (see Chapter 5).

2.2.4 Communication Semantics

Communication between threads imposes an order constraint on their composite diposet. These order constraints are precisely the communication order relations discussed in Section 2.1.4. In this section I will discuss two important communication styles and describe their corresponding

communication order relationsh. As mentioned in Section 1.1.3, message passing is one of the fundamental ways to communicate within a concurrent system. Message passing communication assumes that components are connected via channels through which messages are transmitted. There are two types of message passing communication: synchronous and asynchronous. *Synchronous message passing* requires both the sender and receiver connected by a channel to be synchronized when a communication occurs. *Asynchronous message passing* does not require the sender and receiver to be simultaneously engaged and involves a storage facility in which messages can be placed by the sender until the receiver is ready.

The communication order relations for asynchronous message passing is very simple. Given that the sending event is denoted x and the receiving event is denoted y, an asynchronous message passing communication order relation for x and y is written

$$C_O(x, y) = \{(x, y)\}.$$

In other words, x must precede y. A graphical example of such a relation is shown in Figure 2.16. Here the left thread communicates to the right thread. Event x is the sending event and event y is the receiving event.

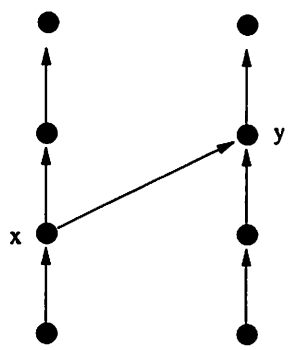

Figure 2.16. A Diposet Representing Asynchronous Communication

The communication order relation for synchronous message passing is significantly more complex than asynchronous message passing. Given that the sending event is denoted x and the receiving event is denoted y, a synchronous message passing communication order relation for x and y is equivalent to that given in Equation 2.1. For convenience I have rewritten Equation 2.1 below. Note that synchronous message passing is symmetric; i.e., $C_O(x, y) = C_O(y, x)$. Thus,

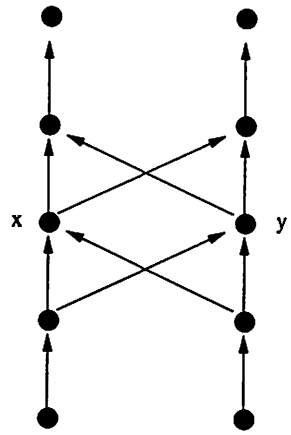

Figure 2.17. A Diposet Representing Synchronous Communication

there is no need to differentiate a sender and receiver. The graphical representation of synchronous message passing is shown in Figure 2.17 in which event x and y are synchronous.

$$C_O(x,y) = \begin{array}{l} \{(x,y')|y' \in y_{up-set_O}\} \cup \{(y',x)|y' \in y_{down-set_O}\} \\ \cup\{(x',y)|x' \in x_{down-set_O}\} \cup \{(y,x')|x' \in x_{up-set_O}\} \end{array}$$

2.2.5 An Example: PtPlot And The Java™Swing Package

I conclude this chapter with an informative and real world example. I will demonstrate how diposets can model the threading mechanism that is part of the Swing package of the Java™ programming language. The Java™Swing package consists of a set of graphical user interface (GUI) components that have a pluggable look and feel. The *pluggable look and feel* lets one design a single set of GUI components that can automatically have the look and feel of any OS platform (e.g., Microsoft Windows™, Sun Solaris™, Apple Macintosh™). As with all GUIs, the Swing graphical user interface must respond both to human input such as mouseclicks and text entry as well as computer input such as new image positions generated by a program or new windows to display. Responding to both computer and human input is an inherently concurrent process. Swing addresses this concurrency with a single event dispatch thread for all GUI operations.

The Swing *event dispatch thread* takes events (e.g., the pressing of a button or clicking of a mouse) and schedules them to occur in a sequential order. The `invokeAndWait()` and `invokeLater()` methods are available so that other threads in a program can access the event

dispatch thread (these methods are part of the `javax.swing.SwingUtilities` class). The `invokeAndWait()` method communicates synchronously with the event dispatching thread. The `invokeLater()` method communicates asynchronously with the event dispatching thread. Improper use of the `invokeAndWait()` or `invokeLater()` methods is a greater source of confusion among Swing users and can result in deadlock.[10]

PtPlot, created by Edward A. Lee and Christopher Hylands, is an example Java™ program that uses the Swing package [Davis *et al.*, 1999, chapter 10].[11] PtPlot consists of Java™ classes (many of which are Swing classes) that plot data on a graphical display. The main thread in the program is part of the `Plot` class `run()` method. This thread (I'll refer to it as the *PtPlot thread*) repeatedly calls the `Plot.addPoint()` method. `addPoint()` synchronizes on the `Plot` object lock and then attempts to draw points on the display. This latter task (drawing points on the display) requires the PtPlot thread to communicate with the Swing event dispatch thread. Separate from the Plot thread are several buttons for modifying the view of the PtPlot display. One such button is the *fill button*. If a user clicks on the fill button the `ButtonListener.actionPerformed()` method will be called and this in turn calls the `Plot.fillPlot()` method. The `Plot.fillPlot()` method is synchronized on the `Plot` object lock. Since the fill button is a swing component, `ButtonListener.actionPerformed()` and all of its contents are part of the event dispatch thread.

In order for the PtPlot thread to actually add points to the display, it must communicate with the event dispatch thread either through the `invokeLater()` method or the `invokeAndWait()` method. Diposets illustrate how the former method is deadlock free while the latter is deadlock prone. Figure 2.18 shows the two separate threads - the PtPlot thread and the event dispatch thread - without communication between them. Two order constraints must be added to this figure. The first constrains the two invocations of the `Plot` lock to not occur concurrently. The second constraint is due to the communication between the PtPlot thread and the event dispatch thread. This second constraint is a function of the `displayPoints` event and the event labelled "`communication event`." Figure 2.19 shows both constraints added to the two threads. The upper section of Figure 2.19 consists of the asynchronous constraint that is imposed by `invokeLater()`. The lower

[10] For a glimpse at the headaches faced by users of the two invoke methods, view `http://forum.java.sun.com/` and search on `invokeLater`.

[11] PtPlot is available at `http://ptolemy.eecs.berkeley.edu/java/ptplot`.

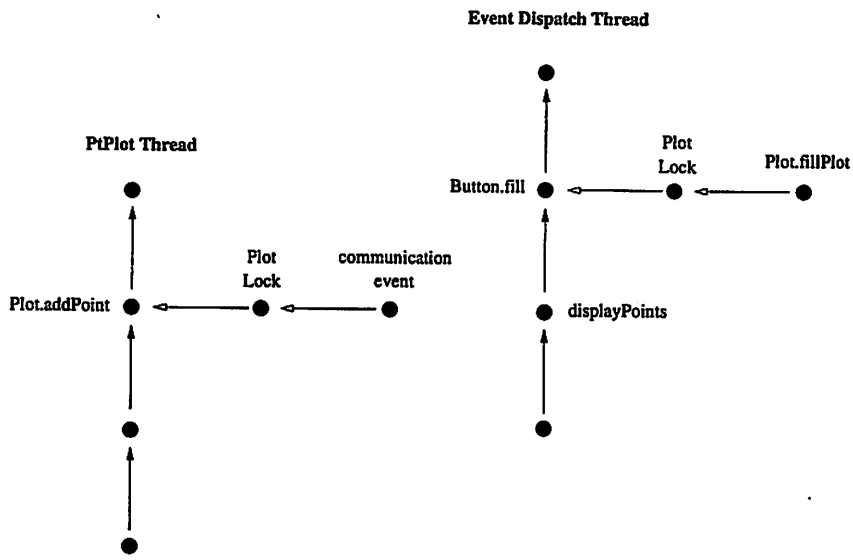

Figure 2.18. The Separate Threads In PtPlot

section of Figure 2.19 uses the synchronous constraint of `invokeAndWait()`. In this latter case a cycle exists.

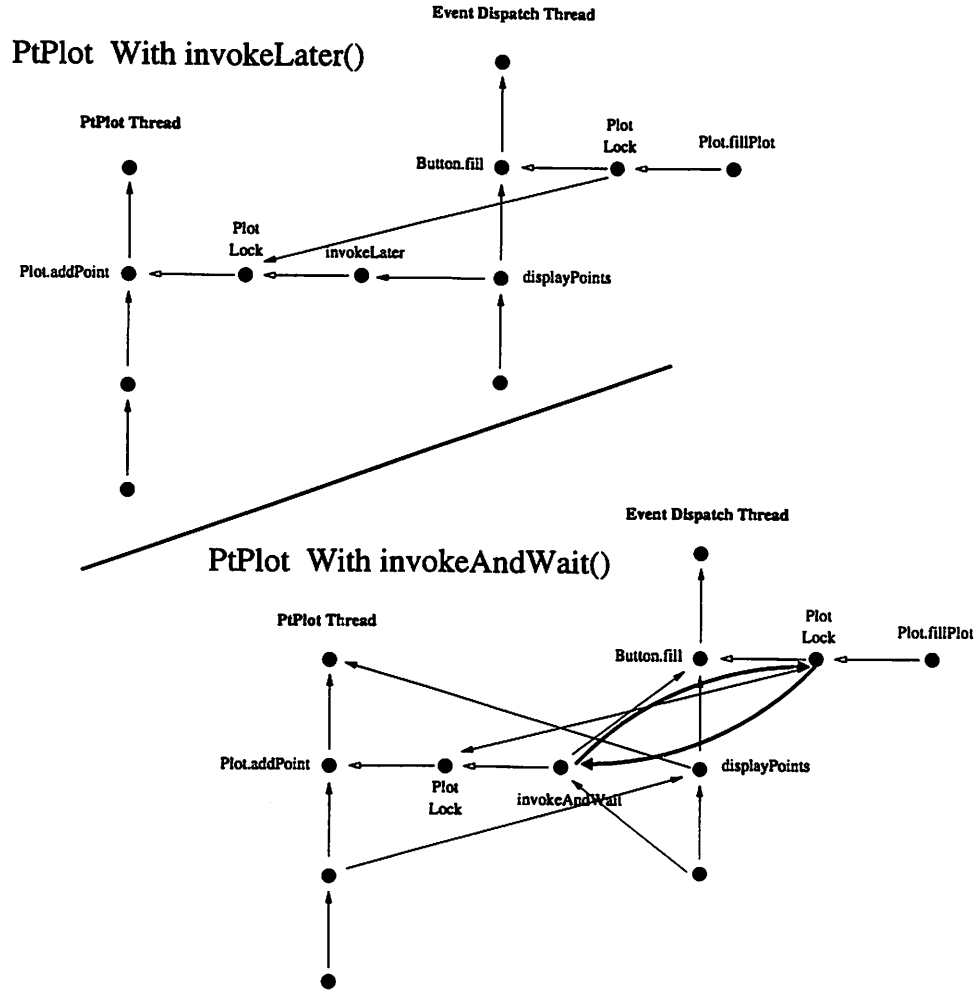

Figure 2.19. PtPlot and the Java™Swing Event Dispatch Thread

Chapter 3

Interfacing Heterogeneous Process Models

Can't we all get along?
- Rodney King, 1992 [1]

In Sections 1.1.3 and 2.2.4 several communication styles were introduced. In particular, synchronous and asynchronous message passing communication was described and diposets were used to represent both of these approaches. Synchronous and asynchronous message passing are two very important classes of concurrent communication, but the range of semantics options for describing communication and computation in a concurrent system goes well beyond these approaches. Any set of semantics serve to constrain and define the manner of communication and computation of a concurrent system. A set of such semantics describing how components in a concurrent system can communicate and compute data is referred to as a *model of computation.*

A model of computation (MoC) is a concept that traditionally has played a behind-the-scenes role in the design of computational systems. Often the constraints imposed by a model of computation fade into the background and only reside in the designer's subconscious. Nevertheless, *all* specification systems realize a particular model of computation. Von Neumann-style imperative programming languages applied to sequential software systems utilize an automata-based model of computation. Verilog and VHDL, two common hardware design languages, both use a discrete event

[1] This quote was made in response to a racial insurrection in Los Angeles spawned by a 1992 Simi Valley, California court verdict.

model of computation. Tools that implement digital signal processors often realize one of several possible dataflow MoCs [Girault *et al.*, 1999; Buck, 1994; Bhattacharyya and Lee, 1994].

A model of computation determines how a component communicates data and computes data. The method by which a component communicates data is realized by a *communication interface* or simply *interface*. A communication interface facilitates data transfer between a component and the other components to which it communicates. A component's communication interface is defined by the component's model of computation. For example, a discrete event (DE) component that keeps track of time must have a mechanism for specifying time stamps in its communication interface. A synchronous dataflow (SDF) component need not incorporate time into its interface as time is not a relavant parameter.

Given a particular model of computation, there are two approaches to executing a network of components. One approach is schedule-based. In the *schedule-based* approach, a schedule is created that specifies an ordering of invocations of each component contained in the network. In many cases the schedule is sequential, although this is not necessary. As each component is invoked, computation of data occurs. A schedule-based execution model presumes that each component's computation is finite. A second approach to execution of a network of components is process-based. *Process-based* execution of a network of components does not assume that each component's execution is finite. For the sake of fairness in the face of possibly infinite computation, the process-based method assigns an autonomous thread of control to each component. Due to the autonomy of each component afforded by the assigned thread, components in a process-based execution are often referred to as *processes*.

For most models of computation, a network of components can be executed in either a schedule-based or process-based manner. Certain models of computation are more amenable to one style or the other. For example, the Synchronous Dataflow (SDF) model of computation [Lee and Messerschmitt, 1987] is best executed according to a schedule-based execution model. This is because it is relatively easy to determine efficient sequential schedules for SDF networks. On the other hand, the distributed discrete event (DDE) model of computation is best executed in a process-based manner since its distributed nature is especially amenable to separate threads of control [Righter and Walrand, 1989].

Models of computation facilitate well defined specification of concurrent systems, but for

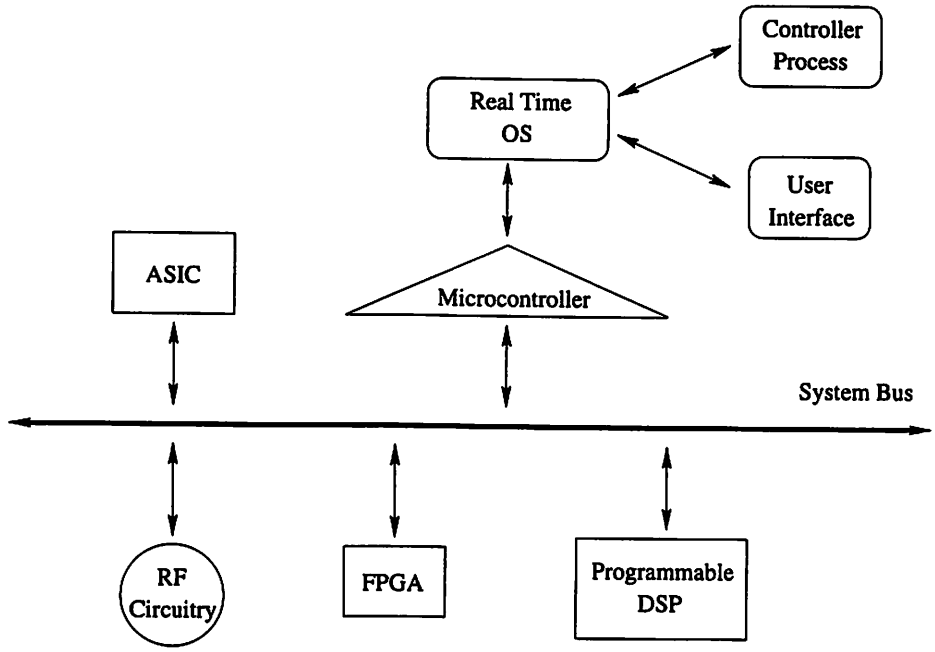

Figure 3.1. A Sample Embedded System

most large, complex systems a single model of computation can not be used alone. Complex systems typically consist of several subsections with different sections best described by different MoCs. As an example, consider an embedded system as displayed in Figure 3.1. This system, with characteristics of cell phones and personal digital assistants, is not easily described by a single MoC. The analog RF front end is best described by a model of computation that uses differential equations. The control-oriented aspects of the embedded system are best described by a discrete event model of computation. The graphical user-interface is suitable for description by a process-oriented model of computation that can easily describe the non-deterministic interface. The voice coder DSP is best described by a dataflow model of computation such as synchronous dataflow.

Heterogeneous application of models of computation is an approach that recognizes the need for multiple MoCs to be used in conjunction with one another for describing complex systems. As discussed in Section 1.1.4, many researchers in the System Level EDA community propose a heterogeneous approach for dealing with complex system design. Given the use of a heterogeneous approach, it becomes implicit that components of different models of computation communicate with one another. I.e., heterogeneity implies that components with different communication interfaces

must communicate. The question becomes *how?* How should heterogeneous components communicate with one another. In general, there are two approaches for handling the interaction of heterogeneous MoCs. The *amorphous* approach to heterogeneity allows components of different MoCs to communicate directly. [2] The *structured* approach to heterogeneity requires that components of different MoCs communicate through an adapter.

The choice of amorphous versus structured heterogeneity has implications on both the communication and computation of a network of components. From the perspective of communication, amorphous heterogeneity implies that a single component must incorporate features of multiple MoCs and, hence, have multiple interfaces. For example, a single component might be required to support both asynchronous message passing and synchronous message passing. A similar phenomenon exists in the realm of computation. Suppose a given component communicates with some components that observe the synchrony hypothesis and others that do not. Should the component in question observe synchrony or not? In effect amorphous heterogeneity burdens each component with the possibility of having to deal with every available model of computation - a burden that renders the model of computation concept useless.

Structured heterogeneity enforces the application of a single model of computation to any single component in a network by using adapters to connect incompatible interfaces. An *adapter* converts the interface of one component into the interface of another. Adapters (also called *wrappers*) play the role of interface translators. A treatment of adapters as object-oriented patterns can be found in Gamma *et al.* [1995]. Adapters are advantageous for several reasons. First, adapters can serve as boundaries for separating computation in addition to separating communication. Using adapters to separate computation can be helpful in managing shared processor resources. Second, an adapter simplifies the job of the designer. A designer with a given expertise (e.g., familiarity with a particular set of communication semantics) can focus on the semantics that he or she is familiar with. The disadvantage of structured heterogeneity is that semantics must be determined for the adapters themselves.

A special class of structured heterogeneity is *hierarchical heterogeneity*. While structured heterogeneity requires that components communicate across MoC boundaries through an adapter, hierarchical heterogeneity adds the has-a relationship to components within a network. To under-

[2] The term *amorphous heterogeneity* is due to Edward A. Lee.

stand hierarchical heterogeneity, consider two components, A and B, that communicate directly to each other without the use of an adapter (i.e., by structured heterogeneity we recognize that they must have compatible communication interfaces and execute according to the same model of computation). If there exists a third component, C, that both A and B communicate directly to, then hierarchical heterogeneity requires that neither A nor B use an adapter to communicate to C or both A and B use an adapter such that the respective adapters serve as boundaries to the same pair of MoCs.

Hierarchical heterogeneity has many advantages. From a syntactic point of view hierarchical heterogeneity allows a network of components to be abstracted into a single component. A single abstracted component can contain another network of components with that network executing according to a different model of computation. Such abstraction allows a designer to view a system at the level of detail desired. Semantically hierarchical heterogeneity can be used to organize heterogeneity in a telescoped fashion that facilitates successive refinement. Milner [1989] suggests that computation can be successively refined into layers of communication[3] (this is also dealt with in Rowson and Sangiovanni-Vincentelli [1997]). Using hierarchical heterogeneity we can continually peer deeper into a component to reveal new networks of communication. Components in a hierarchical system that contain other components are referred to as *composite components*. Components in a hierarchical system that do not contain other components are called *atomic components*.

Hierarchical heterogeneity has a very practical basis that is becoming increasingly relevant from an industrial standpoint. Based on industry trends it is rare for a single company to design a complete system including all subcomponents. Instead, certain firms specialize in subsystems and sell the designs - the intellectual property or *IP* - to other firms that manufacture the complete system [Dalpasso *et al.*, 1999]. Components based on different IP will often have incompatible interfaces [Rowson and Sangiovanni-Vincentelli, 1997; Passerone *et al.*, 1998]. Furthermore, based on time to market constraints and the desire to seek the lowest possible costs, it is common to swap similar IP throughout the design process. The black box perspective that hierarchical heterogeneity affords is very amenable to the "part swapping" of IP.

Defining the semantics of adapters between hierarchical, heterogeneous components is the central question of this chapter. I will consider a solution to this problem in the context of process-based models of computation. Attacking the adapter between processes of different MoCs is ar-

[3] See the beginning of Chapter 1 in Milner [1989] for this discussion.

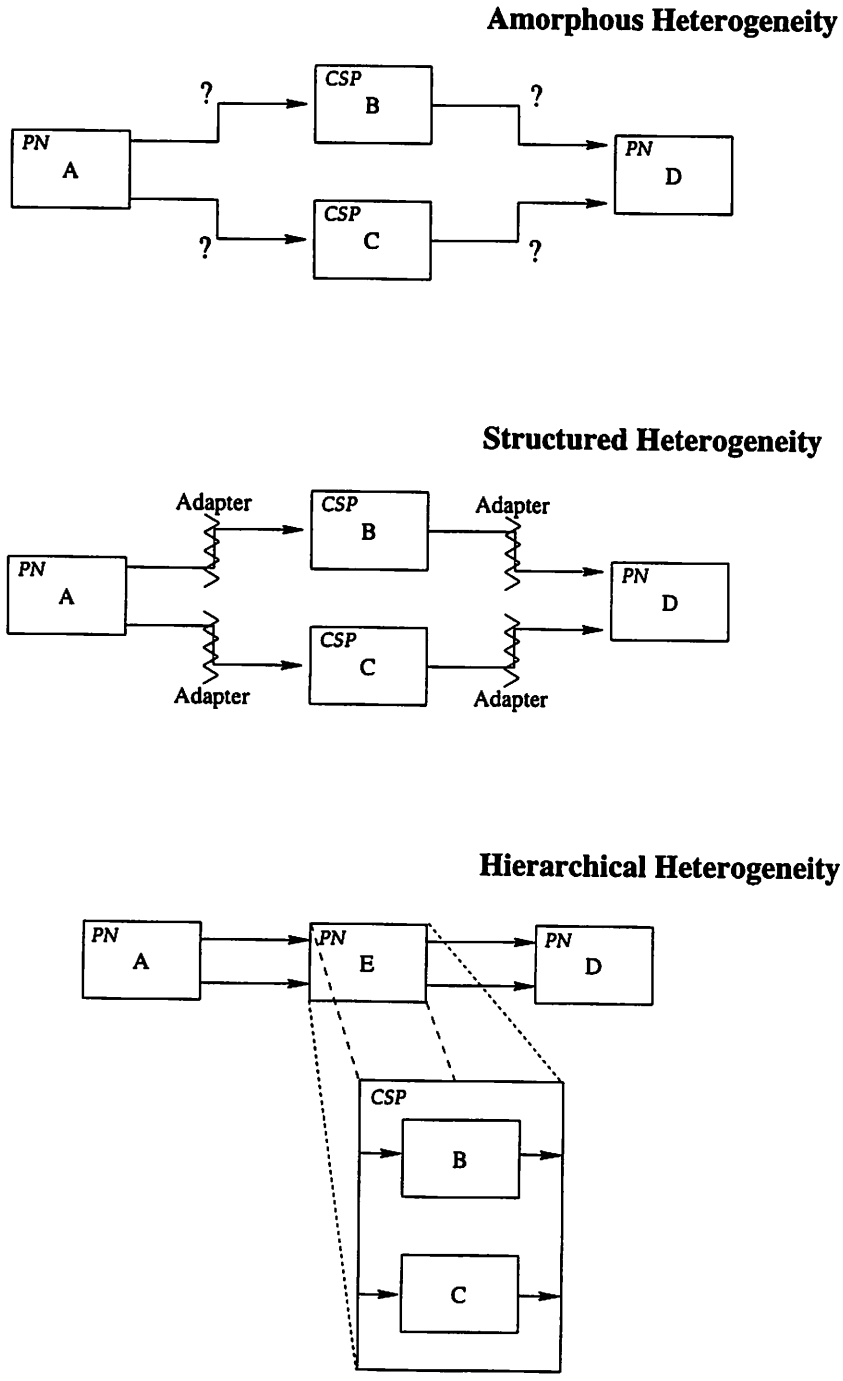

Figure 3.2. Types of Heterogeneity

guably more challenging than the equivalent problem for schedule-based components. The difficulty is analogous to the difference between sequential versus concurrent systems; both systems are challenging but as outlined in Chapter 1, concurrent systems are more difficult.

The remainder of this chapter proceeds as follows. In Section 3.1 I consider criteria against which to measure how effective a given adapter solution is. In Section 3.2 I review the semantics of three process-oriented models of computation that serve as case studies. In Sections 3.3 and 3.4 I propose a solution to the problem of interfacing heterogeneous process-oriented models of computation.

3.1 Assessing The Effectiveness of an Adapter

The goal of an adapter is to translate the communication semantics between the interfaces of heterogeneous components and to disaggregate execution. In order to clarify this goal I consider desired characteristics of interfaces below. These characteristics will serve as a gauge for comparing various adapter alternatives.

Simplicity

We would like adapters to be simple. An overly complex solution would equate an adapter with a component whose sole purpose is to translate communication semantics. The primary problem with making an adapter a component is that this adds an additional execution burden to the original network of components. Instead of simply executing a set of connected components, there must also be execution of the adapter components between them. Another problem with assigning the task of an adapter to a component is that this solution sits on a slippery slope above amorphous heterogeneity. A better option is to design adapters with sufficient simplicity so that they do not perform any computation of data.

Generality

Closely related to the desire for a simple adapter is the desire for an adapter that can be generally applied to a broad set of MoC pairs. The desire for generality is an attempt to avoid the N^2 problem.

Recall that an adapter always occurs between a pair of models of computation. [4] We certainly do not want to have to define a unique adapter between every possible pairing of MoC interfaces; given N models of computation, such an approach would require N^2 adapters. Instead, we would like to design a single adapter that operates properly between any pair of MoCs. Such generality will be advantageous from a software engineering perspective.

Avoidance of Deadlock

We do not want an adapter to introduce the possibility of deadlock. To make this issue clear, I introduce the concept of homosemantic abstraction. *Homosemantic abstraction* is the realization of hierarchy without heterogeneity. It occurs when two components executing according to the same model of computation communicate with one another through an adapter. Homosemantic abstraction facilitates separation of execution even though the components all have compatible communication interfaces. Clearly, a network of components that incorporate homosemantic abstraction should be semantically identical to the same network of components in which homosemantic adapters have been removed. I apply this same reasoning to deadlock. If homosemantic adapters are introduced to a network of components, the network of components should be no more deadlock-prone than prior to the addition of the adapters.

Determinacy

Many models of computation guarantee deterministic execution of a network of components. The determinacy is generally a result of the MoC's denotational semantics; given that the components themselves do not randomly compute data, then execution of the components will result in a deterministic outcome *even if the components are invoked according to a non-deterministic schedule*. Examples of models of computation with guarantees of determinacy in the manner cited above include all dataflow models (e.g., Process Networks, Dynamic Dataflow, Boolean Dataflow and Synchronous Dataflow [Lee and Parks, 1995]) as well as discrete event models [Yates, 1993; Lee, 1999b]. I will apply homosemantic abstraction in a manner identical to my previous use with deadlock: given a network of determinate components, the network of components should maintain determinacy even

[4] Note the tacit constraint that adapters occur between exactly two components. While it is possible to have three (or more) way connections, it is rare and hence I am not considering those cases.

with the addition of homosemantic adapters.

3.2 Process Models

As stated, I am considering the issue of heterogeneous semantics with emphasis placed on the interaction between process models of computation. As a case study, I will consider three particular process models of computation and for completeness I summarize these three models of computation below.

3.2.1 Distributed Discrete Event (DDE)

The distributed discrete event (DDE) model of computation uses asynchronous message passing in which the messages passed are time-stamped events. Each component maintains a local notion of time and components communicate their local notion of time by producing and passing time stamped events. When a component receives an event it advances its local notion of time to that of the received event. By virtue of a component's local clock, all events consumed or produced by a particular component are totally ordered. Events associated with distinct components are partially ordered. Herein lies the distinction between distributed discrete event systems and traditional discrete event systems. In traditional DE systems the set of *all* system events are totally ordered, not just those associated with a single component. Hence, components in a traditional discrete event system must be invoked sequentially while distributed discrete event modeling leverages the natural concurrency existing in a network based on the networks's topology. Distributed discrete event modeling and discrete event modeling have been studied extensively in Chandy and Misra [1981]; Righter and Walrand [1989]; Morgan [1985]; Lamport [1978]; Jefferson [1985].

3.2.2 Process Networks (PN)

Gilles Kahn [Kahn, 1974; Kahn and MacQueen, 1977] developed Process Networks (PN) as a way to take advantage of Dana Scott's work in denotational semantics and apply it to concurrent systems. Components in a process networks model communicate via asynchronous message passing without a notion of time. Communication occurs through blocking reads of FIFO queues. If the queues have bounded memory, then writing to a queue when it is full becomes a blocking write.

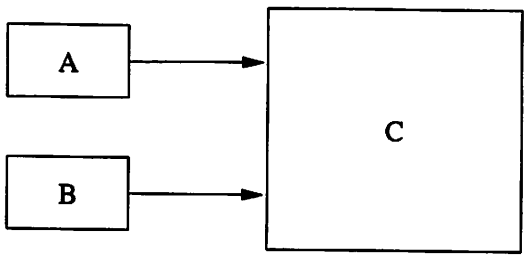

Figure 3.3. Non-deterministic Choice

Each component effectively maps an input stream to an output stream. The set of all streams in the network of components form a complete partial order based on prefix ordering. Based on this CPO of streams, the denotational semantics of process networks guarantees determinacy. By determinacy it is meant that neither relative computation speed nor ordering of invocation of the components in a PN network will impact the outcome of data streams.

3.2.3 Communicating Sequential Processes (CSP)

Communicating Sequential Processes (CSP) is a modeling system developed by Tony Hoare [Hoare, 1985]. Processes in CSP communicate via synchronous message passing without a notion of time. In addition to synchronous message passing, processes in CSP may use non-deterministic choice. Non-deterministic choice allows a single component to consider several possible communication options and then randomly select a single option among the set of choices that are enabled. Consider Figure 3.3 to understand the meaning of non-deterministic choice. In the block diagram, component C can non-deterministically choose input from either the upper or lower channel. C must then wait for communication on either of the channels to be enabled which occurs when either component A or B is ready to communicate to C. Component C completes communication with the first channel that is enabled. If both channels are enabled simultaneously, than component C randomly chooses one of the channels to communicate with. Non-deterministic choice may seem odd, but it is a facility that has parallels in several different modeling languages. The inherent randomness of non-deterministic choice is useful in allowing a designer to partially specify a system. Closely related to CSP is Communicating Concurrent Systems (CCS) developed by Robin Milner [Milner, 1989]. The semantics of CSP and CCS are virtually identical.

3.3 Order & Atomic Processes

It is worth comparing and contrasting the three process models of computation presented thus far, and to do so I refer to a relevant quote:

> A concurrent system is a network of communicating sequential processes.
> *Robin Milner, 1989*

I refer to this quote to draw attention to the word *sequential.* The context of Milner's quote was directed at his communicating concurrent systems (CCS) modeling language, but many other modeling languages assume that the basic computational element is sequential. Certainly all of the modeling frameworks mentioned in this dissertation assume a sequential primitive, including communicating sequential processes, process networks, the Actor's model and many others. Modeling languages that incorporate the synchrony assumption in conjunction with a state transition also implicitly assume a sequential primitive. For example, in the Reactive Modules modeling language [Alur and Henzinger, 1996], the existence of an atomic round during which all components simultaneously change state permits one to extensionally view the state change as occurring sequentially.

Sequential execution implies a total ordering on all operations of a component. In other words, a component's operations can be represented by a thread. From an external point of view, the operations of concern are a component's communication operations. In a message passing system, communication can be either the writing of data messages to a channel (*production*) or the reading of data messages (*consumption*) from a channel. Sequential execution of a message passing component means that all consumptions and productions of a component are totally ordered. A model of computation's semantics determine exactly how such total ordering is realized.

3.3.1 Ordering Communication: Event Driven vs. Data Driven

An important classification of how a model of computation impacts the ordering of a component's communication actions is whether the components are event driven or data driven. *Event driven* models of computation are common in graphical user interfaces (GUI), reactive embedded systems and control systems. *Data driven* models of computation are often used to model the dataflow found in computer architectures as well as data intensive parallel processing schemes such as image processing.

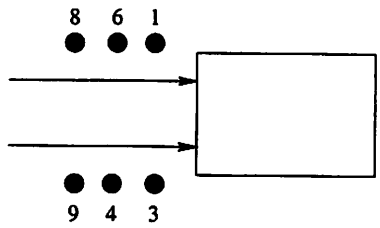

Figure 3.4. Time-Stamped Events Awaiting Consumption by a DDE Component

In event driven models, the ordering of a component's communication actions are determined by the external environment. Event driven models of computation, in which DDE is a special case, have externally determined consumptions. Given a DDE component with multiple input channels, it is not possible to determine *a priori* in what order consumptions of data messages will occur. A DDE component with two input channels, 1 and 2, can not specify that consumption will occur first on channel 1 followed by consumption on channel 2. Instead, the order of consumptions for event driven components is imposed by the environment.

The ordering of incoming time stamped events determines the order a DDE component consumes such data. Consider Figure 3.4 showing a DDE component with pending events (indicated by dots) destined for both input channels. Each number adjacent to an event indicates that event's time stamp. The time stamps shown indicate that the component must consume the messages as specified by the time stamp ordering. A DDE component *can* specify the relative ordering of event productions. Often such production is specified in response to a consumption. I.e., given a consumption on a particular input channel, produce an event on a particular output channel.

In data driven models of computation, a component autonomously makes the decision of whether it will consume or produce a message on any of its input channels. The absence of a message may force a component to wait, as in the case of an attempt to consume a message from an empty channel, but the relative ordering will be completely determined by the component. Hence, a component that decides to consume a message first from channel 1 followed by channel 2, may have to wait (perhaps indefinitely) on channel 1 but the decision to consume from channel 1 *before* consuming from channel 2 will be upheld independent of data availability.

PN is an example of a data driven model of computation and hence, the ordering of consumption and production actions are internally imposed. CSP components without the notion of non-

deterministic choice are also examples of consumption/production ordering due to internal criteria. The non-deterministic choice facility allows a CSP component to specify a set of alternative ordering constraints and then defer to a selection within the set based on external criteria. In effect, non-deterministic choice allows a component to be event driven with respect to both consumption and production.

3.3.2 Reordering Communication

Models of computation in which components internally determine the ordering of communication actions can be further classified based on how the communication actions can be reordered. In Chapter 2 we considered the impact of ordering on such undesirable properties as deadlock. For internally motivated models of computation we would like to characterize the sensitivity to reordering of communication actions. *Reordering* is defined as the act of switching the order of operations within a single thread. Reordering impacts only the order relation and does not impact the containment relation. For example, if two operations are mutually non-inclusive, they will remain so after reordering.

For a given model of computation, reordering may or may not impact a component's interaction with other components. When a reordering does not impact the safety or liveness of a set of components, I say that the model of computation is *reorder invariant* with respect to a set of actions; otherwise the MoC is *reorder variant* with respect to a set of actions. Whether or not reordering will impact liveness or safety will have a profound impact on both the flexibility of component execution as well as how the hierarchical composition of components should be organized. To evaluate the impact of reordering on communication actions for a given model of computation, let us recall the fundamental ordering constraints of communication within both PN and CSP.

Figure 3.5 shows the fundamental ordering constraint realized in process networks with unbounded channels. The consumption of a data message through a channel and from a production simply requires that the consumption (action g) occur after the production (action b). This constraint, characteristic of asynchronous message passing schemes, is shown in the figure with two threads that communicate via a single consumption/production pair. Process networks with bounded channels require an additional ordering constraint. A network with channels that can store N unread messages requires that at least one consumption of data must occur for every $N + 1$ productions. Figure 3.6 illustrates this constraint with a channel that can store two unconsumed data messsages. Actions a, b

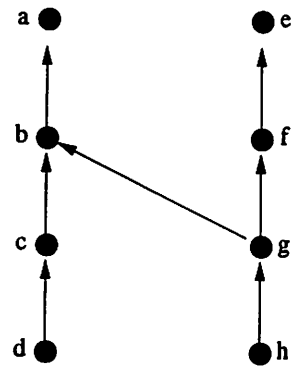

Figure 3.5. The Basic Unbounded Asynchronous Message Passing Order Constraint

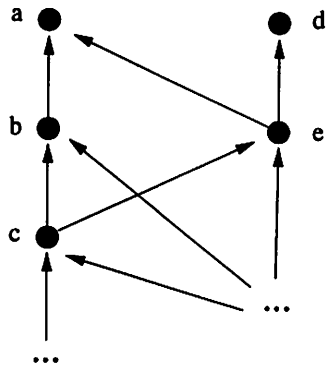

Figure 3.6. The Basic Bounded Asynchronous Message Passing Order Constraint

and *c* are productions by the left thread and actions *e* is a corresponding consumption. The constraint that action *c* must occur after action *e* indicates that the production associated with action *c* can not occur until after action *e* enables sufficient capacity in the channel.

The synchronous message passing feature of CSP places a much tighter ordering constraint on a set of communicating threads than does asynchronous message passing. Even in the case of bounded asynchronous message passing with a channel capacity for one data message, the ordering constraint impacts only three actions in a communication between two threads. Given two threads that communicate via synchronous message passing, an ordering constraint will be imposed on a total of six actions. This is illustrated in Figure 3.7 in which action *b* and *e* are synchronous. Note in particular that the synchronous ordering constraint impacts the predecessor and successor of both *b* and *e*. The choice operator of CSP is not illustrated, but recall that it implements synchronous

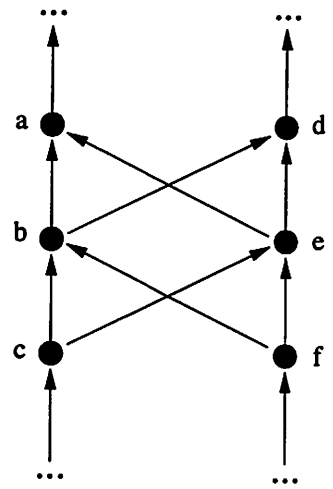

Figure 3.7. The Basic Synchronous Message Passing Order Constraint

message passsing with the allowance for multiple alternatives to be considered.

Unbounded process networks are reorder invariant with respect to a set of consumptions. As an example, consider Figure 3.8. Any two consumption actions of a thread can be rearranged without introducing a cycle in the set of communicating threads. The same can be said for the re-ordering of a set of productions within unbounded process networks. The sketch of the proof for the previous two declarations is virtually identical. First consider the case of two adjacent consumptions (productions) with no intervening actions. Clearly these can rearranged regardless of whether the consumptions (productions) communicate to the same or different threads. Subsequent application of reordering of adjacent consumptions (productions) facilitates the reordering of a set of consumptions (productions).

In general a thread within a process network is *not* reorder invariant with respect to a set of consumptions and productions. As an example, consider Figure 3.9. Bounded process networks have sufficient ordering constraints that components are reorder variant for any set of communication actions. The ordering constraints of synchronous message passing renders CSP components reorder variant. Recall that a synchronous message between two communication actions a and b imposes constraints on each other's respective successors and predecessors of a and b. Hence, if a thread reorders a synchronous communication action, this will lead to new successors and predecessors that can cause cyclic deadlock.

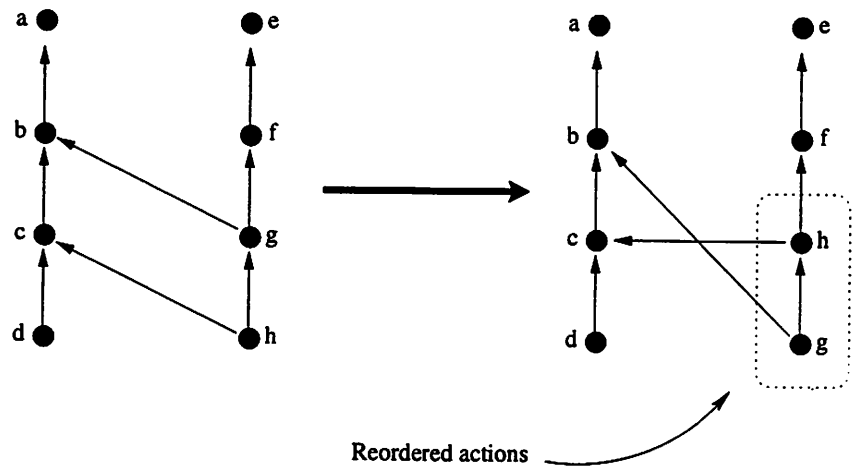

Figure 3.8. Reorder Invariance of Unbounded PN Consumptions

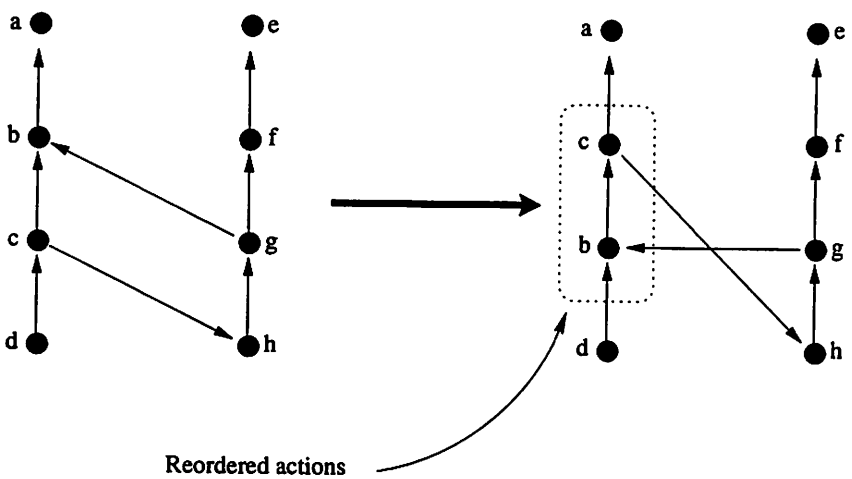

Figure 3.9. Reorder Variance of PN Consumptions with Productions

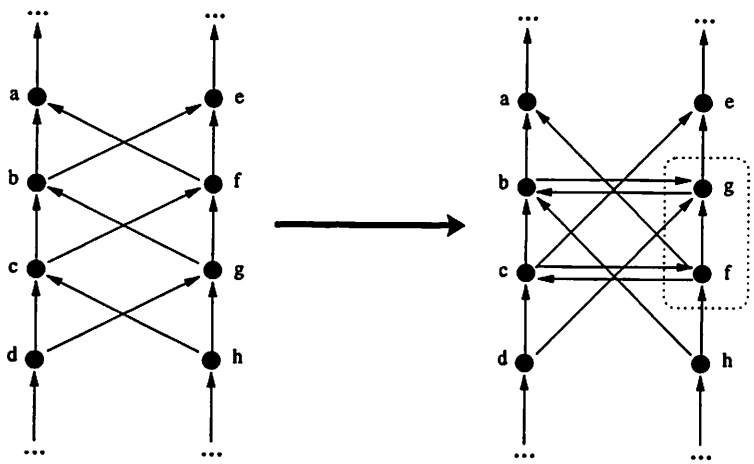

Figure 3.10. Reorder Variance of CSP Components

3.4 Order & Composite Processes

The question of creating an adapter between different models of computation in a hierarchical, heterogeneous network is really a question of determining how a composite component in such a network should execute. Externally a composite component executes according to the model of computation shared by its external neighbors. Internally a composite component contains a set of components that operate according to a model of computation that is generally different from that outside of the composite component. Between the external and internal worlds is an adapter that translates between the two models of computation. If we apply Milner's quote cited in the beginning of Section 3.3, we should execute the adapter of a composite component sequentially. Unfortunately, sequential execution of composite components is generally not possible if the MoCs involved are process models of computation.

The primary problem with sequential execution of the adapter of a composite component is that sequential execution introduces deadlock. As an example, consider Figure 3.11 in which components A, B, C and D are atomic with A and B contained by composite actor E and C and D contained by composite actor F. Assume that A and B perform no actions (produce nor consume any data messages) but C produces an infinite stream of messages that are consumed by D. If we execute composite actor F by performing a blocking read on the top input channel, then F will stall indefinitely. By imposing an order on F's execution we have no way of knowing a priori if our order

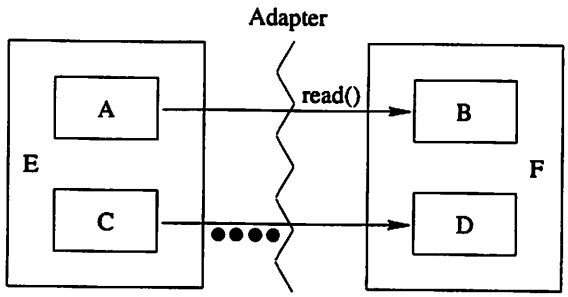

Figure 3.11. Sequential Execution of an Adapter

of execution will result in this kind of stalling. The more general problem with a sequential adapter in a process composite component is that a sequential adapter imposes a total ordering on a set of processes that are partially ordered. In general, this can cause cycles.

Since sequential adapters are deadlock-prone, consider a concurrent adapter instead. A concurrent adapter associates a thread with each channel flowing through the adapter. Each adapter thread waits on data and then passes the data through the channel. An adapter thread talks to components on either side of the adapter according to the prevailing model of computation. Consider a synchronous message passing component, component A, that produces messages that are transfered through an adapter to an asynchronous message passing component, component B. The adapter thread associated with the channel will wait on a synchronous put from A and then do an asynchronous put into B.

3.4.1 Concurrent Adapters

Concurrent adapters are useful for several reasons. First concurrent adapters essentially make the adapter an identity function from the prespective of data transfer. Thus, it is trivial to show that they maintain determinacy in the face of homosemantics abstraction. Second, concurrent adapters can be generally applied to a variety of MoC pairs; the association of a thread to each channel does not change as a function of the MoC. Third, concurrent adapters are conceptually simple. All channels are treated identically. Unfortunately, difficulty still lies ahead. The challenge in concurrent adapters is not solved simply by associating a separate thread to each channel. The difficulty is in determining how the threads communicate with their respective channels and when that

communication occurs. In Sections 3.4.2 and 3.4.3 I consider the significance of the communication semantics of the threads in a concurrent adapter with respect to the process models of computation I have previously introduced.

3.4.2 Non-Deterministic Choice: Blessing & Curse

Non-deterministic choice has been mentioned as a communication style that is part of CSP and is common in many other models of computation. The blessing of non-deterministic choice is that it allows a component to choose between a set of communication alternatives. If any of the choices in the set are valid than communication will be completed. In effect, non-deterministic choice allows a component to "increase its odds" for avoiding a deadlocked situation. Paradoxically, in the context of a composite component's adapter, non-deterministic choice can introduce deadlock conditions.

A very simple illustration can show the problems with non-deterministic choice. Consider an atomic component, A, with two output channels that communicate through an adapter to a single two input atomic component, B, as shown in Figure 3.12. Assume homosemantic abstraction with both the inside and outside MoCs being CSP. If A attempts non-deterministic choice through its two output channels, what will happen? Since the goal of non-deterministic choice is to randomly select an enabled communication channel, then if A views both the upper and lower channels as being valid simply by virtue of their respective adapter threads, then either channel can be selected. Let us suppose further that B is performing a blocking read on the upper channel. Clearly, if A randomly selects the lower channel then execution for the entire system will stall. Such a deadlock is inconsistent with the corresponding topology involving only the two atomic components and no composite components; homosemantic abstraction has introduced deadlock.

To avoid the above scenario with non-deterministic choice, we must define a channel as being enabled not simply based on the existence of an adapter thread. I propose that a channel be defined as enabled only after the possibility of a completed execution has been guaranteed. In other words, an adapter thread should transfer data in an atomic fashion. Applying an atomic transfer mechanism to the above scenario would work as follows. A would check for validity of each output channel. The corresponding adapter threads would not accept a message from A until they had verified that communication on the inside of the adapter would complete. Only the upper adapter thread

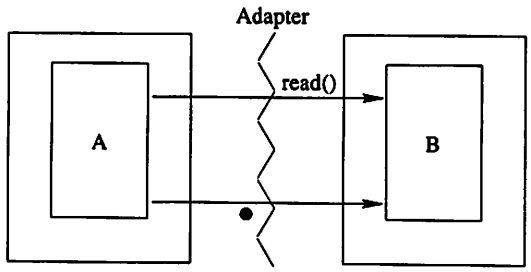

Figure 3.12. The Introduction of Deadlock Via Non-Deterministic Choice

would be validated since this thread could check for the blocking read on the upper channel. Hence, the non-deterministic choice semantic of A would choose the upper channel.

The need for an atomic transfer mechanism is fundamentally related to reorder invariance. A's selection of a valid output channel is equivalent to reordering the consumptions of component B. Since component B is executing according to the CSP model of computation and therefore is not reorder invariant, a non-atomic adapter transfer mechanism leads to deadlock. The same result can occur with bounded process networks. A non-atomic adapter transfer mechanism is not a problem if non-deterministic choice interacts with a set of unbounded PN components, since unbounded PN consists of components that are reorder invariant.

3.4.3 Totally Ordered Event Driven Models

In the previous section we determined that non-atomically transferring data across an adapter could lead to deadlock if non-deterministic choice interacts with components that are reorder variant. Unfortunately an atomic transfer mechanism can cause problems if a reorder variant model interacts with event driven models in which events are totally ordered. Consider an event driven component in which events (from mouse or keyboard activities perhaps) are totally ordered. If the totally ordered events are being transferred across an adapter to interact with a set of reorder variant components, then an atomic transfer mechanism can deadlock. A slight variation of Figure 3.12 can illustrate this as shown in Figure 3.13. If an atomic transfer mechanism exists in the topology shown in Figure 3.13, deadlock will result.

Although there is a paradoxical twist to the difference between the scenarios in Figure 3.12 and Figure 3.13, a simple explanation is available. In the former case, non-deterministic choice re-

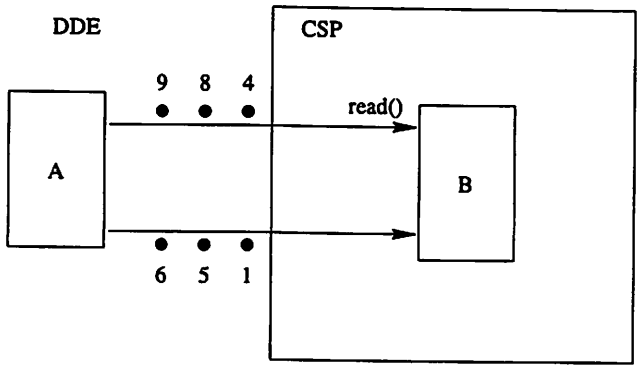

Figure 3.13. Totally Ordered Event Driven Models with Reorder Variant Components

sults in an event driven effect that is not totally ordered. In fact, because of the atomic transfer mechanism, the ordering of communication actions in Figure 3.12 is driven by component B. In Figure 3.13 the time stamps impose a total ordering. This total ordering is due solely to component A and has nothing to do with component B. The remedy is to allow asynchronous message passing across adapters between totally ordered event driven components and reorder variant components. E.g., a non-atomic adapter transfer mechanism.

Chapter 4

Implementation

It is better to practice it than to know how to define it.
- Thomas à Kempis[1]

This chapter serves as a practical illustration of the preceding sections of this dissertation. It includes a discussion of my solution to the problem of interfacing heterogeneous models of computation that was discussed in Chapter 3. I also show the practical implications of reorder invariance and describe my architecture for facilitating heterogeneity and hierarchy of process-oriented models of computation. In addition to these contributions, this chapter describes in detail a large scale system level design environment that served as the framework within which the implementations discussed in this chapter occurred. The large scale system level design environment that I am referring to is called the Ptolemy Project. Under the leadership of principal investigator Edward A. Lee, *The Ptolemy Project* is a software development project that studies the modeling and design of computational systems.

This chapter proceeds as follows. In Sections 4.1 and 4.2, I provide an overview of the general Ptolemy Project excluding process-oriented models of computation. In Section 4.3, I describe the architecture I created as a solution to heterogeneous, hierarchical interaction of process-oriented models of computation. In Section 4.4, I describe the impact of reorder invariance on domain polymorphism within Ptolemy.

[1] François Fénelon, *Christian Perfection* (New York: Harper & Brothers, 1947), p. 194.

4.1 Modeling & Design

The Ptolemy Project [2] studies the modeling and design of complex computational systems. Example computational systems considered in the Ptolemy Project include pagers, cell phones, security systems and computational subsystems found in automobiles (e.g., air bag systems). *Ptolemy II* is the latest software environment to be released by the Ptolemy Project. Ptolemy II facilitates the modeling and design of the kinds of systems listed above. By *modeling* we mean the act of representing a system or subsystem formally. By *design* we mean the act of defining a system or subsystem. Models and designs are complementary. In some cases a system model might serve as a constraint to which a design must adhere. In other cases a system design might be validated by a resulting model.

An *executable model* is one that defines a computational procedure that mimics a set of properties of a system. Executable models might also be called algorithmic or computable models. A simulation is a special class of executable models. A *simulation* is an executable model that is distinct from the system it models. In some cases an executable model may start as a simulation and then evolve into a software implementation of the system. This is often the case in many electronic systems and results in a blurred distinction between a model and the system it represents.

Executable models operate according to a model of computation that specifies the interaction between components within the executable model. The set of interaction rules associated with a given model of computation are the semantics of the model of computation (MoC). In Ptolemy II, a model of computation is realized as a *domain*. All executable models that execute in a particular domain obey a common model of computation. Central to the beliefs of the Ptolemy Project is the maxim of heterogeneous semantics. The premise of this belief is that no single model of computation can effectively model all aspects of all systems. Instead complex systems are most effectively modelled by multiple models of computation with a given MoC being employed to design a particular subsystem as appropriate.

[2] The Ptolemy Project is a dynamic research initiative that is constantly being expanded and improved. For the most up-to-date Ptolemy Project description, see the following World Wide Web page: *http://ptolemy.eecs.berkeley.edu.*

4.2 The Ptolemy II Architecture

Ptolemy II is a second generation system implemented in the Java™programming language. The predecessor of Ptolemy II, *Ptolemy Classic*, was implemented in C++ in the early 1990's [Buck *et al.*, 1994]. Through its use of Java, Ptolemy II offers an infrastructure that is well suited to modeling heterogeneous semantics. Two key features of Ptolemy II that leverage Java are concurrent execution through the Java threading infrastructure and modularization through Java packages.

The threading support offered in Java can be very difficult to program correctly. The support is so low level that users who are not experts in concurrent programming can create software that is unpredictable and deadlock prone. Ptolemy II uses the threading infrastructure of Java to support models of computation that consist of autonomous components (components that control their own execution). In these *process domains*, each component is assigned its own thread of control. The process domains provide a "safety layer" on top of the threading infrastructure. This layer simplifies the use of Java threads by allowing a non-expert to correctly implement a concurrent program. Proper design of the process domains was made significantly easier through the aid of diposets.

The Java package structure allows for easy organization of Ptolemy II into subsystems. This is in contrast to many electronic design automation (EDA) tools that have large, monolithic designs that impose an "all or nothing" feel. In Ptolemy II, as long as package dependencies are not violated, programmers may use only the packages that are relevant to their needs. The package organization of Ptolemy II covers a wide set of semantics and execution features with over ten top-level packages (each of which may consist of several subpackages). The package structure of Ptolemy II is particularly useful in separating domains.

4.2.1 The Ptolemy II Packages

Figure 4.1 shows the key packages of Ptolemy II. Note that the figure consists of a Unified Modeling Language (UML) static structure diagram. UML is a widely used graphical modeling language for describing large, object-oriented software systems. The Unified Modeling Language fuses the best practices of the Booch and Object Modeling Technique (OMT) methodologies. There are several types of UML diagrams, each with special uses. In the case of the UML static structure diagram, syntactic relationships between classes are shown. Figure 4.1 shows how each of the

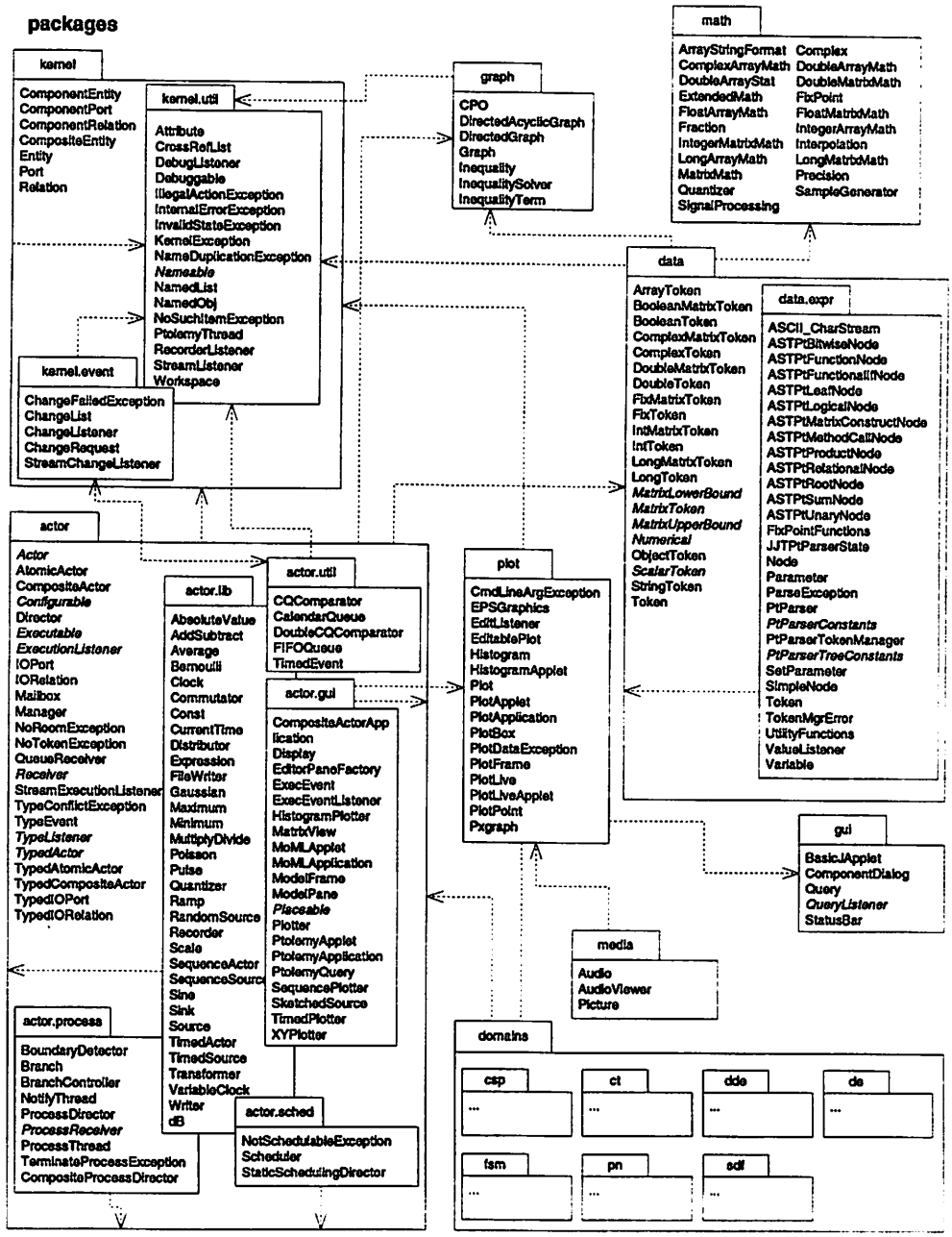

Figure 4.1. The Ptolemy II Package Structure

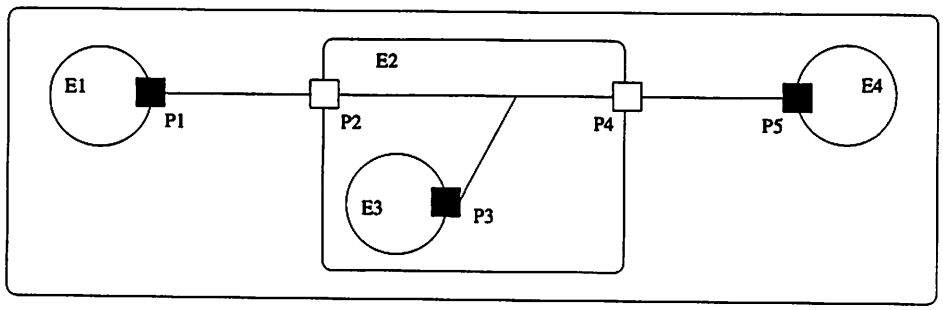

Figure 4.2. A Sample Ptolemy II Graph

Ptolemy II packages are related. Subpackages are shown by block diagram containment; e.g., the kernel package has two subpackages: kernel.util and kernel.event. Arrows represent dependency relationships. As an example, note that the graph package depends on the kernel.util package.

The kernel, actor and domains packages are of special relevance to this discussion. The kernel package, as its name implies, is at the core of Ptolemy II. The primary contribution of the kernel package is an abstract syntax. The abstract syntax of the Ptolemy II kernel allows one to specify hierarchical graphs. A *hierarchical graph* is one in which vertices of the graph may themselves contain graphs. The vertices of the hierarchical graphs in Ptolemy II are called *entities* while the arcs are called *relations*. Relations are connected to entities via *ports*. Note that there is no concept of a port in traditional graph theory [West, 1996; Chen, 1997]. Entities play the role of components (the term used in previous sections of this dissertation) and relations serve as the communication channels through which components communicate to one another.

Hierarchy is supported through containment. A *composite entity* may contain composite entities while *component entities* are a special class of composite entity that can not contain entities. We say that a component entity is *atomic* while a composite entity is not. A composite entity is *opaque* if its contents (the entities and ports that it contains) are visible outside of the composite entity. An opaque composite entity has *opaque ports* as opposed to *transparent ports* for entities that are not opaque. An example of a Ptolemy II hierarchical graph can be found in Figure 4.2. Note how composite entity E2 contains atomic entity E3 as well as ports P2 and P4. Note further that all

three atomic entities - E1, E3, and E4 - are opaque (indicated by the black shaded squares) while the composite entity E2 happens to be transparent with transparent ports (indicated by the white squares).

The hierarchical graphs that can be specified by the Ptolemy II kernel are strictly syntactic and can not be executed. The actor package adds semantics to the graphs and provides an infrastructure for execution. Specific semantics of execution are achieved in the domain packages. Each of the domain packages use the infrastructure of the actor package to implement a specific model of computation. Currently all domains except one realize a message passing form of execution. The one exception is the FSM (Finite State Machine) domain that implements an automata-based style of computation. In this document we are only concerned with the message passing domains and will not describe the architecture of the FSM domain.

The actor package introduces several key classes and interfaces relevant to message passing. These classes and interfaces facilitate executable entities that communicate data. We call these executable entities *actors*, a term inspired by Gul Agha's Actors model [Agha, 1986]. Informally our notion of actor is a node in a hierarchical graph that can process data. Formally an actor is an entity that implements the Actor interface, can contain IOPorts and has a Director and a Manager. *IOPorts* are extensions of ports through which data can flow. In Ptolemy II a unit of data is referred to as a *token*. IOPorts are directional and can be either inputs, outputs or both.

An actor in Ptolemy II is executable by virtue of the fact that it implements the Executable interface. As shown in Figure 4.3, the Executable interface consists of five *action* methods:

- `initialize()`
- `prefire()`
- `fire()`
- `postfire()`
- `wrapup()`

An *iteration* is defined to be one invocation of `prefire()`, any number of invocations of `fire()`, followed by one invocation of `postfire()`. An *execution* is defined to be one invocation of `initialize()`, any number of iterations, followed by one invocation of `wrapup()`. A *Director*

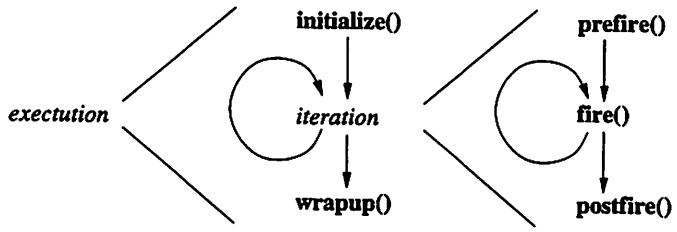

Figure 4.3. Execution and Iteration in a Sample Ptolemy II Model

controls the execution of a set of actors and determines an actor's model of computation. A *Manager* controls the execution of a complete model.

A director specifies an actor's model of computation by allocating an implementation of the *Receiver* interface to each of the actor's input IOPorts. A receiver is contained within an actor's IOPort and specifies how communication through the IOPort occurs. A receiver may support either asynchronous or synchronous message passing. In the case of asynchronous message passing, receivers are used to store tokens. A receiver may assume a notion of time associated with all tokens or it may assume no ordering constraints on tokens that it stores. For synchronous message passing, receivers have implicit states for indicating intermediate stages within a rendezvous. Each distinct implementation of the Receiver interface implies a distinct communication style for the actors that contain the receiver realizations.

Through allocation of receivers, a director controls both the communication and execution of an actor. This means that an actor's model of computation can change depending on the director that controls it. This is quite distinct from making an actor's model of computation an inherent quality. We refer to this characteristic as *domain polymorphism* [Lee and Xiong, 2000]. Through domain polymorphism, code reuse is facilitated: an actor with particular functionality can be implemented once and then used in multiple domains. Furthermore, the functionality of an actor can be changed at runtime with the substitution of a different director.

A very simple example of domain polymorphism is illustrated in Figure 4.4 consisting of two almost identical systems. Both the system on the right and left have a ramp source actor connected to a data plotter actor. The ramp source outputs a stream of increasing integer data values; e.g., 0, 1, 2, 3, ... The data plotter simply reads the incoming data and plots it to a screen. The ramp source and data plotter on the left and right are implemented identically. The difference is in

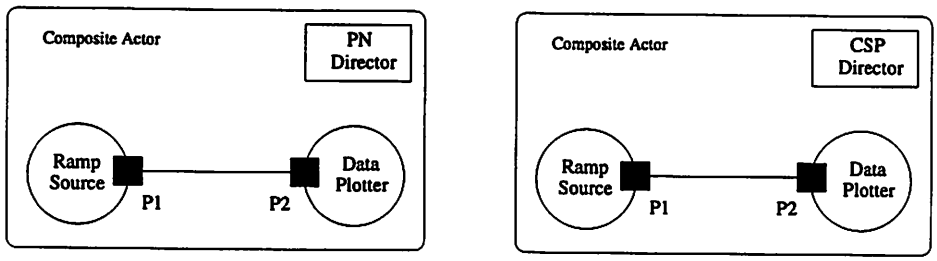

Figure 4.4. Domain Polymorphism: Identical actors in the left and right systems have different communication semantics because of their directors.

the respective directors. The system on the left has a Process Networks (PN) director. This means that input port P2 on the left contains a receiver that receives data asynchronously. The system on the right has a Communicating Sequential Processes (CSP) director implying that input port P2 on the right receives data synchronously. The result is that the two actors on the right execute at the same speed while it is possible that in the system on the left, the ramp source will execute much faster than the data plotter.

4.2.2 Hierarchical Heterogeneity

Ptolemy II supports hierarchical heterogeneity by allowing different directors to exist inside and outside of opaque composite actors. Figure 4.5 shows an example of hierarchical heterogeneity. Opaque composite actor E2 contains a synchronous dataflow (SDF) director implying that E3 executes with SDF semantics. External to E2 a process networks (PN) director is used implying that E1 and E4 execute according to PN semantics. Externally E2 acts like a PN actor while the internals of E2 execute according to SDF semantics.

A *boundary port* is an opaque IOPort contained on the boundary of a composite actor. In Figure 4.5, ports P2 and P4 are boundary ports. A receiver that is contained in, receives data directly from or transmits data directly to a boundary port is a *boundary receiver*. If a boundary port is an input port (i.e., data is transfered from outside of the containing composite actor to the inside through the port) then the boundary port contains boundary receivers external to the composite actor. If a boundary port is an output port (i.e., data is transfered from inside of the containing composite actor to the outside through the port) then the boundary port contains boundary receivers internal to

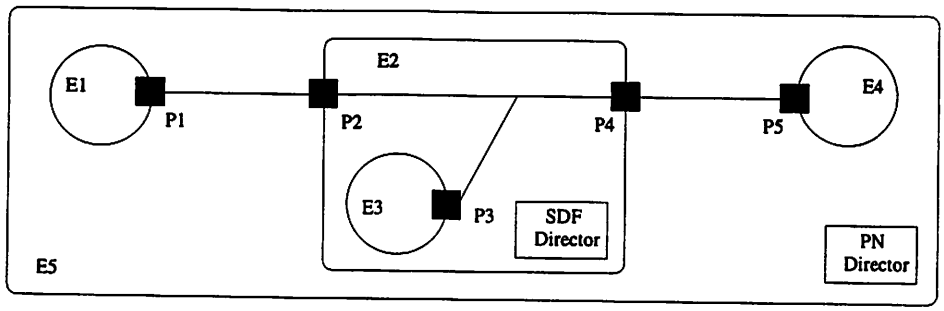

Figure 4.5. Hierarchical Heterogeneity in a Sample Ptolemy II Model

the composite actor. Figure 4.6 displays a boundary receiver contained in an input boundary port as well as a boundary receiver that receives data from the boundary port. Note that pairs of boundary receivers are associated with boundary ports (as is the case for the two boundary receivers in Figure 4.6 associated with the boundary port P2). Data transfer is directional for a pair of boundary receivers; i.e., data flows through one of the boundary receivers first (the *producer* receiver) and then flows through the second one (the *consumer* receiver). In Figure 4.6, the receiver contained in IOPort P2 is the producer receiver; the receiver contained in IOPort P3 is the consumer receiver.

4.2.3 The Process Package

The `ptolemy.actor.process` package (or *process package*) incorporates extensive use of Java™threads to facilitate execution in the process-oriented domains: PN, CSP and DDE.[3] In the schedule-oriented domains of Ptolemy II, each actor's executable methods are invoked by the controlling director. In the process-oriented domains each actor is assigned a unique thread by the controlling director. The director starts the thread and the thread invokes its assigned actor's executable methods. Once the director has handed control of the actors to their threads, the director then monitors the execution of the actors. The actors may continue executing until each actor voluntarily completes execution or until the set of actors deadlock. Determination of whether deadlock has been reached is made by the director who monitors the actors while they are being invoked by their threads.

Monitoring for deadlock is a significant difference between the process-oriented domains and the schedule-oriented domains. All receivers have `hasRoom()` and `hasToken()` methods

[3]The design of the process package excluding heterogeneous interaction was initiated by Mudit Goel and Neil Smyth.

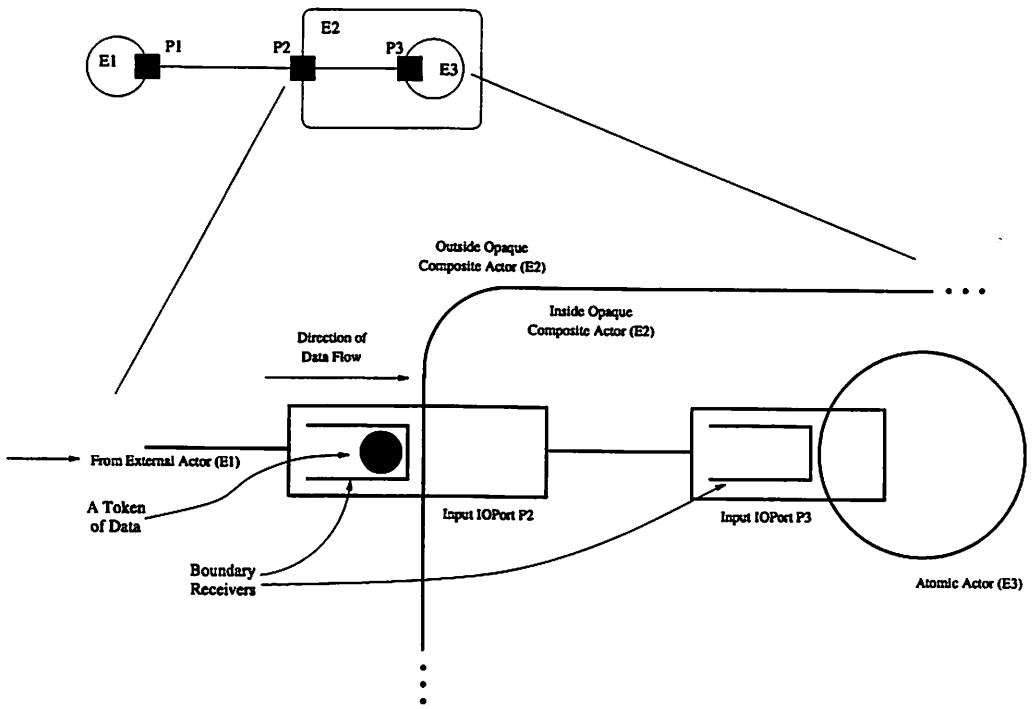

Figure 4.6. Boundary Ports and Boundary Receivers in an Opaque Composite Actor

for determining if an actor is able to transmit data or receive data through the receiver, respectively. The `hasRoom()` and `hasToken()` methods return true when communication through the receiver is enabled and false if communication is not enabled. The definition of *enabled communication* depends on the model of computation. Informally, deadlock occurs when all actors in a network block while attempting to communicate through receivers in which communication is not enabled. All receivers in the schedule-oriented domains implement the Receiver interface. The ProcessReceiver interface extends the Receiver interface and is implemented by each of the process-oriented domains. The ProcessReceiver interface is designed so that when blocking occurs, the total number of blocked actors can be monitored.

In schedule-oriented domains an actor will not attempt to transmit data through a receiver if `hasRoom() = false` for that receiver. Likewise a schedule-oriented actor will not attempt to receive data from a receiver if `hasToken() = false`. Process-oriented actors ignore the `hasRoom()` and `hasToken()` methods of the process receivers. If communication is not enabled a process receiver will force the calling actor to block and wait until communication is enabled. A blocked communication attempt in which an actor waits to receive data is called a *blocking read*. A blocked communication attempt in which an actor waits to transmit data is called a *blocking write*. A set of Ptolemy II process-oriented actors are *deadlocked* if all of them are blocked waiting to communicate.

Once deadlock has been reached in a set of process-oriented actors, the director has the option of resolving the deadlock so that execution can continue. Whether a deadlock can be resolved is domain-specific. In Communicating Sequential Process (CSP) models, for example, deadlock can not be resolved. In the case of Bounded Queue Process Network (Bounded PN) models, it is possible to resolve deadlock in which at least one of the actors is blocked waiting to put data in a full queue. Thomas Parks developed an algorithm for such deadlock resolution that can be applied at runtime [Lee and Parks, 1995]. In any process-oriented model, if deadlock is resolved then the actors continue execution until deadlock is reached again or until the actors voluntarily end execution. Using deadlock as the mechanism for stopping and starting actor execution leads to a unique definition of iteration in the case of process-oriented domains: *in process-oriented domains an iteration lasts until deadlock is reached.*

4.3 Hierarchical Heterogeneity and the Process Package

I extended the Ptolemy II process package to allow interaction between heterogeneous process-oriented domains. In accomplishing my task, I had the very important goal of maximizing code reuse. As outlined in Chapter 1, code reuse simplifies the software development process by allowing the work of individuals as well as groups of engineers to more easily share work. Code reuse allows engineers to leverage the past and prepare for the future. In leveraging the past, I recognized that my implementation was a small part of a large software project (Ptolemy II). Therefore, I introduced a system that did not require significant changes to the previous infrastructure. In preparing for the future, I designed a system that anticipated expansions by providing a suitably general set of classes that would remain useful as future researchers expanded Ptolemy II in years to come. Figures 4.7 and 4.8 consist of static structure UML diagrams of the process package classes and interfaces. Figure 4.7 focuses on the classes and interfaces that are used to monitor deadlock. Figure 4.8 consists of classes and interfaces associated with ProcessReceiver and the mechanism for detecting if a receiver is at a CompositeActor boundary.

My system architecture is founded on a simple dichotomy: *external vs. internal deadlock.* A network of actors is deadlocked if the actors have the same opaque composite actor container and they are each blocked waiting to write to or read from receivers contained in their network. A network of actors is *externally deadlocked* if the network of actors is deadlocked and at least one of the receivers involved is a boundary receiver. A network of actors is *internally deadlocked* if the network of actors are deadlocked and none of the receivers involved are boundary receivers.

Given the external/internal deadlock dichotomy my system works as follows. If the contents of an opaque composite actor are internally deadlocked, then the internal director has sole control. The director may attempt to resolve the deadlock or simply end execution of the deadlocked actors. If the contents of an opaque composite actor are externally deadlocked, then control of the situation is given to the director outside of the composite actor. In some cases, the external director will resolve execution and in other cases execution of the deadlocked actors will simply end. The result of this approach is that the special abilities of each model of computation are used when appropriate. I provide more detail on how my solution is implemented in the following section.

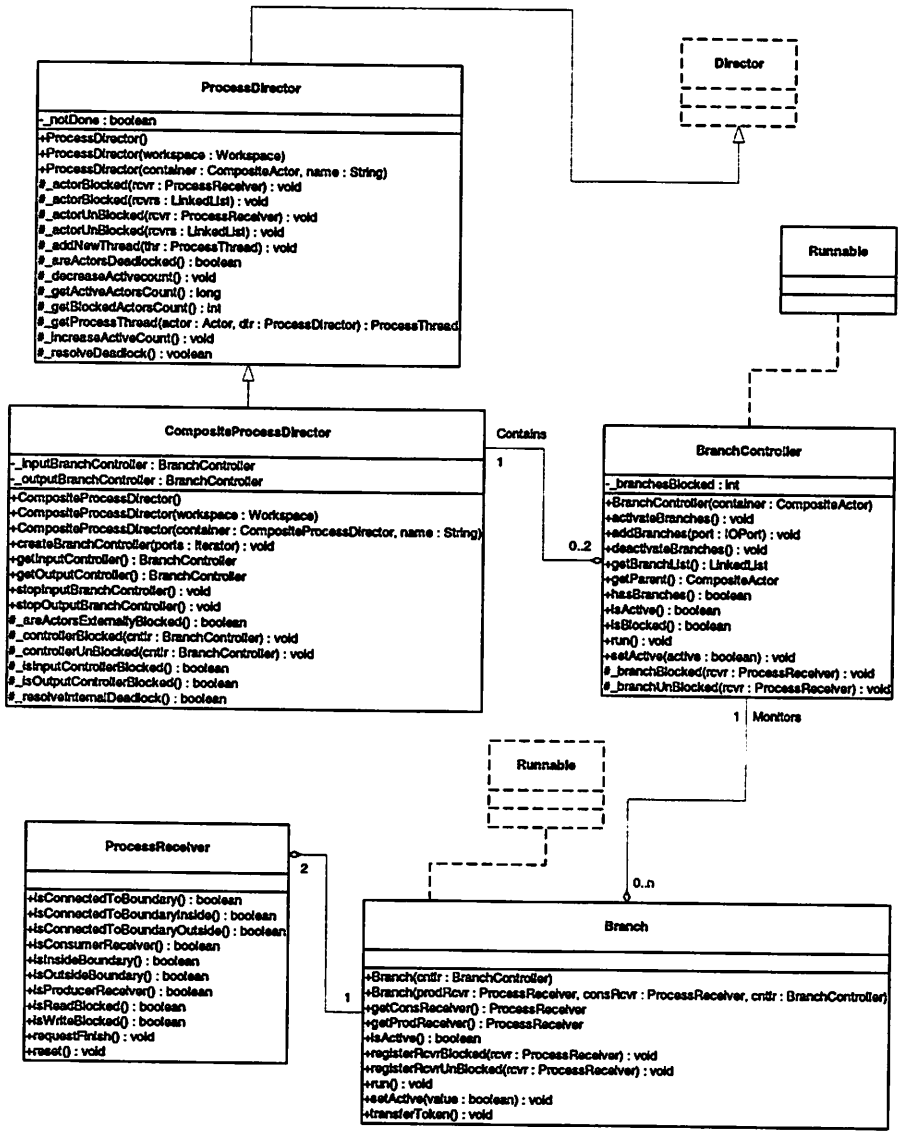

Figure 4.7. The Ptolemy II Process Package: Directors

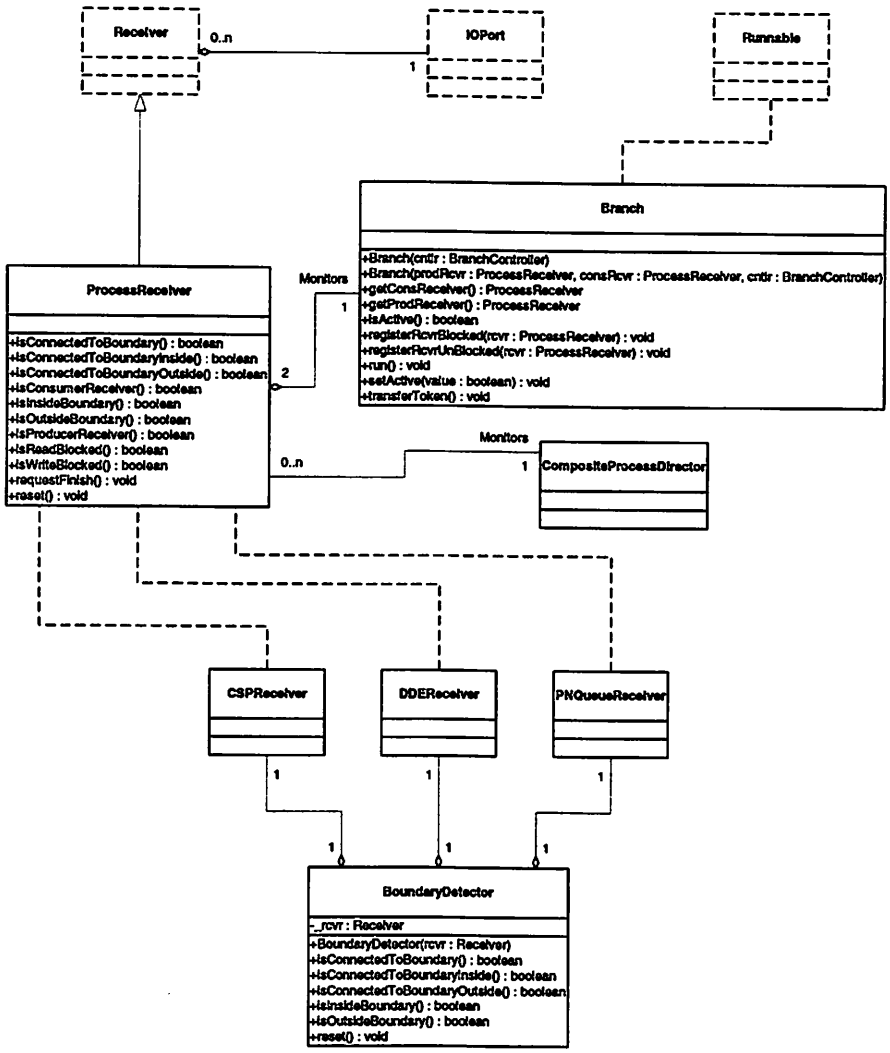

Figure 4.8. The Ptolemy II Process Package: Receivers

4.3.1 Controlling ProcessReceivers at CompositeActor Boundaries

The ProcessReceiver interface was initially written by Mudit Goel and Neil Smyth. It was designed to be implemented by the CSPReceiver, DDEReceiver and PNQueueReceiver classes. Each of these three receiver classes implements the `hasToken()`, `hasRoom()`, `get()` and `put()` methods to facilitate blocking reads and blocking writes. Prior to my extension of the ProcessReceiver class, it was assumed that any object calling the `get()` or `put()` methods of a ProcessReceiver object was an actor; the exception to this rule occurs in the case of CSPReceiver objects, in which `ptolemy.domains.csp.kernel.ConditionalBranch` objects call the receivers to support non-deterministic choice.

My extension of the ProcessReceiver class to facilitate heterogeneous interaction of process domains does not assume that the `put()` and `get()` methods of boundary ProcessReceivers are invoked by actors. In my extension, the `put()` method of boundary receivers contained in input boundary ports is called by an actor but the `get()` method is called by a special proxy. The `get()` method of boundary receivers contained in input boundary ports is called by an actor but the `put()` method is called by a special proxy. In both cases the proxy is realized by the `ptolemy.actor.process.Branch` class.

The `ptolemy.actor.process.Branch` class implements the Java™ Runnable interface. Thus, each instantiation of Branch results in a separate thread of control. Each Branch object is assigned to two boundary receivers. Branch threads are controlled by BranchController objects. Once a Branch thread is started by a BranchController, it attempts to repeatedly pass data between its pair of assigned receivers in the appropriate direction. As an example, consider Figure 4.6. In this case, a Branch object is assigned both boundary receivers shown. The Branch object repeatedly attempts to get data from the boundary receiver in the boundary port and put the data into the receiver of the internal actor.

As with all ProcessReceivers, boundary receivers can incur blocking reads or writes. In such cases the Branches controlling the blocked receivers must register the block with their BranchController objects. This procedure works as follows. Each opaque composite actor consists of two BranchControllers; the input BranchController and the output BranchController. The input BranchController controls N Branch objects that are assigned to a total of N boundary receiver pairs as-

sociated with the composite actor's input boundary ports. The output BranchController controls M Branch objects assigned to M boundary receiver pairs associated with the composite actor's output boundary ports. Each BranchController (input or output) is blocked when the boundary receivers of each of its Branches is blocked; the `BranchController.isBlocked()` method is used to determine such status.

The director inside of an opaque composite actor of a process-oriented model of computation monitors three states: the state of the input BranchController, the state of the output BranchController and the state of the contained actors. The primary state monitored by the director is that of the contained actors. Here the concern is whether the contained actors are deadlocked. Given that the contained actors are deadlocked, the secondary concern of the director is whether input or output BranchControllers are blocked. The action of the director given these states depends upon whether the director's opaque composite actor is contained by a composite actor that is process-oriented or schedule-oriented. Tables 4.1 and 4.2 summarize the actions taken.

Note in both tables (4.1 and 4.2) that the label `postfire() = false` indicates that the contained actors will no longer be permitted to execute. A label of `postfire() = true` indicates that execution may continue for an additional iteration. In several cases the tables indicate that the director will wait until the input or output branch controllers are blocked. In all such cases, blocked input or output branch controllers are imminent. For example, if the contained actors are blocked and the input branch controller is blocked (see the left column of Table 4.1), then the output branches will eventually have no data to transfer out of the composite actor and they will necessarily block.

4.3.2 Allocating Receivers at CompositeActor Boundaries

As with all receivers, ProcessReceivers are allocated to IOPorts by directors. Allocation occurs during an opaque composite actor's `initialize()` method prior to any iterations. Determining the placement of boundary receivers vs. *normal* receivers is a question of topology. One approach for allocating boundary receivers is to let the director determine which receivers should be boundary receivers and which should be normal receivers. The problem with this approach is twofold. First it requires two receiver objects (a boundary and normal receiver) for each model of computation. Maintaining consistency among two separate receiver objects is very difficult. The

second problem is that such an approach does not easily support mutable topologies. It is desirable to not have to replace or re-instantiate receivers if the topology changes (e.g., if a receiver is no longer connected to a boundary port).

I chose not to distinguish boundary and normal receivers. Any receiver can act as either a boundary receiver or a normal receiver; there is no separate class for the two types. To achieve this, each receiver contains a `ptolemy.actor.process.BoundaryDetector` object. An instantiated BoundaryDetector is contained in a receiver and provides the receiver with services for determining if it is a BoundaryReceiver. A BoundaryDetector provides such services via a rather expensive topological sort. Fortunately the result is cached and remains valid until a change in the topology occurs. Branch objects are assigned to receiver pairs as appropriate and the receivers contain the appropriate methods to be invoked by the Branch objects.

Prior to my extension, the `ProcessReceiver.get()` method contained no argument and returned a token of data (`ptolemy.data.Token`). Likewise, the `put()` method contained `ptolemy.data.Token` as the sole argument with a `void` return value. To accomodate the possibility of being part of a boundary, all ProcessReceivers must implement the following methods.

- `get(Branch)`
- `put(Token, Branch)`[4]

When a Branch calls either the `get()` or `put()` methods of a receiver, then it passes itself as the Branch argument. When an actor calls the `get()` or `put()` methods of a receiver the Branch argument is set to `null`. My approach has very few receiver methods that are required solely for boundary receivers. Two of these methods are (`get()` and `put()`). The other methods leverage BoundaryDetector which is a single class that can be used by every model of computation. The result is a very high level of code reuse.

4.4 Domain Polymorphism and Reorder Invariance

Domain polymorphism allows a component's model of computation to be changed during execution. The usefulness of domain polymorphism is that the semantics of a network of components

[4] *Token* = ptolemy.data.Token and *Branch* = ptolemy.actor.process.Branch

Contained Actors	Input/Output Branches		
	Input Blocked, Output UnBlocked	*Input UnBlocked, Output Blocked*	*Input/Output UnBlocked*
Internally Blocked	• Wait until Output Blkd • Deactivate Branches • postfire() = false	• Deactivate Branches • postfire() = false	• Wait until Output Blkd • Deactivate Branches • postfire() = false
Externally Blocked	• Wait until Output Blkd • Register block w/container	• Wait until Input Blkd • Register block w/container	• Do Nothing
UnBlocked	• Do Nothing	• Do Nothing	• Do Nothing

Table 4.1. Actor and Branch States when a Process is Contained by a Process

Contained Actors	Input/Output Branches		
	Input Blocked, Output UnBlocked	*Input UnBlocked, Output Blocked*	*Input/Output UnBlocked*
Internally Blocked	• Wait until Output Blkd • Deactivate Branches • postfire() = false	• Deactivate Branches • postfire() = false	• Wait until Output Blkd • Deactivate Branches • postfire() = false
Externally Blocked	• postfire() = true	• Wait until Input Blkd • postfire() = true	• Do Nothing
UnBlocked	• Do Nothing	• Do Nothing	• Do Nothing

Table 4.2. Actor and Branch States when a Process is Contained by a Non-Process

can be modified in a predictable manner while maximizing code reuse. Domain polymorphism frees a component to make only a minimal assumption about the model of computation in which it will operate. A component can assume a particular function but not be constrained to assume a particular style of communication for the function's input and output data, since the communication style may vary with the model of computation.

The Ptolemy II *Actor Library* package (`ptolemy/actor/lib`) consists of a large set of domain polymorphic actors. These actors do not assume a specific model of computation and are intended for use with a variety of the models of computation that come with Ptolemy II. Example domain polymorphic actors are listed below.

- `ptolemy/actor/actor/lib/Average`

  Outputs the average of M input values.

- `ptolemy/actor/actor/lib/Clock`

  Produces a periodic signal.

- `ptolemy/actor/actor/lib/Gaussian`

  Generates random numbers according to a Gaussian distribution.

- `ptolemy/actor/actor/lib/Sine`

  Produces an output that is equal to the $sin()$ of the input.

In Ptolemy II, all domain polymorphic actors attempt to consume input data through an input port polling mechanism. This means that as a domain polymorphic actor iterates, it checks each input channel in a round robin fashion and consumes data when available. The round robin order is based on topology. As an actor's channels are linked (connected) together, the order in which the actor will poll input channels is determined. If the linking order of an actor's input channels are changed, then the order of input channel polling will change as well.

Given the round robin polling mechanism of Ptolemy II, care must be taken when attempting to execute a domain polymorphic actor in certain models of computation. If a model of computation is not reorder invariant, then the use of domain polymorphic actors can lead to deadlock. The possibility for deadlock is quite subtle but very deadly. Consider Figure 4.9 in which both actor A and B are domain polymorphic. If actor A's top port is linked before A's bottom port while B's bottom port is linked before B's top port, then in a round robin polling scheme actor B would attempt to consume data from its bottom port first, while actor A would attempt to produce data on its top port first. In a non-reorder invariant domain such as CSP, this would lead to deadlock.[5] Fortunately the solution is simple. As long as the channels for actors A and B are linked in the same order, deadlock due to the round robin polling will be avoided.

[5]This type of problem initially was noticed with the Ptolemy II Butterfly Demo. Typically executed in the Synchronous Dataflow (SDF) domain, a user attempted to execute the Butterfly Demo in the Communicating Sequential Processes (CSP) domain. An inconsistency in the order of linked channels served as one of the first clues to the issue of reorder invariance.

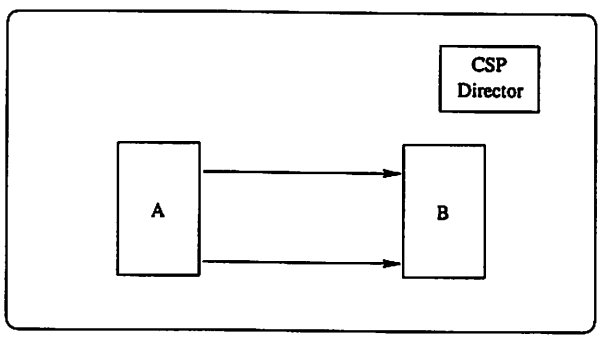

Figure 4.9. Deadlock Potential with Domain Polymorphic Actors

Chapter 5

Conclusion

What good is a new born baby?
- Benjamin Franklin, 18th Century[1]

In this dissertation, I consider the difficulty of modeling and designing complex, concurrent systems. By *concurrent* I mean a system consisting of a network of communicating components. By *complex* I mean a system consisting of components with different models of computation such that the communication between different components has different semantics according to the respective interacting models of computation. In Chapter 1, I showed how the components in a complex, concurrent computational system are related to one another. I recognized that two particularly important relationships found in complex, concurrent systems are the *order* relation and the *containment* relation. The order relation represents the relative timing of component actions within a concurrent system. The containment relation facilitates human understanding of a system by abstracting a system's components into layers of visibility. The consequence of improper management of the order and containment relationships in a complex, concurrent system is deadlock. *Deadlock* is an undesirable halting of a system's execution and is the most challenging type of concurrent system error to debug. In Chapter 2, I showed that no methodology is currently available that can concisely, accurately and graphically model both the order and containment relations found in complex, concurrent systems. The result of the absence of a method suitable for modeling both order and containment is that the prevention of deadlock is very difficult. To fill this void I created the diposet.

[1] Benjamin Franklin's question was made in response to the question "What good is a hot air balloon?"

5.1 Primary Contributions

- **I created the diposet for representing order and containment in complex, concurrent systems.** The *diposet* is a formal, mathematical structure that represents order and containment relations in a single entity. Chapter 2 consisted of several theorems and proofs demonstrating the ability to rigorously manipulate diposets.

- **I showed that the diposet robustly represents complex, concurrent computational systems.** I provided several examples that show that the diposet is well suited for graphically modeling significant systems. My examples illustrated that diposets can represent a wide variety of communication semantics including asynchronous and synchronous message passing.

- **I described how diposets can serve as the core of an automated compile-time deadlock detection mechanism.** I defined deadlock in Definition 2.10 and using this definition, I described a conservative approach for automatically determining the possibility of deadlock in software systems modeled by diposets.

5.2 Secondary Contributions

- **I introduced the concept of reorder invariance.** *Reorder invariance* is a characteristic of a model of computation that determines the possible order in which communications can occur for components in a concurrent system. Reorder invariance impacts how a model of computation supports domain polymorphism.

- **I implemented a software system to model and design complex, concurrent systems.** My implementation was part of the *Ptolemy Project* led by principal investigator Edward A. Lee at UC Berkeley. The software system used threads in the Java™programming language to support concurrency. Complex designs were facilitated through a run-time deadlock detection mechanism that incorporated hierarchical, heterogeneity.

5.3 Future Work

- **Diposets and Static Methods**

  While I explored the use of diposets for modeling a very broad set of software constructs, I did not consider static methods. Considering the best approach for modeling static methods with diposets would extend the applicability of diposets and further justify their usefulness.

- **Diposets and Shared Memory**

  I applied diposets only to message passing systems. The set of message passing systems is large enough to single handedly justify diposets. Nevertheless, shared memory systems are widely used (The Yale Linda Group[2], Javaspaces™, etc.) and merit consideration for being modeled by diposets.

- **Implementation of an Automatic Compile-Time Deadlock Detection Tool**

  A study of the feasibility of an automatic deadlock detection system is needed. It is certain that such a tool will be computationally complex. An implementation will clarify the practicality of such a tool.

5.4 Final Remarks

I have benefited tremendously from the work involved in this dissertation. I look forward to the opportunity to extend the concepts contained herein and to collaborate with other researchers on improving and expanding these ideas for the betterment of society.

[2] See *http://www.cs.yale.edu/Linda/linda.html*

Bibliography

Agha, G. A. (1986). *ACTORS: A Model of Concurrent Computation in Distributed Systems*. The MIT Press Series in Artificial Intelligence. MIT Press, Cambridge.

Alexander, C., Ishikawa, S., and Silverstein, M. (1977). *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press.

Allen, R. and Garlan, D. (1997). A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*.

Alur, R. and Henzinger, T. A. (1994). Real-time system = discrete system + clock variables. In T. Rus and C. Rattray, editors, *Theories and Experiences for Real-time System Development*, AMAST Series in Computing 2, pages 1–29. World Scientific.

Alur, R. and Henzinger, T. A. (1996). Reactive modules. In *Proceedings of the 11th IEEE Symposium on Logic in Computer Science*, pages 207–218.

Andrews, G. R. (1991). *Concurrent Programming: Principles and Practice.* Benjamin/Cummings, Redwood City, California.

Backus, J. (1978). Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Communications of the ACM*, **21**(8), 613–641.

Basten, T., Kunz, T., Black, J. P., Coffin, M. H., and Taylor, D. J. (1997). Vector time and causality among abstract events in distributed computations. *Distributed Computing*, **11**(1), 21–39.

Benveniste, A. (1998). Compositional and uniform modeling of hybrid systems. *IEEE Transactions on Automatic Control*, **43**(4), 579–584.

Bhattacharyya, S. and Lee, E. (1994). Looped schedules for dataflow descriptions of multirate signal processing algorithms. *IEEE Transactions on Signal Processing*, **42**(5).

Boehm, B. W. (1976). Software engineering. *IEEE Transactions on Computers*, **C-25**(12), 1226–1241.

Booch, G. (1994). *Object-Oriented Analysis and Design*. The Benjamin/Cummings Pulishing Company, Redwood City, CA, 2nd edition.

Booch, G., Rumbaugh, J., and Jacobson, I. (1999). *The Unified Modeling Language User Guide*. Addison-Wesley, Reading, MA.

Breal, M. (1991). *The Beginnings of Semantics: Essays, Lectures and Reviews*. Stanford University Press, Stanford, California.

Brookes, S. D. (1999). Communicating parallel processes. In *Symposium in Celebration of the Work of C.A.R. Hoare*.

Brookes, S. D. and Dancanet, D. (1995). Sequential algorithms, deterministic parallelism, and intensional expressiveness. In *22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Francisco, CA. ACM Press.

Brooks Jr., F. P. (1975). *The Mythical Man-Month*. Addison-Wesley, Reading, MA.

Brooks Jr., F. P. (1987). No silver bullet: Essence and accidents of software engineering. *Computer*, **20**(4), 10–19.

Brown, R. (1988). Calendar queues: A fast o(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, **31**(10), 1220 – 1227.

Buck, J. (1994). Static scheduling and code generation from dynamic dataflow graphs with integer-valued control streams. In *Proceedings of the 28th Annual Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, CA.

Buck, J., Ha, S., Lee, E. A., and Messerschmitt, D. G. (1994). Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal of Computer Simulation*, **4**, 155–182.

Chandy, K. M. and Misra, J. (1981). Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, **24**(11), 198–206.

Chang, K.-T. and Krishnakumar, A. S. (1993). Automatic functional test generation using the extended finite state machine model. In *Proceedings of the 30th Design Automation Conference*, pages 86–91. The Association for Computing Machinery, Inc.

Chase, C. M. and Garg, V. K. (1998). Detection of global predicates: Techniques and their limitations. *Distributed Computing*, **11**, 191–201.

Chen, W. (1997). *Graph Theory And Its Engineering Applications*. World Scientific, Singapore.

Chu, P.-Y. M. and Liu, M. T. (1989). Global state graph reduction techniques for protocol validation in the efsm model. In *Eighth Annual International Phoenix Conference on Computers and Communications*, pages 371–377.

Conway, M. E. (1963). Design of a separable transition-diagram compiler. *Communications of the ACM*, **6**(7).

Dalpasso, M., Bogliolo, A., and Benini, L. (1999). Virtual simulation of distriputed ip-based designs. In *Proceedings of the 36th Design Automation Conference*, pages 50–55. The Association for Computing Machinery, Inc.

Daniel, R. (1998). Embedding Ethernet connectivity. *Embedded Systems Programming*, **11**(4), 34 – 40.

Davey, B. A. and Priestley, H. A. (1990). *Introduction to Lattices and Order*. Cambridge University Press.

Davis, J. S., Galicia, R., Goel, M., Hylands, C., Lee, E. A., Liu, J., Liu, X., Muliadi, L., Neuendorffer, S., Reekie, J., Smyth, N., Tsay, J., and Xiong, Y. (1999). Ptolemy II: Heterogeneous concurrent modeling and design in java. Memorandum No. UCB/ERL M99/40, University of California, Berkeley, Department of EECS, University of California, Berkeley, CA, 94720.

Davis, R. E. (1989). *Truth, Deduction, and Computation: Logic and Semantics for Computer Science*. Computer Science Press.

Davis II, J. S., Goel, M., Hylands, C., Kienhuis, B., Lee, E. A., Liu, J., Liu, X., Muliadi, L., Neuendorffer, S., Reekie, J., Smyth, N., Tsay, J., and Xiong, Y. (1999). Overview of the ptolemy project.

Memorandum No. UCB/ERL M99/37, University of California, Berkeley, Department of EECS, University of California, Berkeley, CA, 94720.

de Bakker, J. and de Vink, E. (1996). *Control Flow Semantics*, chapter True Concurrency, pages 473–490. Foundations of Computing. MIT Press, Cambridge, Massachusetts.

Dijkstra, E. (1965). Solution of a problem in concurrent programming control. *Communications of the ACM*, **8**(9), 569.

Dijkstra, E. (1968a). The structure of the "the" multiprogramming system. *Communications of the ACM*, **11**(5), 341–346.

Dijkstra, E. W. (1968b). *Programming Languages*, chapter Co-operating Sequential Processes, pages 43–112. NATO Advanced Study Institute. Academic Press, London.

Fidge, C. J. (1991). Logical time in distributed systems. *Computer*, **24**(8), 28–33.

Foster, I. and Kesselman, C., editors (1999). *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1st edition.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Garey, M. and Johnson, D. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York.

Gibbs, W. W. (1994). Software's chronic crisis. *Scientific American*, **271**(3), 86–95.

Girault, A., Lee, B., and Lee, E. (1999). Hierarchical finite state machines with multiple concurrency models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits*, **18**(6).

Godefroid, P. (1996). *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State Space Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin.

Gordon, M. J. C. (1979). *The Denotational Description of Programming Languages*. Springer-Verlag.

Gunter, C. A. (1992). *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing Series. The MIT Press.

Halbwachs, N. (1993). *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, Dordrecht.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice-Hall International, New Jersey.

Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Massachusetts.

Hudak, P. (1989). Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, **21**(3), 359–411.

Jefferson, D. R. (1985). Virtual time. *ACM Transactions on Programming Languages and Systems*, **7**(3), 404–425.

Jones, C. B. (1999). Compositionality, interference and concurrency. In *Symposium in Celebration of the Work of C.A.R. Hoare*.

Kahn, G. (1974). The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress 74*, pages 471–475, Paris, France. International Federation for Information Processing, North-Holland Publishing Company.

Kahn, G. and MacQueen, D. B. (1977). Coroutines and networks of parallel processes. In *Proceedings of the IFIP Congress 77*, pages 993–998, Paris, France. International Federation for Information Processing, North-Holland Publishing Company.

Kienhuis, A. (1999). *Design Space Exploration of Stream-based Dataflow Architectures: Methods and Tools*. Delft University of Technology, Amsterdam, The Neterlands.

Kundu, J. and Cuny, J. E. (1995). The integration of event- and state-based debugging in ariadne. In *International Converence on Parallel Processing*, volume II, pages 130–134. CRC Press.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, **21**(7), 558–565.

Lea, D. (1997). *Concurrent Programming in Java.* Addison-Wesley, Reading, MA.

Lee, E. (1999a). Embedded software - an agenda for research. Memorandum No. UCB/ERL M99/63, University of California, Berkeley, Department of EECS, University of California, Berkeley, CA, 94720.

Lee, E. (1999b). Modeling concurrent real-time processes using discrete events. *Annals of Software Engineering*, **7**, 25–45.

Lee, E. and Messerschmitt, D. (1987). Synchronous data flow. *Proceedings of the IEEE.*

Lee, E. and Xiong, Y. (2000). System-level types for component-based design. Memorandum No. UCB/ERL M00/8, University of California, Berkeley, Department of EECS, University of California, Berkeley, CA, 94720.

Lee, E. A. (1997). A denotational semantics for dataflow with firing. Memorandum No. UCB/ERL M97/3, University of California, Berkeley, Department of EECS, University of California, Berkeley, CA, 94720.

Lee, E. A. and Parks, T. M. (1995). Dataflow process networks. *Proceedings of the IEEE*, **83**(5), 773–801.

Lee, E. A. and Sangiovanni-Vincentelli, A. (1997). A denotational framework for comparing models of computation. Memorandum No. UCB/ERL M97/11, University of California, Berkeley, Department of EECS, University of California, Berkeley, CA, 94720.

Luckham, D. C., Vera, J., and Meldal, S. (1995). Three concepts of system architecture. Technical Report CSL-TR-95-674, Stanford Computer Science Lab.

Lynch, N. A. (1996). *Distributed Algorithms.* Morgan Kaufmann Publishers, San Francisco.

Lynch, N. A. and Fischer, M. J. (1981). On describing the behavior and implementation of distributed systems. *Theoretical Computer Science*, **13**(1), 17–43.

Magee, J. and Kramer, J. (1999). *Concurrency: State Models & Java Programs.* John Wiley & Sons.

Mattern, F. (1989). Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226.

Matthews, S. G. (1993). An extensional treatment of lazy data flow networks. *Theoretical Computer Science*, **151**(1), 195 – 205.

Meyer, B. (1999). Every little bit counts: Toward more reliable software. *Computer*, **32**(11), 131–133.

Milner, R. (1989). *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, London.

Milner, R. (1993). Elements of interaction: Turing award lecture. *Communications of the ACM*, **36**(1), 78–89.

Milner, R. (1999). Computing and communication - what's the difference? In *Symposium in Celebration of the Work of C.A.R. Hoare*.

Minas, M. (1995). Detecting quantified global predicates in parallel programs. In *First International EURO-PAR Conference*, volume 966 of *Lecture Notes in Computer Science*, pages 403–414, Stockholm, Sweden. Springer-Verlag.

Morgan, C. (1985). Global and logical time in distributed algorithms. *Information Processing Letters*, **20**(4), 189–194.

Moschovakis, Y. N. (1994). *Notes on Set Theory*. Springer-Verlag.

Mowbray, T. J. and Malveau, R. C. (1997). *CORBA: Design Patterns*. Wiley Computer Publishing.

Murphy, D. (2000). Still flying high. *San Francisco Examiner*, page J1.

Murthy, P. K. (1996). *Scheduling Techniques for Synchronous and Multidimensional Sychronous Dataflow*. Ph.D. thesis, University of California, Berkeley.

Neggers, J. and Kim, H. (1998). *Basic Posets*. World Scientific, Singapore.

Nicollin, X. and Sifakis, J. (1994). The algebra of timed process, ATP: Theory and application. *Information and Computation*, **114**, 131–178.

Nygaard, K. and Dahl, O. (1981). *The Development of the Simula Languages*. Academic Press, New York, NY.

Passerone, R., Rowson, J., and Sangiovanni-Vincentelli, A. (1998). Automatic synthesis of interfaces between incompatible protocols. In *Proceedings of the 35th Design Automation Conference*. The Association for Computing Machinery, Inc.

Peterson, J. (1981). *Petri Net Theory and the Modeling of Systems*. Prentice Hall, Englewood Cliffs, NJ.

Petri, C. (1962). *Kommunikation mit Automaten*. German language, University of Bonn, Bonn, Germany.

Pino, J. L., Bhattacharyya, S. S., and Lee, E. A. (1995). A hierarchical multiprocessor scheduling system for dsp applications. In *Proceedings of the 29th Annual Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, CA.

Press, I. C. S., editor (1997). *Proceedings of the Sixth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, Tunis, Tunisia.

Righter, R. and Walrand, J. (1989). Distributed simulation of discrete event systems. *Proceedings of the IEEE*, 77(1), 99–113.

Rowson, J. A. and Sangiovanni-Vincentelli, A. (1997). Interface-based design. In *Proceedings of the 34th Design Automation Conference*, pages 178–183. The Association for Computing Machinery, Inc.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1991). *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ.

Schmidt, D. A. (1986). *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Newton, Massachusetts.

Schneider, F. (1997). *On Concurrent Programming*. Graduate Texts in Computer Science. Springer Verlag, New York.

Sebesta, R. W. (1996). *Concepts of Programming Languages*. Addison-Wesley, 3rd edition edition.

Shiple, T. R. (1993). Survey of equivalences for transition systems. Unpublished, Department of EECS, University of California, Berkeley.

Smyth, N. (1998). *Communicating Sequential Processes Domain in Ptolemy II*. Memorandum No. UCB/ERL M98/70, University of California, Berkeley, Department of EECS, University of California, Berkeley, CA, 94720.

Stoltenberg-Hansen, V., Lindstrom, I., and Griffor, E. R. (1994). *Mathematical Theory of Domains*. Cambridge University Press.

Sztipanovits, J., Karsai, G., and Bapty, T. (1998). Self-adaptive software for signal processing. *Communications of the ACM*, **41**(5), 66–73.

Tennent, R. D. (1991). *Semantics of Programming Languages*. Prentice Hall, London.

Tomlin, C., Pappas, G. J., and Sastry, S. (1998). Conflict resolution for air traffic management: A study in multiagent hybrid systems. *IEEE Transactions on Automatic Control*, **43**(4), 509–521.

van Glabbeek, R. J. and Weijland, W. P. (1996). Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, **43**(3), 555–600.

Vuillemin, J. (1974). Correct and optimal implementations of recursion in a simple programming language. *Journal of Computer and System Sciences*, **9**(3), 332–354.

Wegner, P. (1997). Why interaction is more powerful than algorithms. *Communications of the ACM*, **40**(5).

West, D. (1996). *Introduction to Graph Theory*. Prentice Hall, Upper Saddle River, New Jersey.

Wexler, J. (1989). *Concurrent Programming in Occam 2*. Ellis Horwood Series in Computers and Their Applications, England.

Wilson, L. B. and Clark, R. G. (1993). *Comparative Programming Languages*. Addison-Wesley, second edition edition.

Winskel, G. (1994). *The Formal Semantics of Programming Languages: An Introduction*. Foundations of Computing. MIT Press, Cambridge.

Yates, R. K. (1993). Networks of real-time processes. In *International Conference on Concurrency Theory*, Lecture notes in computer science, pages 384–397.

Young, J. S., MacDonald, J., Shilman, M., Tabbara, A., Hilfinger, P., and Newton, A. R. (1998). Design and specification of embedded systems in java using successive, formal refinement. In *Proceedings of the 35th Design Automation Conference*. The Association for Computing Machinery, Inc.

Yourdon, E., editor (1979). *Classics in Software Engineering*. Yourdon Press, New York, NY.

Yourdon, E. (1993). *Decline & Fall of the American Programmer*. Yourdon Press, Englewood Cliffs, New Jersey.

Appendix A

The Semantics of Programming Languages

Effective communication requires a well defined vocabulary as well as clear rules for how to use the words contained within the vocabulary. To precisely convey ideas and avoid misunderstanding, a vocabulary must clearly associate meaning to the words the speaker uses. The notion of associating meaning to words is embodied in the word *semantics*. When Michel Bréal introduced this word in 1900, it referred to the study of how words change their meanings. Since 1900, the meaning of semantics has itself changed and today semantics is generally understood as the study of the attachment of meaning to words or sentences.

There are three major branches in the discipline of semantics: natural language, mathematical logic and programming languages. In the case of natural languages, meaning is associated with words and phrases as spoken and written by human beings. With mathematical logic the words and phrases are expressions of logic . With programming languages the words and phrases are keywords and variables. Mathematical logic and programming languages share the trait of dealing with *artificial* languages, in that the languages of mathematics and programming are designed. This is in sharp contrast to natural languages which are not designed and exist prior to their study in a semantic framework.

A commonality between languages of all types is the requirement of an alphabet. An *alphabet* is a set of symbols or characters such as a, b, π and 5. Combinations of these characters lead

to strings of words and sentences. A language is simply a set of strings formed from a given alphabet. Language semantics assign meaning to the strings. The bedfellow of semantics is *syntax*. The syntax of a language provides rules for how characters can be correctly combined. A programming language requires a syntax and semantics which give meaning as well as rules for combining the keywords of a language. Closely related to a programming language is a *model of computation* (MoC). Informally an MoC is a programming language without an explicit syntax. There are three major approaches to the semantics of programming languages: axiomatic, operational and denotational.

A.1 Axiomatic Semantics & Predicates

In *axiomatic semantics*, the meaning of a string S is described in terms of a pre-condition and post-condition. A *pre-condition* of S is a predicate that holds true prior to the execution of S. Similarily a *post-condition* of S is a predicate that holds true after the execution of S. The goal of axiomatic semantics is to use rules of inference to deduce the effect of executing a program consisting of a set of statements. For this reason, axiomatic semantics are particularly amenable to proving properties about a given program.

A.2 Operational Semantics & Automata

Operational semantics defines an abstract machine with a set of data structures and operations. The semantics of the abstract machine are assumed to be known. The semantics of a particular programming language can then be described in terms of this abstract machine. The result is that operational semantics specify how the state of the abstract machine changes as a program is executed, or how a computation is carried out. This is similar in spirit to the notion of a Turing machine, which is effectively the canonical abstract machine. Operational semantics are particularly useful to compiler writers but often involve too much implementation detail to be of use by others such as language users.

In order to fully specify operational semantics, a technique must be available for describing the abstract machine. Most often, the abstract machine is represented by a transition system [Hopcroft and Ullman, 1979; Winskel, 1994; de Bakker and de Vink, 1996; Lynch, 1996]. de Bakker and de Vink provide a very general definition of a transition system as follows.

Definition A.1. TRANSITION SYSTEM

A transition system, $\mathcal{T}$, is a triple $(Conf, Obs, \rightarrow)$ in which

- $Conf$ is a set of *configurations*.
- Obs is a set of *observations*.
- $\rightarrow \subseteq Conf \times Obs \times Conf$.

□

Typically, the configuration of a transition system is based on a notion of state, an input symbol from an input alphabet and in some cases a notion of memory (representing past configurations). The set of possible configurations of a transition system consist of both *initial* and *final* configurations. The observations of a transition system are based upon actions that correspond to characters in an output alphabet. In some cases the set of observation actions can be empty, meaning that for each transition the null observation occurs. The transition relation $\rightarrow$ indicates how transitions can occur from one configuration to another and the observation that results.

There are several concrete examples of transition systems that may be familiar to many readers. A *finite automaton* is a transition system in which the configuration is based on a finite set of states, Q, and a finite set of input symbols, Σ. In the case of a finite automaton, the transition relation becomes a transition function in which the domain is $Q \times \Sigma$ and the range is Q. A *pushdown automaton* augments a finite automaton with an infinite stack. A *Turing machine* augments a finite automaton with an infinite capacity memory. As can be guessed based on the memory augmentation, a Turing machine is more complex than a pushdown automaton which is more complex than a finite automaton.

A.3 Denotational Semantics & Recursion

While operational semantics focuses on the "how" of execution, *denotational semantics* focuses on the "what." Denotational semantics gives information on what mathematical function is being computed by a program. Operational semantics describes a program in terms of the understood meaning of an abstract machine. Denotational semantics describes a program in terms of the understood meaning of mathematical objects. For each entity (string) contained within a programming language, a mathematical object and function which maps the entity to the mathematical object is defined. The mathematical objects can then be rigorously manipulated, unlike their corresponding programming language entities. The name denotational semantics indicates the fact that mathematical objects *denote* the meaning of their corresponding entities.

Denotational semantics observes two principles, the first being that a program computes (or denotes) a particular mathematical function. The second (often called the *compositionality principle*) being that the meaning of a program is composed of the meanings of its syntactic parts.

Semantic denotation of a mathematical function is relatively straightforward if the function happens to be finite (i.e., the domain is finite). In such cases, the function of the program is simply the composition of the constituent functions at each step of the algorithm. Unfortunately, many interesting programs can only be represented by infinite functions. How do we describe the behavior of these infinite objects?

Clearly there are some infinite functions which are easy to describe. One example is the identity function, $I(n) = n$. Nevertheless, there are many more infinite functions that can not be easily described. A very important class of infinite functions include recursive and/or indefinitely iterative functions. A recursive definition of a given entity is one in which the name of the entity "recurs" in its own definition. A real world example of recursion can be found in two facing mirrors. If one attempts to describe the image in one of the mirrors, the description will contain itself. Engineering and mathematical examples of recursive functions include the factorial function and feedback systems. In indefinite iteration the number of repetitions of a repetitive operation are not known *a priori*, leaving open the possibility of an arbitrarily large number of repetitions. A while loop is an example of indefinite interation.

Recursion and indefinite iteration are interchangeable. Any function involving a while

loop can be rewritten using a recursive function and visa versa. Examples of recursion and iteration are pervasive in programming languages. [1] The popularity of these kinds of functions is due in part to the fact that although they are infinite, they can be easily specified using finite descriptions.

The key problem with a recursive definition is that it may not uniquely represent a function. Consider for example the following recursive definition operating on the integers:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ f(n+1) & \text{otherwise} \end{cases} \tag{A.1}$$

It is clear that $f(0) = 1$ and hence that f applied to any negative integer is also 1. Things become less clear if we apply f to positive integers. To make this plain, note that the following functions both satisfy Equation A.1:

$$f_1(n) = \begin{cases} 1 & \text{if } n \leq 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$f_2(n) = \begin{cases} 1 & \text{if } n \leq 0 \\ 2 & \text{otherwise} \end{cases}$$

There is no way to determine whether f equals f_1 or f_2 or for that matter any one of the infinite other possibilities. Part of the uncertainty relates to the fact that our definition of f itself depends on the definition of f. We can eliminate confusion by re-writing f as an argument of a function.

$$F(f;n) = \begin{cases} 1 & \text{if } n = 0 \\ f(n+1) & \text{otherwise} \end{cases} \tag{A.2}$$

F, which is sometimes called a *functional*, operates on a function and is completely defined. By substition we have that $\forall n \in domain(f), F(f;n) = f(n)$ or more succinctly $F(f) = f$. An equation of this latter form says that f is a *fixpoint* of F and hence the program computes a function which is a fixpoint of F.

There are many fixpoints of F including f_1 and f_2. Note that f is a function mapping integers to integers and, as with all functions, we can think of f as a set of ordered pairs. This begs the question, which set of ordered pairs (e.g., $f = f_1$ or $f = f_2$) is the best fixpoint solution for F.

[1] L. Peter Deutsch stated that to iterate is human but to recurse divine. Perhaps the problem is our desire to meddle in the affairs of God.

The standard approach for selecting the best fixpoint is to define a partial order on the sets of possible choices for f. In this case, set inclusion is used to order the sets. Thus, the fact that f_1 is a subset of f_2 means that f_1 is considered less than f_2. Indeed, based on set inclusion f_1 is less than every other possible fixpoint for F. We call this smallest fixpoint function the *least fixpoint*, and this reasoning leads us to interpret f as being equivalent to f_1.[2]

Dana Scott and Christopher Strachey developed denotational semantics in part to apply the above reasoning to recursive functions in programming languages. At the core of Scott and Strachey's denotational semantics is a theory of computation developed by Scott known as *domain theory*. Interested readers can find detailed expositions of domain theory in Davey and Priestley [1990]; Gordon [1979]; Stoltenberg-Hansen *et al.* [1994]; Tennent [1991]; Winskel [1994]. Additional discussion on partially ordered sets can be found in Section 2.1.1 of this dissertation.

[2]The intuition behind choosing the smallest function (or set of ordered pairs) for f is as follows. Smaller sets generally provide more information about their contents than larger sets. Certainly, the set of human beings (a very large set) implies less information than the set of 55 year old Nigerian males living in Alaska (a relatively small set). An emphasis on maximal information is common within the field of computer science.

Index