5-3-2007

Formal Object Interaction Language: Modeling and Verification of Sequential and Concurrent Object-Oriented Software

Jason Andrew Pamplin

FORMAL OBJECT INTERACTION LANGUAGE: MODELING AND VERIFICATION OF SEQUENTIAL AND CONCURRENT OBJECT-ORIENTED SOFTWARE

By

JASON ANDREW PAMPLIN

Under the Direction of Ying Zhu

ABSTRACT

As software systems become larger and more complex, developers require the ability to model abstract concepts while ensuring consistency across the entire project. The internet has changed the nature of software by increasing the desire for software deployment across multiple distributed platforms. Finally, increased dependence on technology requires assurance that designed software will perform its intended function.

This thesis introduces the Formal Object Interaction Language (FOIL). FOIL is a new object-oriented modeling language specifically designed to address the cumulative shortcomings of existing modeling techniques. FOIL graphically displays software structure, sequential and concurrent behavior, process, and interaction in a simple unified notation, and has an algebraic representation based on a derivative of the π-calculus.

The thesis documents the technique in which FOIL software models can be mathematically verified to anticipate deadlocks, ensure consistency, and determine object state reachability. Scalability is offered through the concept of behavioral inheritance;

and, FOIL's inherent support for modeling concurrent behavior and all known workflow patterns is demonstrated. The concepts of process achievability, process complete achievability, and process determinism are introduced with an algorithm for simulating the execution of a FOIL object model using a FOIL process model. Finally, a technique for using a FOIL process model as a constraint on FOIL object system execution is offered as a method to ensure that object-oriented systems modeled in FOIL will complete their processes based activities. FOIL's capabilities are compared and contrasted with an extensive array of current software modeling techniques. FOIL is ideally suited for data-aware, behavior based systems such as interactive or process management software.

INDEX WORDS: object-orientation, Formal Object Interaction Language (FOIL), concurrency, π-calculus, process verification, behavioral inheritance, formal methods

FORMAL OBJECT INTERACTION LANGUAGE: MODELING AND VERIFICATION OF SEQUENTIAL AND CONCURRENT OBJECT-ORIENTED SOFTWARE

By

JASON ANDREW PAMPLIN

A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

in the College of Arts and Sciences

Georgia State University

2007

by

JASON ANDREW PAMPLIN

Major Professor:	Ying Zhu
Committee:	Rajshekhar Sunderraman
	Roy Johnson
	Geoffrey Hubona
	Xiaolin Hu

Electronic Version Approved:

Office of Graduate Studies

College of Arts and Sciences

Georgia State University

May 2007

ACKNOWLEDGEMENTS

Many people assisted me in the formulation and production of this research. I would like thank Dr. Raj Sunderraman for spending time with me drinking coffee and reviewing the initial stages of the FOIL algebra with me. Thanks to my advisor Dr. Ying Zhu for his investment in me. Thanks to Dr. Roy Johnson for his critical eye for details and his exuberant moral support. He was most afraid when I took a job, my last year. His distraught look alone ensured that I would indeed finish. Additional thanks go to Dr. Hubona for his support even when he wasn't around and for Dr. Hu for asking the hard questions.

I would like to thank my children for their patience. While they didn't exactly have a choice, they handled my seemingly endless involvement with school and this dissertation with grace and support. I am truly blessed. Many thanks to my father, Paul, who managed to tolerate and support me all through high school. He really should have executed me but chose to spare my life. Thanks to Walt Lucas and my brother, Jeremy Pamplin, for putting up with my informal user trials on them.

Not surprisingly I would like to thank my wife, Dawne. Not only did she support me completely when I decided to quit my well-paying corporate job to pursue my Ph.D., but she has also proofread every paper that I have ever submitted EVER… including this one. She has spent long hours pursuing my ambition. Such support is hard to find in anyone, much less your God-given mate and life-long friend. It is safe to say that nothing that is presented here would be possible without her.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

The use of computers for information management is still in its infancy. While many advances have been made in the last four decades, it is obvious by looking at the history of other sciences that this is not very long. COBOL was the first widely adopted language developed specifically with the intent of managing information, but was mainly centered on the storage, access and viewing of data. A milestone in data management technology occurred in 1970 with the advent of the relational database [1] and the entity-relationship diagram offered in 1976 [2]. Structured query language (SQL) provided the ability to retrieve data from files quickly and easily; however, improvements in this form of data storage peaked in the early 1990's. The addition of new features such as different programming language support and generic drivers, while making access and programming to such systems easier, does not really enhance what can be done with the technology.

The limitations of the relational database management system (RDBMS) gave rise to a need for even more expressiveness in the data representation mechanism. Thus, while object-oriented languages have existed since the 1960's, their real benefit has not been fully realized. The creation and rapid adoption of Java as a programming language shows that developers of information management systems can use more expressiveness in their data modeling than a relational model could provide. Recent development and research points toward the adoption, over time, of full-fledged object management systems (OMS). There are already several commercially available object management systems.

Process modeling has taken a very different development path from that of information management. As a field of study, it is much older. Ancient civilizations produced amazing structures through the method of partitioning work into smaller tasks and forming a project by aggregating these pieces in the correct order or sequence. Modern process improvement and management came about during the industrial revolution of the mid-1800's when automation of some tasks by machine could be considered. The computer, especially the personal computer, is a machine that can automate administrative tasks in the same way that manual labor was automated in the previous century. The modeling of processes to be automated by computer naturally used the same methods as those used in machine-driven automation. Since early computers lacked the ability to execute anything but a purely procedural model (hence the use of procedural programming languages), this was not a serious problem. ***Thus, process modeling techniques continued to improve, but remained primarily procedural in nature***.

Today, procedural programming languages have largely been abandoned when projects require a large amount of code. Several million lines of procedurally-based code become unmanageable because developers lack the ability to memorize the code. Object-oriented software allows developers to model systems like the real world that they already know, thus providing easier management and comprehension of large projects. ***But, process modeling, primarily performed by business analysts, has continued along its procedural-oriented trajectory. As information management progresses toward a purely object-oriented architecture, compatible techniques for managing the next layer (i.e. the business layer) must be adopted***.

The Formal Object Interaction Language (FOIL), presented in this thesis, was developed with this goal in mind – ***to provide a comprehensive object-oriented framework for sequential and concurrent systems with a formal mathematical representation that can be used for verifying corresponding process models***. The following introductory sections summarize the current practice, set forth design goals, and define FOIL, concluding with the expected contribution of this work.

1.1 Motivation

Object-oriented software architecture has become the dominant architecture of choice for large software systems [3] over the last half-century. Modeling of object-oriented systems was made easier with the advent of the unified modeling language (UML) class diagram [4] which allowed for specification of objects and their relationships to each other in a way that could be used to generate code for production. Thus, UML has offered a significant improvement in specifying, documenting, and producing high-quality software.

UML is not, however, an ideal solution for modeling all software system types. In particular, software systems having high behavioral characteristics, as compared with their data and data manipulation requirements, become cumbersome and error-prone using UML, especially if the behavior of the system has a significant degree of parallelism. UML requires a large number of diagrams to completely specify a system's behavior and generally requires full structural specification to be completed first. Additionally, there is no inherent mechanism to ensure that the various diagrams are consistent.

This problem of diagram consistency is especially poignant when ensuring that the business requirements as provided by analysts are consistent with structural and behavioral requirements as provided by developers. This has relegated many in the field to refer to such ability to ensure congruency as “art”. Yet, in other engineering disciplines, the artistic aspects of design have more to do with solving problems associated with difficult or complex functionality while maintaining aesthetic appeal. The assurance that a design will perform its desired function once built is, in other engineering disciplines, decidedly more methodical and computational in nature.

The ability to model an object’s behavior is a prime concern as part of ensuring quality performance and accuracy during implementation. Yet, in an age of increasing use of mobile and distributed systems, few modeling techniques provide intuitive notations for representing concurrent behavior and interaction. Even fewer have a formal semantic for mathematically understanding this concurrent behavior once modeled. Of the modeling frameworks that do have these characteristics, many of them are difficult to read or have limited or no object-orientation.

In structural modeling, a diagram should show the data requirements as well as the relationships between data. Behavioral modeling must support concurrency to ensure that its expressiveness is sufficient. The model must support a process modeling capability that has the ability to be verified against the structural and behavioral aspects of the model. Historically, attempts to create a hybrid graphical modeling language have resulted in severe concessions of these requirements.

A detailed review of previous research in this area is given in chapter 2.

1.2 Design Goals

On one end of the software modeling spectrum is the Entity-Relationship (E-R) diagram [2] from which the UML class diagram is derived [5]. The E-R diagram is the most basic structural software representation, since a database, in its simplest form, does not exhibit behavior. The E-R diagram is easy to read and understand, and has relational algebra as a formal underpinning. These characteristics have made the E-R diagram a proven, time-tested modeling notation. The E-R diagram naturally led to the Unified Modeling Language (UML) class diagram. This transition disposed of any real mathematical basis for the language, but its simplicity and ability to specify the concrete and abstract structure of software has made it a resounding success.

On the opposite end of the software modeling spectrum is the Petri-net [6]. The Petri-net is presumably the most basic behavioral software representation. It has no problem modeling complex concurrent behavior and has an underlying mathematical foundation to minimize modeling errors and verify correctness. The notation has a small symbol set and is relatively easy to comprehend. These characteristics have made the Petri-net diagram a proven, time-tested modeling notation. Due to its ability to model concurrent behavior and general lack of structural specification, Petri-nets have been primarily used for process modeling.

The E-R diagram has no mechanism for modeling a software system’s behavior. The Petri-net, on the other hand, is strictly behavioral in its modeling and only accounts for data indirectly, meaning that additional data-based decisions in system behavior require extension of the model to include new places, transitions and tokens. Neither diagram is object-oriented, making comprehension and scalability of large models difficult. Object oriented modeling techniques such as UML do not inherently support a formal semantic.

Despite their shortcomings, these major software modeling frameworks have all enjoyed extended and wide-spread success. ***Based on the success of these modeling frameworks, the***

hallmarks of a long-lasting and widely accepted graphical software modeling language would be:

- ability to model software structure
- object-orientation
- simple easily-understandable notation
- inherent support for concurrency
- ability to model system processes
- an underlying mathematical basis

However, if a single uniform modeling language could meet all of these requirements, then there are other logical extensions that would follow. For instance, if behavior and process can be modeled then a more refined version of inheritance could be offered that comprises more than mere structural conformity. Finally, if the modeling of various aspects of software development – structure, behavior, and process – can be either integrated or verified against each other, then full software system verification can be performed.

1.3 Formal Object Interaction Language

This thesis presents the Formal Object Interaction Language (FOIL). ***FOIL graphically displays software structure, sequential and concurrent behavior, process, and interaction in a simple unified notation, and has an algebraic representation based on a derivative of the π-calculus [7, 8].*** This gives FOIL significant practical advantages over other graphical modeling languages, particularly for data-aware, behavior-based systems.

The FOIL notation borrows what is good in the Unified Modeling Language (UML) and adds a small set of symbols to allow the modeling of a class's behavior. Thus, in addition to providing the structural detail of a system's attributes and methods, a FOIL model provides a much more expressive picture of an object's:

- Instantiation – when, how and under what conditions objects are created at runtime;
- Behavior – how objects perform their work both internally and in relation to other objects;
- Collaboration – how objects interact with one another to perform work; and
- Constraints – the conditions necessary for object behavior.

The added behavioral notation in FOIL allows for expressing the internal control flow of an object including the splitting and merging of threads of execution. This ability to model concurrent behavior within an object is distinctive, but FOIL's support for concurrent processing of multiple instances of objects makes it truly unique. This behavioral notation allows for a more specific type of inheritance where objects are not generalized based on mere interface conformity but must also conform in their general behavioral characteristics.

The concurrent object modeling capability of FOIL has a well-defined mathematical representation derived from a well-known and time-tested calculus. This mathematical representation allow for the creation of laws, forms, and operations to be applied to the object model. This allows for the building of complete system expressions. Based on these expressions, certain properties of the object system, such as state reachability, deadlock capability and inherent inconsistencies, can be identified. Additionally, algebraic reductions can be done on these expressions during run-time to track full system state in an efficient manner. Most importantly, with the addition of some simple rules, the acceptability of certain behavior by a system can be determined and enforced to ensure that object systems perform as designed.

FOIL also supports a process modeling notation to allow for specifying what work a system is designed to perform. This provides a link between what is modeled in an object-oriented fashion and what is expected from a procedural standpoint. The FOIL process modeling notation is nearly identical to that used for the structural and behavioral specification of the object system. This makes FOIL the only graphical modeling language to use the exact same notational elements to represent the structural, behavioral and procedural aspects of a system.

Finally, a FOIL process model has an underlying algebraic representation whose construction is identical to that of an object model, allowing a single construction algorithm to be used for both. This process expression can be analyzed through a simulation technique to determine if a given object model can perform the provided process (achievability). More detailed analysis can show if a process can be determined to always complete (process determinism) or whether a process can complete regardless of independent internal control flow for a given object model (complete achievability). Most import, if a process model exhibits complete achievability against an object model, the algebraic process expression can be used as an enforcement constraint on object system execution to ensure that processes will complete.

1.4 Contribution and Application

The Formal Object Interaction Language (FOIL) is designed to be a complete and comprehensive graphical modeling language. FOIL is meant to have a user friendly graphical notation while providing more expressive power. It was intended that FOIL be able to model structure, behavior and process with a single notation, and with a common mathematical underpinning. Complete support for behavioral inheritance and concurrency were key design goals. Finally, the ability to verify that a process can be completed by an object model is a unique advantage. It is likely that there are modeling languages and frameworks that are superior to FOIL in one or more of these areas. This thesis was specifically written to show that FOIL is unique in its ability to perform well in ALL of these major design areas.

It is understandable that attempting to combine the structure, behavior, and process aspects of software into a single modeling framework would require trade-offs. But, most attempts to do so have resulted in major concessions in simplicity, expressiveness or formality which are the hallmarks of modeling frameworks that have experienced wide-spread acceptance and longevity.

Combining various aspects of a software system's structure, behavior, and process into a unified modeling notation have been attempted [9-16], but have had significant difficulties. One of the primary advantages of the UML class diagram is its simplicity [17]; thus, a new notation should have a small number of notational elements to maintain this quality. But, a new notation must also be expressive enough to provide for a detailed comprehension of the objects' behavior, both by humans and the underlying computational system. The FOIL notation (chapter 3) does this while adding only four new symbol types.

1.4.1 Single Unified Notation

Efforts have been made to combine various functional aspects of UML modeling diagrams, to form a more compact representation of a system. In 1991, shortly after the rise of UML, the object behavior diagram [16] was offered as a solution for compact representation, essentially combining the class and state diagrams (structural and behavioral aspects). A more recent effort was called object charts [12], and combined these two diagrams with more detail. The TROLL object-oriented specification language [13] allows for the combination of structure, behavior and process, but this language has been primarily textual with no completely defined graphical notation.

Efforts to combine the process and structure of software have also been offered. One method involved making UML activity diagrams object-oriented [18]. Another attempt at improving compactness was the development of object process methodology (OPM) [19], which combines the structural and process aspects of a system into a single diagram. Object Connectivity Nets (OCoN) [9] were developed to combine structural, process, and behavioral aspects of a system.

While all of these modeling systems have made progress toward a single-diagram notation, they all have significant drawbacks in one of two areas: mathematical basis or concurrency. The Formal Object Interaction Language (FOIL) offers a single-diagram notation without sacrificing mathematical basis, concurrency modeling, or object-orientation.

1.4.2 Concurrency

In UML, concurrency is supported at the process level through the activity diagram but modeling parallel operations on object states in the lower levels of system design requires the insertion of written notations. Object behavior [16] and object chart [12, 14] models assume that an object is in a single state; thus, these models do not support concurrency. The OPM [19] also has difficulty expressing concurrency. The Object Petri-net [15] is a successful blending of the concurrency modeling notation of Petri-nets with object-oriented design.

1.4.3 Expressive Power

A fully-expressive modeling system is able to correctly model all known types of event patterns [20], such as those with concurrency and resource dependency. It is challenging to model all patterns without adding additional complexity to the modeling framework. For example, UML models some of the more complex patterns by placing textual annotations on top of the graphical model [21]. It is desirable for the modeling framework to have sufficient expressive power to model complex patterns without sacrificing usability or formal semantic.

OCoN [9] models, with their very compact notation, are not suitable for complex patterns. In particular, concurrency is difficult to represent. In practice, all examples of OCoN models show sequential patterns. Object petri-nets [15] have excellent expressive power, showing concurrency and resource dependency easily. They have been demonstrated, in workflow modeling, to effectively model all known patterns. However, this expressive power comes with much complexity, as additional places, transitions and tokens are required for each resource dependency.

In addition to the above modeling systems, there are many others that are expressive enough to show all known patterns; many of these also supporting formal methods [22-24]. Some of these modeling frameworks lack a simple notation, or they only model process, neglecting structure and behavior. FOIL is based on π-calculus which has concurrency as its main advantage (chapter 4). Thus, FOIL easily handles concurrency while maintaining a simple object-oriented notation (chapter 5) that models structure, behavior and process (chapter 7). FOIL also has the expressive power to model all known workflow patterns (chapter 6).

1.4.4 Application

FOIL is a non-activity-centric model. Developers can work in an environment for process modeling that is closest to how they model systems. But, probably the most important aspect of this difference applies to how large organizations develop their processes. Using FOIL, individual groups can define and manage the processes for individual objects under their charge. They can respond to events that other groups respond to, but they define and control only their objects when such events are received. The system can aggregate these actions to events to form the typical UML Activity diagram. **This means that a single individual who understands the complete process diagram for an entire organization is no longer required.** This has profound ramifications to the development, management and maintenance of the FOIL system.

E-commerce, Enterprise Resources Planning (ERP) and workflow systems are just a few examples of software that require data manipulation, have a high behavioral component, are distributed and thus require the concept of concurrency, and need to be verifiable. These systems are becoming larger and more common. Yet, there is significant room for improvement in modeling data-aware, behavior-based systems that require concurrency. FOIL offers a complete modeling framework that fills the gap left by current modeling approaches.

2. BACKGROUND AND RELATED WORK

Design goals for software modeling languages discussed in section 1.2 include *object-orientation*, capability to model *process*, support for *concurrency* and a *formal* mathematical basis. Many software modeling tools have been developed that include some or all of these features. The following sections include a discussion of the best currently available models for meeting each of these goals individually. To conclude this chapter, special attention is given to the modeling methods that meet more than one goal. A thorough review of pertinent literature suggests that there is no comprehensive modeling language which adequately meets all of the given design goals. The Formal Object Interaction Language (FOIL), as described beginning in chapter 3 has all of these desirable characteristics.

2.1 Object-Orientation

Object-oriented systems have been around for nearly 40 years and have been shown to be the modeling method of choice for large software systems. The task of comprehending very large systems comprised of nothing but functions quickly becomes overwhelming. Object-oriented modeling allows programmers to comprehend software in the same way they comprehend everything else. Objects are created and, once created, they may interact with other objects. The concept of encapsulation is also familiar, as many real world objects have internal parts which, when performing as they should, can not be accessed by the average user.

2.1.1 The Case for Object-Orientation

There are multiple reasons to use an object-oriented approach to building software. Among them are:

- Increased code-reuse through generalization relationships
- Simpler code through the use of polymorphism
- Developer and user safety provided by encapsulation and data hiding
- Well defined application programming interface
- Current popularity
- Well-studied repository of known design patterns

The largest advantage of object-oriented design is the concept of real-world modeling. Object-oriented design simplifies requirements gathering. Such gathering is a matter of identifying the objects at work and determining their communication to each other. It is understood that there are other methods for software architecture, such as Aspect-Oriented and Service-Oriented; however, Object-Oriented (OO) software architecture has become the dominant method of choice for large software system development [3] over the last half-century.

2.1.2 Unified Modeling Language (UML)

Modeling of object-oriented systems was made easier by the advent of the unified modeling language (UML) class diagram which allowed for specification of objects and their relationships to each other in a way that could be used to generate code for production. The benefits in the specification improvements as well as the reduction in time spent coding basic functionality into software was impressive.

The history of the UML class diagram, as well as object-oriented programming languages, reveals that these techniques are really just layers added to the previously defined technology. This is intuitive since it is clear that a computer simply executes a series of ordered instructions and thus in itself has no concept of an object, class, inheritance, etc. These are merely abstractions built onto an existing functional programming framework. This is actually true of the UML class diagram as well. The similarities between the UML class diagram and the Entity-Relationship diagram are hardly coincidental.

2.1.3 Modeling Structure

In 1970, it was proposed that users should not have to know the internal structure of data on computer systems in order to access that data in a meaningful way. Thus, a relational abstraction was offered to achieve this purpose [1]. This later resulted in the creation of the structured query language (SQL) and the data definition language (DDL). Its simple grammar and easy-to-learn semantic has made it the most widely used programming language in the world. Surprisingly, the diagrammatic representation of this relational model was not offered until six years later in the form of the entity-relationship (ER) diagram [2]. The idea of abstracting data into an intuitive framework was brilliant and allowed the continued improvement of data management architectures without having to worry about whether users of such systems would have to keep up.

Advances in data storage technology continued with IBM and Oracle as the main players. This handled the problem of data complexity to some degree, but application code bases continued to grow and organizations increasingly found it difficult to manage them. Object-oriented software had been around since 1967 with the creation of Simula-67 but was not in wide use. The introduction of C++ by Bell Labs brought object-oriented programming to the mainstream; however, it was not until the mid-80's that *modeling* of object-oriented technology was offered. Object-oriented design offered many advantages over the traditional methods, despite the fact that there are minor differences in how such modeling is done [4, 25].

The basic concept of the class diagram is very similar to that of the ER diagram. Each class is represented by a box that lists the class name and the attributes that make up that class. In addition, the methods (i.e. functions) that can be performed by this class are listed. Different font types or colors indicate the scope and accessibility of attributes and methods in a class.

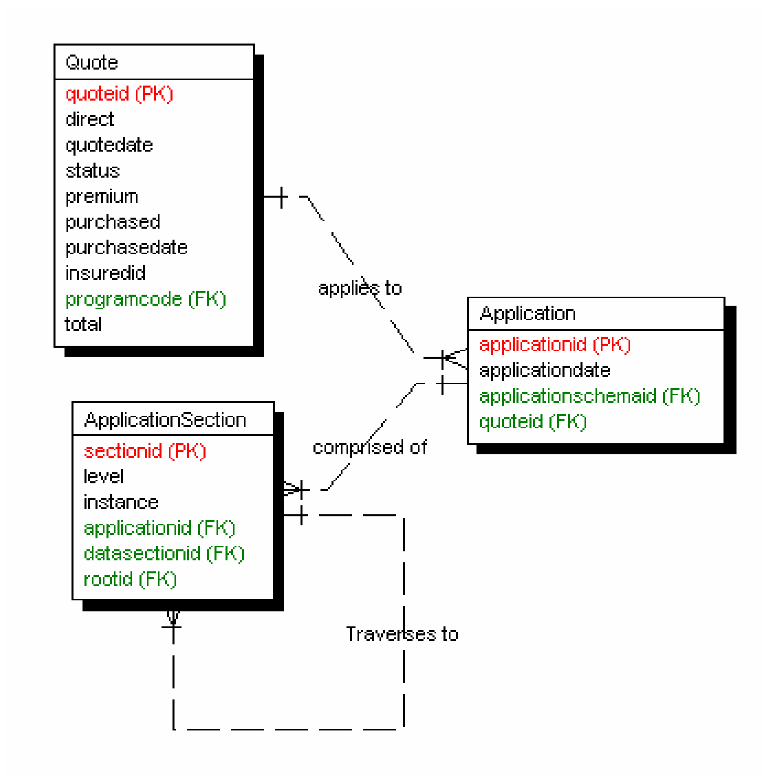


Figure 2.1 An ER Diagram

Connecting lines between classes show the relationship that classes have to one another. These lines have different shapes on the end of them to indicate what type of relationship exists between different classes.

In recent years, the concept of object persistence has been studied. Persistence is the permanent writing of an object to disk such that this object can be recreated from that data at a later time. It is easy to see that an ER diagram with a table existing for all persistent objects could be easily constructed. Likewise, an ER diagram can be transformed into an object diagram with additional information required. In fact, there are several frameworks that do this. Thus, for the set of persistent objects in a system, a class diagram represents a superset of detail required for an ER diagram [26].

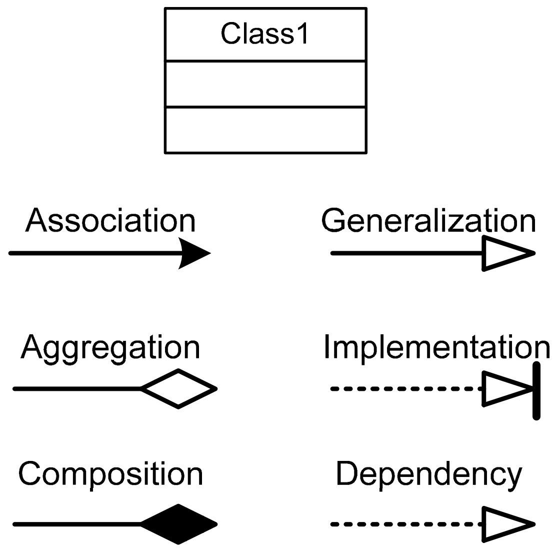


Figure 2.2 Class Diagram Legend

The fact that the data relationship can be inferred by the object relationship has resulted in the development of pure object management systems. These systems allow one to define objects with attributes, methods and relationships in a DDL-like language called object definition language (ODL). Similarly, one can query this system to retrieve actual instances of objects using the object query language (OQL) [27]. Many implementations are built on top of a relational database system.

2.1.4 Modeling Behavior

The behavior of objects is a determination of what happens to objects as activities are performed on them. Thus, this ties process to objects. It could be argued that process can be inferred from the recorded changes to object condition. This is the basis behind the technique of process mining [28]. It is safe to assume that the reverse it not true.

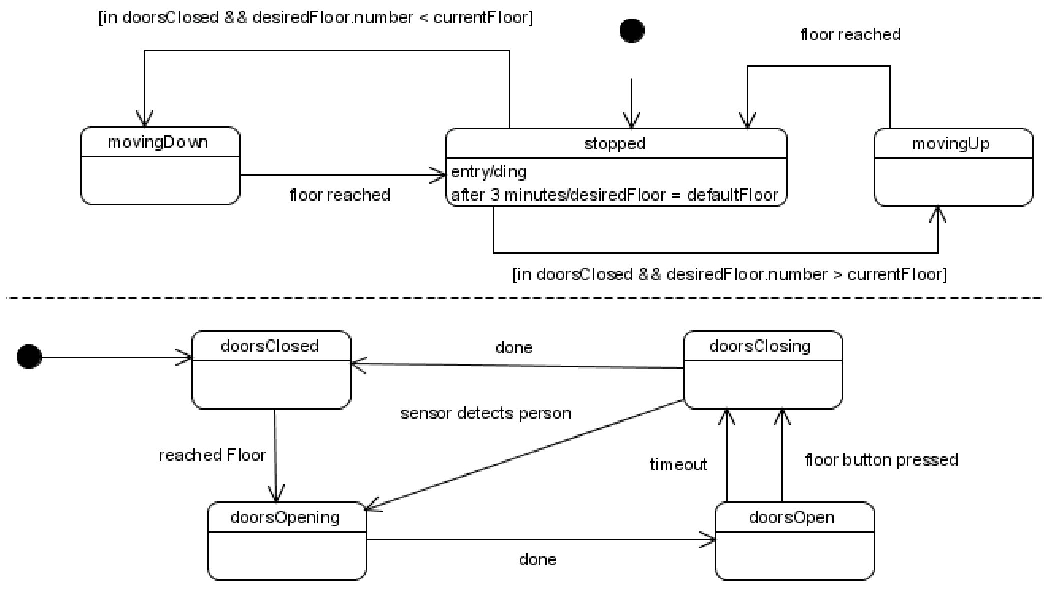


Figure 2.3 UML State Diagram for Elevator

The main method in UML for modeling behavioral changes to objects is the state diagram. The state diagrams in UML are basically comprised of boxes that represent states of an object. In this box is a list of events that cause transitions to other states. These transitions are represented by arrows. Attached to these arrows may be conditions that are evaluated to determine which transition is to be taken. Figure 2.3 shows an example of a UML state diagram.

It is interesting to note that the UML state diagram has no formal basis thus making correctness difficult to determine. It should be obvious from this statement that the state diagram offered by UML is not the same as that traditionally associated with finite state automata for which a well understood formal semantic exists. Non-determinism is difficult to model in the UML-style state diagram. This means that objects can generally never be in multiple states at the same time. While it is true that any system can be modeled in a deterministic way, it is also true that non-deterministic modeling can offer significant simplification of complex state changes. Since the state diagram (offered as optional for simple objects) is recommended for complex object behaviors, it stands to reason that this sort of modeling simplification would be needed.

The limitations of the state diagram have a direct impact on modeling processes themselves. Suppose for instance that three activities must be completed before an object's state changes to *complete* but the order of these activities in unimportant. From a workflow perspective this is a relatively simple pattern consisting of a parallel split followed by a synchronizing merge. However, with no ability to be in multiple states at once, how does one determine what has and hasn't been done to the object by looking at its state? The UML state diagram could be modeled to account for this but it would consist of six states. As the number *n* of prerequisites for completion increase, the number of states required to model this condition increases as a factorial of *n*.

So, what is the next layer in programming simplification? It seems that if object-oriented (OO) systems are comprised of a series of interactions between various objects, then modeling of the behavior of those objects would be beneficial. This is especially true if one considers the number of attributes and methods required in each object simply to store and modify and object's state. There are OO design patterns that can be used to make the state-based tracking of objects easier but the modeling of such abstractions make comprehension of what an object is actually doing quite difficult.

Interactive software systems can be especially hard to model as there are requirements for when and how objects can change state. Interactive systems of this nature are really a form of discrete event system; however, in such a system, the developer does not necessarily have control of when events will be received by the software. Thus, objects must be able to verify that they are in the correct state to respond to events. In addition, the system should be able to verify that processing of an event will not put the object or system in an unstable or deadlocked state. Thus, a modeling notation that can support an underlying formal semantic is preferable.

Creating a new notation that shows a class's structure and behavior in a single diagram with support for a formal semantic is difficult. One of the primary advantages of the UML class diagram is its simplicity; thus, a new notation must have a limited number of notational elements to maintain this quality. But, a new notation must also be expressive enough to provide for a detailed comprehension of the behavior of the objects both by a human and an underlying computational system.

2.2 Process Modeling

Process modeling is generally associated with an understanding of the dynamic behavior of an organization, business or system [29]. This should not be confused with the behavior of an individual object or entity within the system. A process model represents the "big picture" idea of what the business or system is actually accomplishing. This is highly useful in an organizational setting as it allows for analysis of whether or not the organization's goals are actually being met by the technology in use. In fact, it is a common (but not necessarily recommended) [18] practice to create a process model after a system is in place and functioning in order to determine what it is actually accomplishing.

The process model, while indispensable in analyzing organizational effectiveness, is not sufficient for the complete specification of a software system. There are several reasons for this.

For starters, process models do not, in and of themselves, contain the necessary level of detail required to completely specify a system. This is especially true for systems that are implemented in an object-oriented fashion. This means that the system is a combination of objects which communicate with each other in order to perform a particular task. This detail is generally not captured by a process model and, indeed, is not really even desired. Analysts, in general, are not concerned with the underlying implementation details. Rather they are generally analyzing whether organization goals are being met.

The historical approach to modeling a process or workflow is “activity” based. This is natural since most definitions of the term workflow deal with the sequencing of tasks (activities) for performing a given job. The terms “job”, “task”, “activity” and “process” are often used in interchangeable and confusing ways. There are currently two major standards bodies working on process modeling. The object management group manages the standard for the unified modeling language (UML) while the Business Process Management Initiative (BPMI) manages the business process diagram (BPD) standard. Both of these groups have similar approaches to dealing with workflow modeling but noticeable differences in their notational technique. Neither of these standards can model all of the workflow execution patterns identified by recent research.

2.2.1 Workflow Patterns

When most people think of workflow or process they generally think of a sequential set of activities performed by one or many individuals in a particular order. While this is certainly accurate in some instances it is an overly simplistic understanding of the problem. Since activities can be performed by one or more individuals, it is logical to assume that greater productivity can be gained by having separate individuals perform non-resource dependent activities concurrently. This is indeed the case; however, the complexity can continue to be compounded by the fact that resource dependency is not always predictable. The result is multiple patterns of workflow execution that can be quite complex.

Valuable and long-term research has been done on the various patterns that emerge in the process of modeling actual workflows. These have been collected and validated over many years through the input of people and organizations with actual experience in modeling business processes. These patterns range from simple to complex and offer significant challenges in finding a modeling technique with enough expressive power to accommodate all of them. The following list of collected workflow patterns comes directly from http://www.workflowpatterns.com [30, 31].

2.2.1.1 Basic Control Patterns

- Sequence - execute activities in sequence
- Parallel Split - execute activities in parallel
- Synchronization - synchronize two parallel threads of execution
- Exclusive Choice - choose one execution path from many alternatives
- Simple Merge - merge two alternative execution paths

2.2.1.2 Advanced Branching and Synchronization Patterns

- Multiple Choice - choose several execution paths from many alternatives
- Synchronizing Merge - merge many execution paths. Synchronize if many paths are taken. Simple merge if only one execution path is taken
- Multiple Merge - merge many execution paths without synchronizing
- Discriminator - merge many execution paths without synchronizing. Execute the subsequent activity only once
- N-out-of-M Join - merge many execution paths. Perform partial synchronization and execute subsequent activity only once

2.2.1.3 Structural Patterns

- Arbitrary Cycles - execute workflow graph w/out any structural restriction on loops

- Implicit Termination - terminate if there is nothing to be done

2.2.1.4 Patterns Involving Multiple Instances

- MI without synchronization - generate many instances of one activity without synchronizing them afterwards
- MI with *a priori* known design time knowledge - generate many instances of one activity when the number of instances is known at the design time (with synchronization)
- MI with *a priori* known runtime knowledge - generate many instances of one activity when a number of instances can be determined at some point during the runtime (as in FOR loop but in parallel)
- MI with no *a priori* runtime knowledge - generate many instances of one activity when a number of instances cannot be determined (as in WHILE loop but in parallel)

2.2.1.5 State-based patterns

- Deferred Choice - execute one of the two alternatives threads. The choice which thread is to be executed should be implicit.
- Interleaved Parallel Routing - execute two activities in random order, but not in parallel.
- Milestone - enable an activity until a milestone is reached

2.2.1.6 Cancellation Patterns

- Cancel Activity - cancel (disable) an enabled activity
- Cancel Case - cancel (disable) the process

2.2.2 Business Process Diagram (BPD)

The Business Process Management Initiative (BPMI) is a standards body working with other organizations such as the Object Management Group (OMG), Workflow Management Coalition (WfMC), and Organization for the Advancement of Structured Information Standards (OASIS). Together they collect the best of the industry in terms of process management practices and augment this with their own standards where none exists. These organizations have been very instrumental in raising awareness of many of the process management issues in the industry today.

BPMI has developed its own graphical process modeling notation known as a Business Process Diagram (BPD). This diagramming notation is basically activity-centric in its approach, combined with various symbols to show logical sequencing of activities. Figure 2.4 [21] shows three separate notations for modeling the parallel split workflow pattern. While these notations have minute differences in meaning, they are essentially the same. This notation struggles at times with over-complexity. This is also evident in the use of the diamond shape with a large number of symbols representing different forms of process splits and joins. This makes the notation difficult to learn and not very intuitive to the novice.

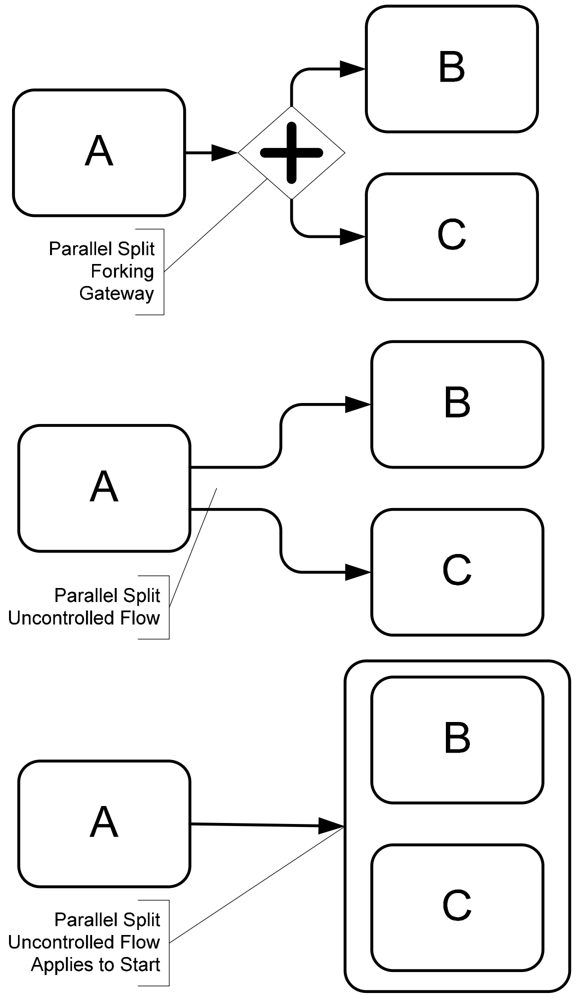

A
B
C
Parallel Split Forking Gateway
A
B
C
Parallel Split Uncontrolled Flow
A
B
C
Parallel Split Uncontrolled Flow Applies to Start

Figure 2.4 Business Process Diagram Notation [21]

2.2.3 UML 2.0 Activity Diagram

The Object Management Group (OMG) is heavily involved in the specification of the Unified Modeling Language (UML) as well as the Business Process Diagram (BPD). UML has become the most pervasive modeling framework in use today. UML has become popular because of the major need that it has filled and the language-independent results. The main contribution of UML to business process modeling is the use of the "Activity Diagram" [32]. Given the similarities in the UML Activity Diagram notation and the BPD it is reasonable to speculate that these notations will eventually be merged into a single specification.

Figure 2.5 [21] shows the basic parallel split workflow pattern as modeled in the UML 2.0 [33] Activity Diagram. The use of the synchronization bar makes this notation simpler than its BPD counterpart thus eliminating the primary drawback of the BPD. However, the notation has no built-in notational support for modeling different split patterns, such as a choice, without resorting to simply annotating the lines with conditional expressions. Of course, these conditional expressions could result in an exclusive choice, parallel split or multiple choice patterns based on how they are written. Thus, all three patterns have essentially the same notation and evaluation of the conditional expressions is involved in order to determine which pattern is being modeled.

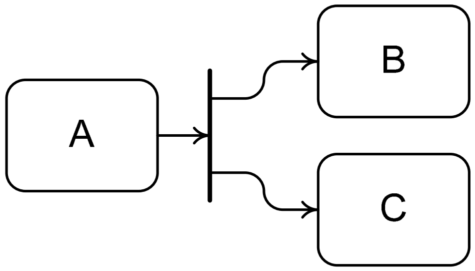


Figure 2.5 UML 2.0 Activity Diagram [21]

2.2.4 Critique of Current Practice

UML and BPD are the two major business process modeling frameworks in use today. While these notations have some significant differences, they suffer from some of the same problems. The problems with these notations are inherent to the underlying framework and assumptions that went into them. These problems are not a result of insufficient thought in improving current modeling techniques; the underlying assumptions and intentional limitations placed on that thought have limited growth potential. In fact, while many areas of software engineering have made significant improvements in the last decade, the lack of such improvement in the area of process modeling suggests that the current approaches have reached their upper bound.

2.2.4.1 Procedural in Nature

Some would argue that the modeling of the procedural aspects of a business, by definition, must also be procedural. However, all software is basically procedural in nature yet current software engineering practices use object-oriented approaches. As the complexity of software increases, the ability to model software in a human-friendly manner allows for the organization of these large projects to be more manageable. It can be argued that the same is true with workflow modeling.

The current approaches use the “activity” as their central figure. This approach can be merged into an object-oriented framework by using objects as inputs and outputs to these activities. These activities have objects (sometimes many of them) that are manipulated by the activities. In addition, the activity may also produce objects or cause changes to existing objects. These changes in object state are not modeled by the either the BPD or UML. Even with the number of different diagrams offered in UML in addition to the activity diagram, no single notation exists to correlate the business process with the production, manipulation or consumption of the objects modeled in the class diagram. A complete picture of a business process in UML requires a minimum of four diagrams which the developer has to jump between to gain enough information to program the application. This leads to the second major flaw of the approach.

2.2.4.2 Business Oriented

Many will argue that the procedural nature of current modeling techniques is inherent to the problem. A common assumption is that business people lack the ability to comprehend models designed for developers. Yet, object-oriented modeling was specifically designed to be a natural way of looking at the world. Humans, in general, think in an object-oriented manner. The UML activity diagram and BPD were specifically designed to be easy to understand for business analysts, but the sole purpose is undoubtedly to gather requirements for the development of software. Yet the conversion from a procedural process to an object-oriented framework is not intuitive and thus requires a great deal of effort to do properly.

In addition, the lack of expressiveness in the current modeling techniques makes converting complex workflow patterns into workable software a complex task, sometimes requiring the use of additional objects to control the activity flow. While current notations are useful for specifying procedures for business people, it is of little help to the developer.

2.2.4.3 Not Standardized

The ability to accurately model the procedural aspects of a business, organization or complex job is of immense value. Currently, there are numerous methods for modeling business processes, but no single standardized approach. There is also a large array of products claiming to model and implement business workflows. Some of these tools are very sophisticated but lack the full expressive power required to model many complex processes. Research on new workflow modeling techniques [22, 24, 34-38], which reached its height in the late 1990's, has slowed considerably in recent years despite the fact that there are many looming problems with the current state of the art.

Conceptual modeling is a core prerequisite for understanding and using a technology to the fullest. There have been attempts to address some of the issues involved with inconsistent modeling but they have not gained traction in either academia or industry. Many attempts to improve workflow representation merely attempt to augment or modify the current approach. Attempts to use non-procedural notation have resulted in systems with poor flexibility or usability. The poor uniformity and inadequate power of current modeling techniques ripple through other areas of the technology, making them less useful.

2.2.4.4 Complex Distribution Paradigm

It was not until the introduction of the Internet that large-scale distributed systems could be built cheaply. Unfortunately, the migration from the original single enterprise workflow systems to the web-based version has been accomplished by adding layer after layer of abstraction onto the existing paradigms [39]. This is why in workflow circles today, the base components are processes. In many implementations, such processes are wrapped as objects in an object-oriented system so that they look and behave like objects. Such band-aids only serve to complicate an already complicated process.

2.2.4.5 No Formal Semantic

The decision to not have UML tied to a formal language was a conscious one. It was believed that such ties would make the modeling framework too difficult to understand and manipulate. Some efforts to add a formal semantic to UML have been attempted [10, 15, 40-42] . Petri-nets have shown that for some complex applications a simple modeling notation can be both easy to understand and tied to a formal semantic. It could be argued that the lack of a formal semantic makes it harder to model complex systems in much the same way as writing a program without a debugger is difficult. In fact, the very languages that modern UML-based modelers generate have a formal semantic. This is a serious drawback to current business process modeling techniques.

2.2.4.6 Limited Visualization Capability

Obviously, with no consistency in notation, the visualization of a process varies a great deal. In addition, the current activity-based methods do not express enough detail to be truly useful to the software developer. However, some research has suggested the idea of using multiple perspectives to communicate the same model to different users. Combined with the use of modern 3D graphics technology, which is readily available in all new personal computers, visualizing a business process from different perspectives can be done in an intuitive and user-friendly manner. The addition of this third dimension allows for communication of information that is lost using current two-dimensional user interfaces.

2.3 Concurrency

Not much attention has been paid to modeling concurrency in the popular modeling notations. Yet, there is much recent research into concurrency support in languages and language extensions [43-45]. Moreover, research into code mobility [8, 46] and distributed systems [9, 47] shows a clear need for an object-oriented, graphical modeling language that has inherent support for concurrency.

The problem of concurrency in software modeling has been around for quite some time but few attempts have been made to address it. The introduction of Petri-nets [6] was a great milestone in modeling concurrent processes. The Petri-net’s use of tokens allows for intuitive understanding of concurrent actions. For complex systems, Petri-nets do not scale very well as new places must be added for each decision or data point required [48].

In UML, concurrency is supported at the process level through the activity diagram but modeling parallel operations on object states in the lower levels of system design requires the insertion of written notations. In addition, the difficulties in modeling concurrent systems in UML are well known [49]. Object behavior [16] and object chart [12, 14] models assume that an object is in a single state thus these models do not support concurrency. The OPM [19] also has difficulty expressing concurrency. The Object Petri-net [15] was a successful attempt to blend the concurrency modeling notation of Petri-nets with object-oriented design. However, this modeling framework suffers from the same scalability problems as straight Petri-net models.

A fully-expressive modeling system is able to correctly model all known types of event patterns [30], such as those with concurrency and resource dependency. It is challenging to model all patterns without adding additional complexity to the modeling framework. For example, UML models some of the more complex patterns by placing textual annotations on top of the graphical model [21]. It is desirable for the modeling framework to have sufficient expressive power to model complex patterns without sacrificing usability or formal semantic.

OCoN [9] models, with their very compact notation, are not suitable for complex patterns. In particular, concurrency is difficult to represent. In practice, all examples of OCoN models show sequential patterns. Object Petri-nets [15] have excellent expressive power, showing concurrency and resource dependency easily. They have been demonstrated, in workflow modeling, to effectively model all known patterns. However, this expressive power comes with much complexity, as additional places, transitions and tokens are required for each resource dependency.

In addition to the above modeling systems, there are many others that are expressive enough to show all known patterns; many of these also supporting formal methods [22-24]. Some of these modeling frameworks lack a simple notation, or they only model process, neglecting structure and behavior. FOIL is based on π-calculus which has concurrency as its main advantage. Thus, FOIL easily handles concurrency while maintaining a simple object-oriented notation that models structure, behavior and process.

2.3.1 Petri-Nets

This modeling technique was first introduced by Carl Petri in 1962 as part of his doctoral thesis. The concept of a Petri-net is quite simple. There are only two kinds of objects in a Petri-net, a place and a transition. A place is represented by a circle and a transition is represented by a thin rectangle. The Petri-net is primarily concerned with the movement of tokens. Directional lines connect places with transition with other places. These lines represent the movement of tokens in the model called *firing*. Each line can optionally have a number representing the number of tokens required to enable firing.

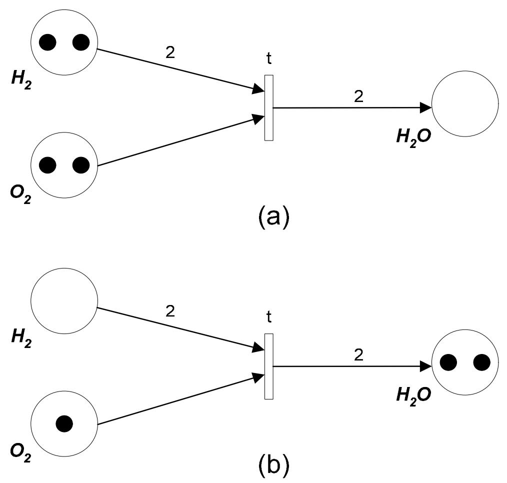


Figure 2.6 Example Petri-net [50]

Figure 2.6 [50] shows an example Petri-net [48] showing a basic chemical reaction of hydrogen and oxygen to form water. In this example, there are two initial places (markings) with two tokens each. The transition *t* is enabled when the token conditions represented by the arrows is met. In part (a) of Figure 2.6 this is true since the H_2 firing requires two tokens. Likewise, the O_2 requires only one token; two tokens exist, so that firing is also enabled. Thus, if all firings for a given transition (in this case *t*) are enabled then we say that the transition is enabled. The result is shown in part (b) of Figure 2.6. Notice that there is a remaining token in O_2 since only one token was consumed by transition *t*. Also, notice that the output of transition *t* is two tokens as indicated by the firing despite the fact that three tokens were consumed by transition *t*.

A Petri-net, in its essence, is really a weighted digraph with rules for token movement and manipulation. The Petri-net takes care of the non-deterministic way in which flows occur in the real world. Concurrency is inherent to the model. In fact, if concurrency is removed, what remains is a simple state diagram. Another great advantage is the existence of a formal specification, reduction, transformation and comparison framework which is very similar to that of basic push-down automata.

After their introduction in the 60's, the 1970's saw a great deal of interest in Europe on applying Petri-nets to various problems. The problems for which the Petri-net has been applied are too numerous to list. Some of the primary ones are workflow modeling, data flow modeling, complex state machines, and communication protocols.

The popularity of Petri-nets and their formal semantic have fostered much research into their capabilities. A Petri-net is characterized by several properties that determine what can be done with it. Some of them are:

- **Boundedness** – A Petri-net is bounded if its set of reachable places is finite.
- **Reachability** – this determines whether given an initial marking M_0 and another marking N, is there a set of firings for which a Petri-net can transition from M_0 to N.
- **Liveness** – a Petri-net is live if every transition which occurs can always occur again. This was shown to be recursively equivalent to reachability.
- **Deadlock Free** – a Petri-net is deadlock free if every reachable marking enables some transition.
- **Conflict Free** – for every place s that has multiple output transitions, every output transition of s is also one of its input transitions.
- **Free Choice** – whenever an arc connects a place s to a transition t, then a Petri-net is free choice if every transition t is the unique output for s or every place s is a unique input for t.

This does not represent a complete list of all the terms used to describe a particular Petri-net; however, they are the most important ones and generally determine whether other properties are decidable. For instance, it has been determined that reachability can be computed in polynomial time for bounded, conflict-free Petri-nets [51].

One of the major downsides of a Petri-net is its inability to account for data in its model. Modeling data specific choices into Petri-nets generally requires one or more additional places be added to represent that data. It was found that some applications of the technology were not feasible due to the number of places required to model them. One solution to the problem has been the introduction of a number of tools designed to help. Improvements to how Petri-nets are modeled have been offered to help resolve some of these complexity issues. The concepts of coloring and hierarchies allowed for the production of larger models with reduced complexity [52, 53]. The combining of these techniques is referred to as a high-level Petri-net [54].

2.4 Formal Methods

A successful modeling system is supported by formal methods that verify that the model does not contradict itself, and that it will function as designed. (That it will function as *desired* requires *good design*.) The creators of many modeling frameworks have intentionally declined to use a formal semantic, because formal methods add complexity to the model. The current modeling systems bear this out: the simple models (i.e. UML [4] and OPM [19]) do not support formal methods; while the more complex models (Object Petri-nets [15] and object charts [12]) do support formal methods.

The Object Constraint Language [55] has been offered as a gap-filler in the area of formal specification. This text-based language can be used to augment a UML diagram to provide a formal framework. Thus, the formalizing of UML using OCL or other methods [10, 42] does not have a strong graphical component. TROLL [13], which uses temporal logic, also suffers from little or no graphical correspondence. While these modeling languages can be viewed graphically, the mathematical underpinnings cannot be viewed in the same way. Object Petri-nets [15] and object charts [12] are supported by formal methods and have a well-known graphical semantic, but suffer from scalability [48] and expressiveness issues [12]. The FOIL model can display large, highly expressive models with minimal scalability issues while maintaining a mathematical foundation.

Process Algebra is the mathematical representation of a calculation, communication, or message passing system. Such a representation allows for formal reasoning about the equivalence of processes. Process calculi are not a recent invention, however, different calculi are being introduced regularly as scientists customize or refine the principles that go into them.

2.4.1 λ-Calculus

The λ-calculus is algebra used to represent sequential processes and can be considered the first process algebra. It was first proposed by Alonzo Church in 1936 as a way to determine computability for certain problems [56]. Church's λ-calculus allowed him to determine that the Entscheidungsproblem (English: decision problem) was not calculable. Incidentally, Alan Turing accomplished this same thing in the same year using a different approach which is now referred to as the Turing machine.

λ-calculus is based on the concept of binding variables, meaning that a defined variable may have any value until it is bound. The operator used to bind variables is λ in the form of λ*var*(*expr*)*arg* where *var* is the variable being bound, *expr* is the expression for which the binding is being applied, and *arg* is the value, expression or variable being bound to *var*. A variable is considered free if it is not bound to any particular value or expression. Thus, for example, in the expression $\lambda x(x+y)z$ the variable x is a bound variable while y and z are both free [57].

In the calculus, lower case letters represent variables and uppercase letters are used for processes. The distinction is based on the idea that processes may be defined as a relationship between variables in a different definition whereas variables are local in scope. The definition of process is done with the $\equiv$ symbol. Thus, we might define a process P as follows:

$$P \equiv \lambda x(x+x) \Rightarrow Py = y + y$$

As with any algebra, its utility relies on the ability to convert a particular statement into equivalent statements using defined rules. In the λ-calculus the main operation is called a reduction. Actually λ-reduction is a mixture of 3 separate reduction operations. β-reduction is the operation that does most of the work [58]. Β-reduction can really be considered a simple substitution as can be shown in the following example:

$$\lambda x(x\,y)z \xrightarrow{\beta} (z\,y)$$

Thus, β-reduction allows for rewriting complex expressions into simpler ones. Applying the β-reduction indiscriminately can result in expressions which are not equivalent. The following example demonstrates how a wrong result can be generated if only β-reductions are applied:

$$(\lambda x(\lambda y(x\,y))y)z \xrightarrow{\beta} \lambda y(y\,y)z \xrightarrow{\beta} (z\,z)$$

The reason for the error is that during the λ*x* operation the y is a free variable. Likewise, in the inner λy the x is a free variable. This problem is solved through the use of the α-reduction. The α-reduction allows the arbitrary substitution of any free variable. Using this reduction, the proper equivalent expression can be created:

$$(\lambda x(\lambda y(x\,y))y)z \xrightarrow{\alpha} (\lambda x(\lambda d(x\,d)y)z \xrightarrow{\beta} \lambda d(y\,d)z \xrightarrow{\beta} (z\,d)$$

This λ-reduction is correct. The final reduction available is called the η-reduction and stipulates that for any process P, λx(Px) is equivalent to P alone as long as there is no occurrence of x in P. This should be obvious as any β-reduction on x regardless of the argument value will result in P.

Of course, the λ-calculus is not suitable for algebraically modeling a distributed workflow system as it only functions in a sequential manner. Many processes could be executing in parallel. However, the λ-calculus is the basis from which most modern process algebras are derived. In particular, the β-reduction remains generally unchanged from one calculus to another.

2.4.2 π-Calculus

While the λ-calculus can be considered the first process algebra, it was not originally invented for that purpose. In fact, the term “process algebra” is a relatively new term in computer science. The first process algebra to be referred to as such was called Communicating Sequential Processes (CSP) in 1984 [59]. This was the first calculus to consider a variable as simply a communication. From a high-level perspective this makes sense. If you consider that a computer must perform some sort of operation in order to access memory to retrieve a variable value, then a function, communication or variable are all really the same thing. CSP as the name implies, however, was still sequential in nature and thus not suitable for distributed computational modeling.

In 1982, Robin Milner introduced the Calculus of Communication Systems (CCS) [7]. This calculus modeled the communication of two distinct entities that could occur in parallel. This introduced the concept of parallelism into process algebra. In 1999, he introduced the π-calculus [8] which added the concept of mobility to the algebra. The π-calculus is based on the concept of naming [60]. In other words, everything in the π-calculus is a name that represents a communication channel. Thus, when a process passes a variable in π-calculus it is really passing a communication channel for accessing that variable [61]. Thus, the actual location of that variable is not important.

The notation of the π-calculus is somewhat different than the λ-calculus but uses some of the same elements. Upper case letters still represent processes but lower case letters represent names of a communication channels used to access resources. The following is a list of the constructs used in the π-calculus:

- **P|Q –** Process P executes concurrently with Process Q.
- **P.Q –** Process P and Q execute sequentially
- **x(y).P –** wait to receive a communication on channel x, bind the input to y and then execute process P.
- **ōu.P --** output value of u over channel o then execute P. It should be noted that P will always execute regardless of whether another process receives u or not.
- **!P** – execute P one or more times concurrently.
- **(νx)P** – create a new communication channel x available to process P only. Another way of saying this is, "Process P creates a new channel x".
- **P.0** – Execute P and then terminate.
- **P+Q** – Execute either P or Q but not both.

The π-calculus can be used to show that two processes are equivalent through the use of reduction rules. The main reduction rule which demonstrates the ability for processes to communicate is:

$$\overline{x}y.P \mid x(z).Q \rightarrow P \mid Q[y/z]$$

This says that when y is output on channel x then P and Q will execute concurrently with z substituted for y in Q. In other words, a message is received on x which was transmitted as y but will be assigned as z, then Q will execute. Note that P would execute regardless of whether any other process received the y sent along channel x; however, Q will not execute until it has received something (which it will call z) on channel x. Additional rules are:

- $P \rightarrow Q \equiv P \mid E \rightarrow Q \mid E$ - concurrent operations can never inhibit computation.
- $P \rightarrow Q \equiv (\nu x)P \rightarrow (\nu x)Q$ - restrictions on scope can never inhibit computation.

- $P \equiv P' \; and \; P' \equiv Q' \; and \; Q' \equiv Q \Rightarrow P \equiv Q$ - concurrency is both commutative and associative.

The syntax for various flavors of π-calculus may vary, but generally they are the same. They always have some representation for actions, sequence, parallel composition, synchronizing actions, nondeterministic choice, emission, reception, process, local process, and recursive process. One notation that will be used is the action label notation:

$$P \xrightarrow{\alpha} Q$$

This indicates that P after completion of action α will become Q. This allows for modeling of mobile, distributed event-driven systems. In fact, π-calculus has already been used to model many different types of systems, including workflow systems [62].

2.5 Synergistic Attempts

A complete survey of currently proposed frameworks for modeling software is beyond the scope of this paper. The body of knowledge in this area is far too large. This following is a brief survey of models or frameworks which are of significance in designing a new way of thinking about workflow and a new approach to modeling them.

2.5.1 Objects-Rules-Roles

The best attempt to date at a full-fledged object-oriented approach to modeling workflow separates data (objects), flow (rules) and users (roles) [23]. This approach does not offer a visual model of the workflow or even a unified conceptual view of a workflow. The proposed system requires the use of inheritance or composition to model a given workflow using abstract workflow and data components. This approach has significant problems and does not even supply a modeling or workflow specification language.

The reason this framework is notable stems from its attempt to use a purely object-oriented framework to implement a workflow system. This is the only system surveyed here that is not activity-based. In fact, activities can be abstracted from rules as to how objects interact with each other as would be done in any object-oriented implementation of a workflow. In addition, this model is event-driven rather than activity-driven. Thus, the performance of activities can be done by the workflow system or any other outside system. Thus, this model and SEAM are the only ones to specifically address and cater to workflows performed by computers in a heterogeneous environment.

2.5.2 SEAM – State-Entity-Activity-Model

A recent attempt to unify models into a design that can take advantage of formal methods is called the State-Entity-Activity-Model (SEAM) [22]. This model is based on set theory and provides a single view of the workflow pattern rather than many different views used by current mainstream techniques.

SEAM starts by modeling entities. This process is a good idea as it makes translation to an OO framework relatively straightforward for the developer. Entities can be modeled to have attributes but not methods – precluding a complete OO implementation. However, this is still easier to translate to OO than mainstream process modeling techniques. The entity-attribute is similar to the standard ER diagram, which makes sense, given that implementation has been on a standard RDBMS.

SEAM also attempts to make the model and language temporal. This is a good idea as workflows are, by their very nature, temporal. This is done, however, by adding temporal components to the language and the corresponding underlying database rather than using an inherently temporal database system [63].

Figure 2.7 [22] shows an example SEAM. As can be seen, the model is not entirely intuitive and the complexity of the language specified is fairly significant. Thus, there is quite a large learning curve in dealing with this model. In addition, the limitations in the actual "flow" modeling mean that modeling complex patterns is either very difficult or completely impossible. In addition, the model complexity and learning curve make it unlikely to be used by business professionals.

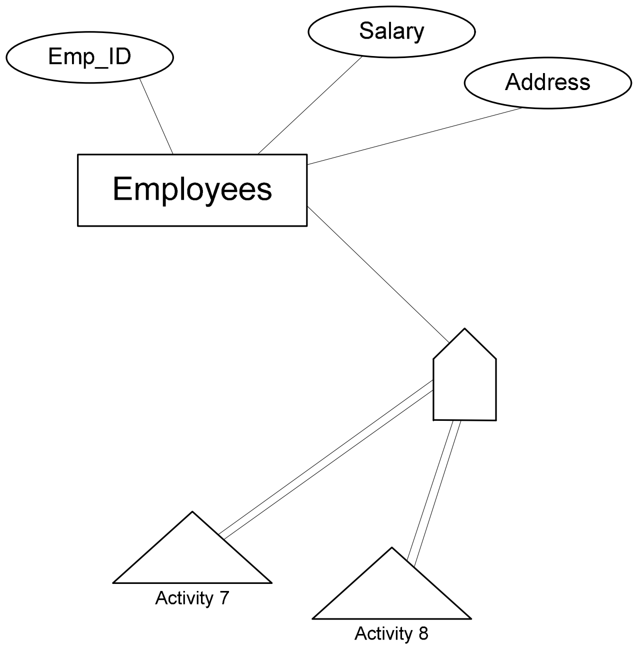


Figure 2.7 SEAM Example Model

SEAM is a good attempt at simplifying workflow modeling for the developer. This is done by having models that can be tested with formal methods as well as having a single view of the model which includes both data and process. It is a non-activity centric model that is very scalable. This model represents the best step in the direction of viewing workflows differently; any new attempts at workflow modeling would benefit from becoming familiar with this framework.

2.5.3 Petri-Net Workflow

Petri-nets are a token-based flow modeling system and have been used in a variety of applications such as logistics, controllers and protocols. They can be tested with formal methods and easily deal with difficult resource management, concurrency and data flow complexity issues. Many workflow systems use the concept of tokens, or threads of execution, to delineate when processes split or merge in either a synchronous or asynchronous way. Thus, using Petri-nets to model workflows is a logical choice.

Figure 2.8 shows an example of one technique for using Petri-nets to model a workflow system. This technique alternates the activities of the workflow with Petri-net nodes that manage token movement. With this technique, very complex flow patterns can be reproduced relatively easily. Splits and joins are easy to manage regardless of any outside constraints on token movement. Even multiple instance patterns can be reproduced with the introduction of new tokens into a given activity. Extending Petri-nets to use color and time further add to the power of this modeling language to express complex patterns.

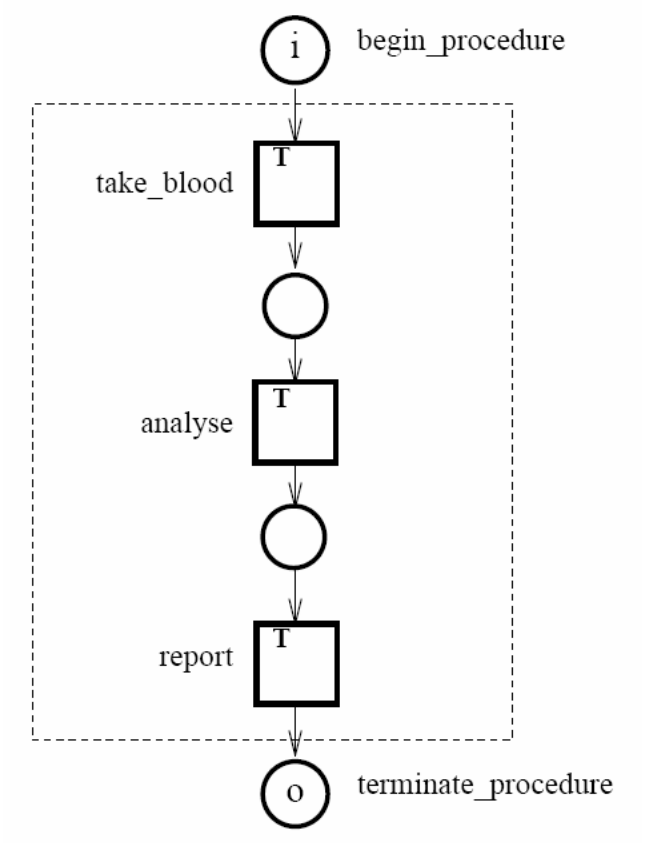


Figure 2.8 Petri-net based Workflow[64]

Petri-nets are considered a high-level modeling tool and are generally used for modeling processes that have little or no data interdependencies. This creates difficulties when modeling workflow systems which tend to have a many data constraints. In addition, this approach is still essentially activity-based and thus suffers from the same drawbacks as current mainstream activity-based modeling approaches. However, the power of Petri-nets to model complex flows makes this an approach that requires serious consideration when developing new techniques.

2.5.4 YAWL – Yet Another Workflow Language

This approach starts with the use of Petri-nets and attempts to develop a new language which can express all of the currently identified patterns encountered in workflow modeling. This approach supports all but one of the workflow patterns, is easy to understand and has a formal semantic. YAWL successfully preserves the power of Petri-nets to describe process and provides a straightforward way of expressing some complex patterns in a simpler notation than that of Petri-nets. The symbols offered in this modeling language are very easy to understand and offer the best usability of all the approaches surveyed in this paper.

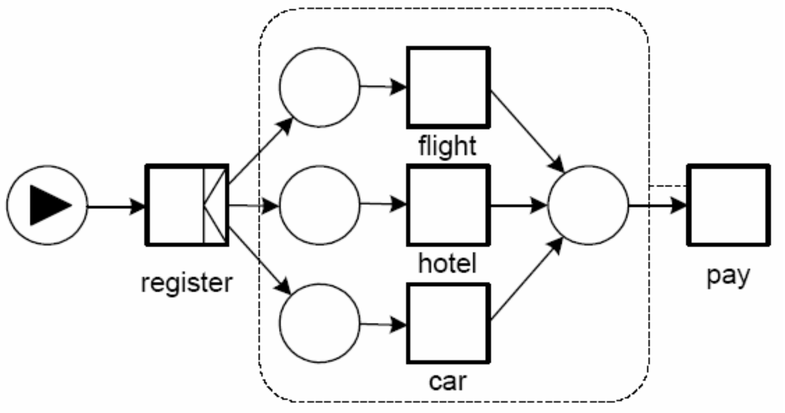


Figure 2.9 YAWL Diagram

2.5.5 Object-Process Methodology

One of the best single-diagram methodologies is called the object-process methodology[19, 65]. This notation mixes the OO-based class diagram notation with the processes that change their state. Thus, objects interact with processes, while special notation describes how these objects change state as a result of interaction. Figure 2.10 shows an example object-process model that demonstrates some of the finer features of this notation. The circle in the center represents a process that has been expanded to show the details within it.

This hierarchical structure allows for hiding of unneeded complexity while allowing for detailed specification.

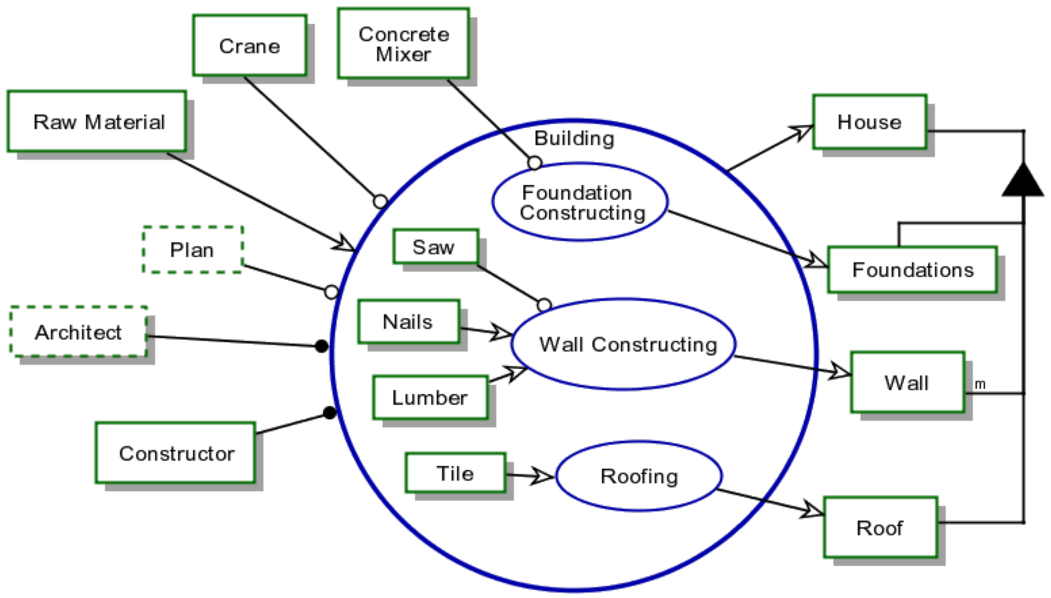


Figure 2.10 Object-Process Model

Notice that composition, inheritance and other OO design patterns can be easily represented in this notation. This is, by far, the most complete unified modeling technique [66]. The interactions between process and objects are intuitive and simple. The object-process model does not have a formal semantic.

2.5.6 Object Petri-Nets

Object Petri-Nets [15, 67-69] (OPN) are currently the best solution for providing a concurrent, object-oriented language with a formal semantic while providing high usability. As such, OPNs demand a very detailed analysis of their capabilities and liabilities in order to demonstrate the advantages of FOIL.

Petri-nets [6], on the other hand, exhibit many strengths lacking in UML. The Petri-net easily models complex concurrent behavior and has an underlying mathematical foundation.

The notation has a small symbol set and is relatively easy to comprehend. These characteristics have made the Petri-net diagram a proven, time-tested modeling notation. The success of the Petri-net made it a suitable launching point for an OO modeling language. Colored Petri-nets (CPN) were introduced [53] to blend the process interaction capabilities of Petri-nets with the data capabilities of high-level programming languages. This was shortly followed by adding hierarchical support to CPNs (HCPN) [52]. Recent improvements include the adaptation of HCPNs for OO design [70] or extension of HCPNs to a fully specified OO language called the Object Petri-net (OPN) [15, 67, 68].

Object Petri-nets provide support for hierarchy and inheritance by allowing a class to be the token of another OPN class. The outside process model controls the flow of tokens (objects) through a common message processing interface. The internal life-cycle of objects is represented using a finite state machine (FSM) that responds to the same messages as the encompassing Petri-net model. Through the use of super-places and super-transitions, a great deal of flexibility has been added to the language. A thorough survey suggests that the OPN is the best attempt to date for providing a concurrent OO modeling language with formal verification and has been shown to be effective in modeling real world problems [71-73].

The problems with OPN mostly arise from its roots as a process language rather than an object-oriented one. While OPN models can be reduced to simple UML class diagrams from a structural point of view, the behavioral nature of inheritance is not fully addressed. The formal framework for OPN applies to objects that are already instantiated not to the instantiation process itself. In the literature for OPNs, instantiation is assumed but not explicitly modeled.

In addition, OPN requires, in many cases, that objects perform functions that are not natural in an OO methodology or that overarching objects be added to perform these processing functions. If one supposes that a major benefit of OO design is modeling software that is mapped onto the real world, then such object extensions should be avoided. A primary example of this can be found in [68] where the Table object is charged with determining if a dining philosopher problem is deadlocked. In the real world, tables do not do much of anything. The position of this thesis is that in OO design, objects, not processes, should interact with one another to perform work.

Finally, CPNs have thorough support for concurrency but the OPN methodology assumes an FSM for the object life-cycle and thus “concurrency within an object is not considered” [68]. This is unfortunate, as real world modeling might require that such support be present. For instance, in the classic dining philosopher problem, it is generally assumed that a philosopher will pick up the left chopstick and then the right, but in reality they would likely pick up both concurrently. One could model each *Hand* of a *Philosopher* to achieve such concurrency in OPN but this is an unnecessary abstraction which adds complexity to the model.

2.6 Conclusion

Of all the modeling languages available today, most of them do not support even three of the main design goals outlined in this thesis. None of the modeling languages surveyed successfully implemented all of them. By far, the most complete framework allowing for modeling of structure and behavior, a formal semantic, and concurrency support is the Object Petri-net (OPN). But, as provided by the literature, OPN does not support direct process modeling and has no mechanism to verify proper process operation. OPNs have a few other problems: they deviate from the ‘real-world’ character of object-orientation; do not account for lifecycle concurrency; do not consider object instantiation; and can quickly become very complex because of the way objects are extended as tokens or places. Overall, FOIL provides a modeling framework that can meet all of the design goals, including process modeling and verification, while maintaining a well-known object-oriented nature.

This rest of this thesis is organized as follows: Chapter 3: Introduction to the graphical elements that make up FOIL; Chapter 4: Introduction to the FOIL algebraic representation and the laws and identities that provide for mathematical manipulation; Chapter 5: Explanation and examples of behavioral inheritance, concurrency modeling and model verification; Chapter 6: Demonstration of how FOIL can be used to model all known workflow patterns; Chapter 7: Detailed explanation of how FOIL can be used to determine the ability of a process to accomplish its work, given a FOIL object model; and Chapter 8: Discussion of FOIL’s benefits and limitations as well as direction for future research.

3. FOIL NOTATION

Formal Object Interaction Language (FOIL) provides a diagrammatic notation designed to leverage what is good about the class diagram and provide more information about the behavior of objects after instantiation. Important extensions such as *Ports* are made to model concurrency aspects of an object's behavior. Also, FOIL explicitly models an object's event firing, and uses an event mechanism to expressly show the relationship between multiple objects' communications and individual objects' behaviors. Such relationships are implicit in UML and have to be deduced by designers from multiple diagrams.

This chapter informally presents the diagrammatic notations of the major components of FOIL. A formal representation of FOIL modeling, especially concurrency modeling, is provided in chapter 4.

3.1 Behavioral Representation

One of the key features of FOIL is its constraint on the behavior of objects. Current software modeling techniques focus almost exclusively on the structure or interface of an object, but not on the behavioral aspects. While state charts and other devices work to give developers an idea of what the behavior of an object should look like, they little information as to what behavioral constraints should be applied to an object. Additionally, inheritance of objects does not extend to the behavior [74]. FOIL does both in a single notation, such that inherited objects are modeled to perform their interface conforming methods in an consistent manner.

3.1.1 States

Much like state diagrams, FOIL uses *states* to represent the status or stages in the behavior pattern of an object. FOIL differentiates between different types of states (i.e., between active and passive states, and between accepting and non-accepting states. Such differentiations represented by diagrammatic notations and captured by FOIL algebra, are necessary to increase the expressive power of behavior modeling. Meanwhile, the state of an object in FOIL can be complicated since FOIL allows for an object to be in multiple states simultaneously in much the same way as *non-deterministic finite automata*. Figure 3.1 shows three different notational element combinations used to indicate the state of an instantiated object.

A state can be perceived as both an attribute and a method. It functions as an attribute in that it indicates a quality of the object's temporal nature. It functions as a method in that, upon arrival at a state, it may perform a manipulation of the object or system. States arrived at concurrently are assumed to execute their actions in a random order (see 3.1.3). This should be considered when modeling a software system as there are ways to ensure that states execute in a specified order by modeling them sequentially (see Firing). All state execution methods are considered to be protected and cannot be executed from outside the instantiated object or one of its children.

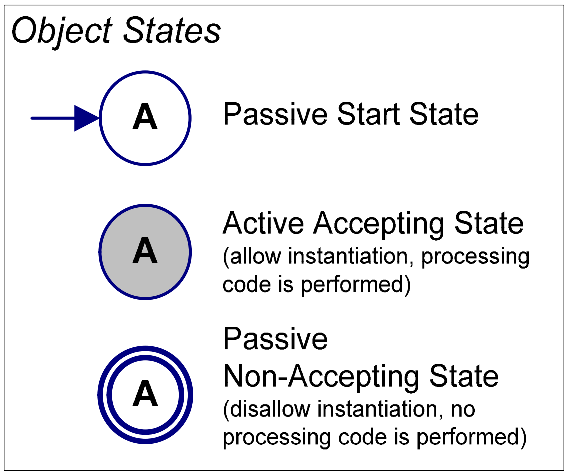

Object States
A
Passive Start State
A
Active Accepting State
(allow instantiation, processing code is performed)
A
Passive Non-Accepting State
(disallow instantiation, no processing code is performed)

Figure 3.1 FOIL Object States

An active state is one that performs an unspecified action upon arriving and is represented by shading the state grey. This action will always take place after pre-firing events (see Firing). A passive state, indicated without shading, acts more like an attribute in that it merely indicates the status of the object and does not do any real work. For an active state, its associated action can modify the specifics of any post-firing events including canceling the event firing; however, it can never choose to post-fire a different event as this would undermine the formal nature of the notation.

An accepting state is denoted by a single circle and indicates that this object may instantiate new objects if requested. This only holds true if an object instantiation transition exists for that object (see 3.1.2). If an event is received that requires an object to create a new instance of a class, the object must be in an accepting state in order to “accept” the event. A non-accepting state is the converse of the accepting state in that any event received that would normally instantiate a new object is “not eligible”. A state may be accepting or non-accepting independent of whether it is active or passive.

The start state is the initial state of an object after instantiation. The start state is denoted by a black arrow with a start point outside the class definition and pointing to the state. Thus, an active start state can be viewed as a constructor while a passive start state would be analogous to an empty or default constructor. The final state is implicit and need not be explicitly drawn by the modeler. The final state indicates that after completion of the state execution the object has nothing left to do. It is important to consider that some objects may not have a final state as they may perpetually loop through states throughout the execution of the system. Since multiple concurrent threads of execution can exist in a model, the completion of a final state does not necessarily mean that the object is finished, since other threads may still be in progress.

3.1.2 Transitions

Transitions are the primary means of modeling the behavior of objects. A transition represents a progression from one state to another and is triggered by an event that is either internal or external to the object. In this way, transitions are the behavioral constraints placed on an object. Different from transitions used in traditional state diagrams, the execution of a transition depends not only on the triggering event, but also on the event's eligibility determined by the object's state. This eligibility can be checked using FOIL algebra and is enforced during runtime. This extra "eligibility checking" is important in modeling asynchronous and concurrent behaviors of objects.

Figure 3.2 shows the various notational elements used to represent object transitions. Transitions are always represented by a directional arrow labeled with the name of the event which may cause state change. The passing of data as part of the event mechanism may be additionally specified with parameters.

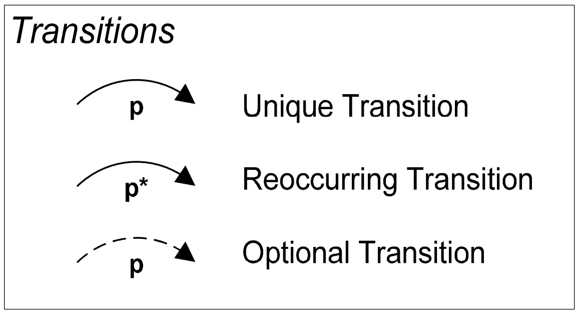


Figure 3.2 FOIL Transitions

A unique transition is one where the target object only expects to receive the event once in a given iteration. Therefore, a looping construct is not limited by the use of unique events. The specification of the iterative uniqueness of an event is an important aspect of the modeling language as it allows the FOIL algebra to enforce rules about the acceptability of an event based on its possible reception in the future. If the system is aware that an event will only occur once per iteration, the system may refuse to accept an occurrence of that event because another object that requires it is not ready to receive it. A reoccurring transition is used to indicate that the number of times this event will be received is indeterminate.

An optional transition is used to show that this transition may or may not occur. Thus, two optional transitions from a single state would need output ports in order for both options to be available (see Ports). An option, which has not been taken, remains available in the model until such time as the object flow invalidates that possibility. For example, if an optional parallel split was modeled but only one option had been taken, the second option would remain available unless the merge point for the two threads is passed by the first option. Thus, the second option would be invalidated since that thread could never be merged.

An object instantiation is represented by a standard UML relationship notated with an event. This notation is used to represent the creation of an object by the occurrence of an event. This also indicates a relationship between two objects as the source object of the arrow represents the object "responsible" for its instantiation. Object instantiation can only occur if the "responsible" object is in an accepting-state (see 3.1.1).

3.1.3 Ports

Ports are used to model concurrency, both asynchronous and synchronous. Figure 3.3 shows the notation for the types of ports. Ports may contain numbers within them to indicate a quantity. An empty port is assumed to have a quantity of one. There are two basic types of ports: input and output.

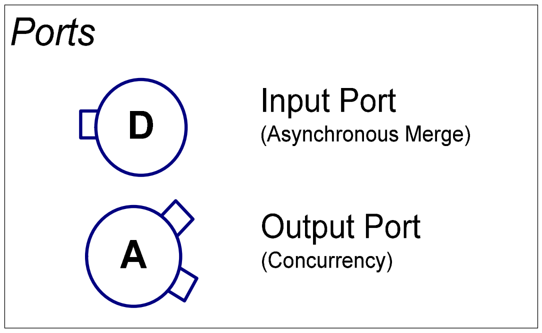


Figure 3.3 FOIL Transition Ports

The output port indicates the number of threads of execution required to leave an object before the object is no longer in that state, which creates a parallel split. In Figure 3.3, the output ports indicate that there are two transitions required out of state *A* in order for the object to be considered NOT in state *A*. Extra threads of execution are implicitly created as a result of output ports. Once the object's state transitions out of a port it must create a new thread in order to remain available for the other output ports.

The input port indicates the number of threads of execution required into a state in order to allow the object to transition out of that state. For example, in Figure 3.3 the input port on state *D* means that the object's internal workflow could proceed beyond state *D* when a single thread has transitioned to it. This is only meaningful when multiple threads are expected such as in a parallel split situation. Multiple threads of execution may be merged without the use of an input port; however, such merging will always be synchronous. Input ports are mainly used to allow for asynchronous merging of parallel threads of execution.

3.1.4 Firing

Error! Reference source not found. Figure 3.4 shows the various event firing notations. So far, the interactions between objects have been modeled through the fact that independent objects react to the same events and that some objects can instantiate others. This is not sufficient to handle all event patterns and can result in a model that is difficult to understand. In order to alleviate this problem the idea that an object itself fires events is required.

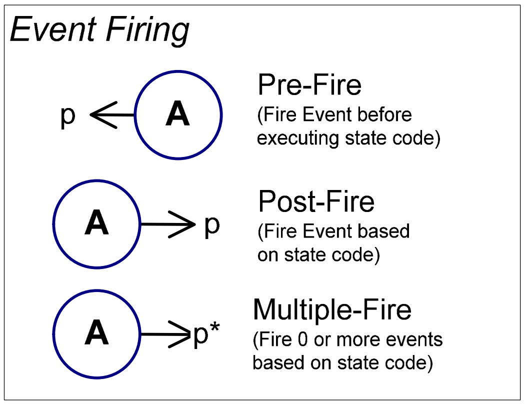

Event Firing
p
A
Pre-Fire
(Fire Event before executing state code)
A
p
Post-Fire
(Fire Event based on state code)
A
p*
Multiple-Fire
(Fire 0 or more events based on state code)

Figure 3.4 FOIL Event Firing

Pre-firing causes an event to be triggered prior to executing actions required by the target state. In practice, states may have code which they execute as a result of a transition to them. The pre-fire ensures that an event is triggered prior to executing that code. Post-firing is similar, but occurs after executing the state code. Finally, multi-firing is a post-fire that allows multiple instances of an event to be fired. The determination of how many events to fire occurs within the state code at run-time.

3.1.5 Interleaving

The final notational element is interleaving. Interleaving requires that an object exhibit multiple behaviors sequentially but in no specified order. The notation of a dotted box is used to indicate that the items in the box should be interleaved. This notation is provided in FOIL for purposes of usability. Since interleaved execution can be modeled as a choice among multiple sequential possibilities, this pattern can be modeled using the notational elements previously outlined. However, this would be, in the best case, cumbersome and, in the worse case, completely unreasonable. This is because the number of combinations per sequential choice added to the model would grow excessively fast (on the order of n!). Thus, this notation provides a means to model such cases while avoiding this state explosion problem. How state explosion is handled in FOIL algebra will be covered in chapter 4.

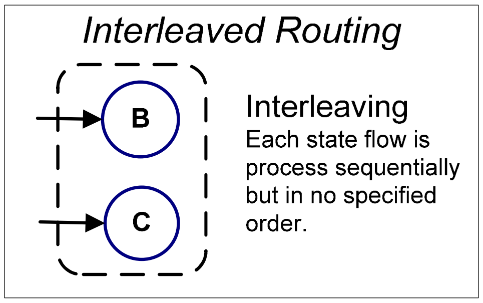

Interleaved Routing
B
C
Interleaving
Each state flow is process sequentially but in no specified order.

Figure 3.5 Interleaved State Routing

3.1.6 Event Scope

Events in FOIL cause objects to enact their behaviors; however, what if the intent is to enact the behavior in a specific object. In FOIL, this is accomplished through a mechanism referred to as event scope. When an event is fired, it may be annotated with the object or objects for which it applies. Since each object determines its own reaction to an event, the presence of such annotations would cause the object to ensure that it was in the list before reacting to the event. Likewise, the absence of such annotations would ensure that an object will always react to the event. Typically these annotations are only shown when their presence is of significance to overall system operation.

3.2 Object Modeling

Given the notation for behavior specification in FOIL, the definition of an object class can be modeled that accounts for its structure, as in traditional modeling techniques, but also constrains to its behavior. Since state attributes and method calls have more to do with an object's behavior than its structure, the text representation of an object's structure need not explicitly define these. This lends itself to a more graphical representation of an object with fewer low detail text elements.

3.2.1 Basic Object

Representing an object with the Formal Object Interaction Language (FOIL) is relatively easy. Using the notational elements outlined above, each object is represented by its attributes, method and behavior as shown in Figure 3.6. In this example, a *Quote* object is defined. The *Quote* starts life in the *Open* state and either transitions to *Expired* or *Ordered* depending on the input event. The shading on the *Open* and *Ordered* states indicate that they are active and thus will perform processing upon the object arriving at the state. The *Expired* state does not execute any actions.

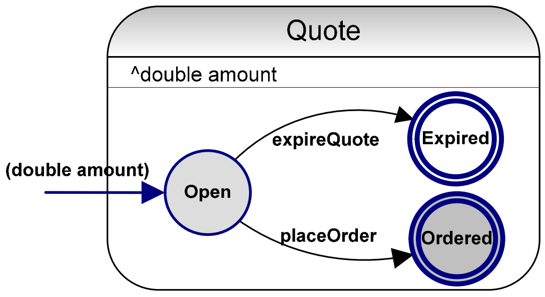


Figure 3.6 Basic Quote Object in FOIL

Attribute representation is abbreviated in FOIL, as with the Business Object Notation (BON) [75], to reduce the number of specifically defined methods. Since behavior aspects of an object are clearly defined by the notational elements, most of the remaining methods involve the storage and access of data.

The read-only attribute qualifier (^) is also shown in Figure 5. The *amount* would be set by object instantiation as indicated by the input parameter for the start state. It may be required that the *amount* value be retrievable from outside the object. A class diagram would represent this as a private attribute with an accessor method provided. Methods in FOIL can still be specified in the typical manner.

3.2.2 Instantiation

Relationships between classes are shown in the same way as in the UML class diagram. Thus, FOIL conforms to the traditional forms of object relationship: aggregation, composition, association, and generalization.

Instantiation of objects of one class by another is indicated by using the association symbol offered in traditional UML class diagrams with an added event notation. This means that an association that does not have an event is treated as knowledge of one object by the other. From a FOIL point of view, this represents a possible communication channel (see Communication). An association with an added event qualifier indicates that an object of the class will be instantiated when the event is received and the source object is in an accepting state.

The fact that an object must be in an accepting state is a significant difference between FOIL and the other attempts at hybrid notations. Rather than just the behavior of a single object being represented, FOIL offers the ability to see how objects are created and what rules are required for such creation in a graphical way. Previous hybrid object-oriented (OO) notations neglected the graphical representation of instantiation rules and thus made it difficult to see the process overriding the behavior of individual classes.

Figure 3.7 shows an example of object instantiation. In this example, the attributes associated with the *Account* class are omitted for brevity. It is clear from the notation that a *Payment* object can only be created if the account is in the *Active* or *Overdue* state. Note that the asterisk (*) on the association indicates that more than one *receivedPayment* event is expected. This could also be done with multiplicity values for the relationship.

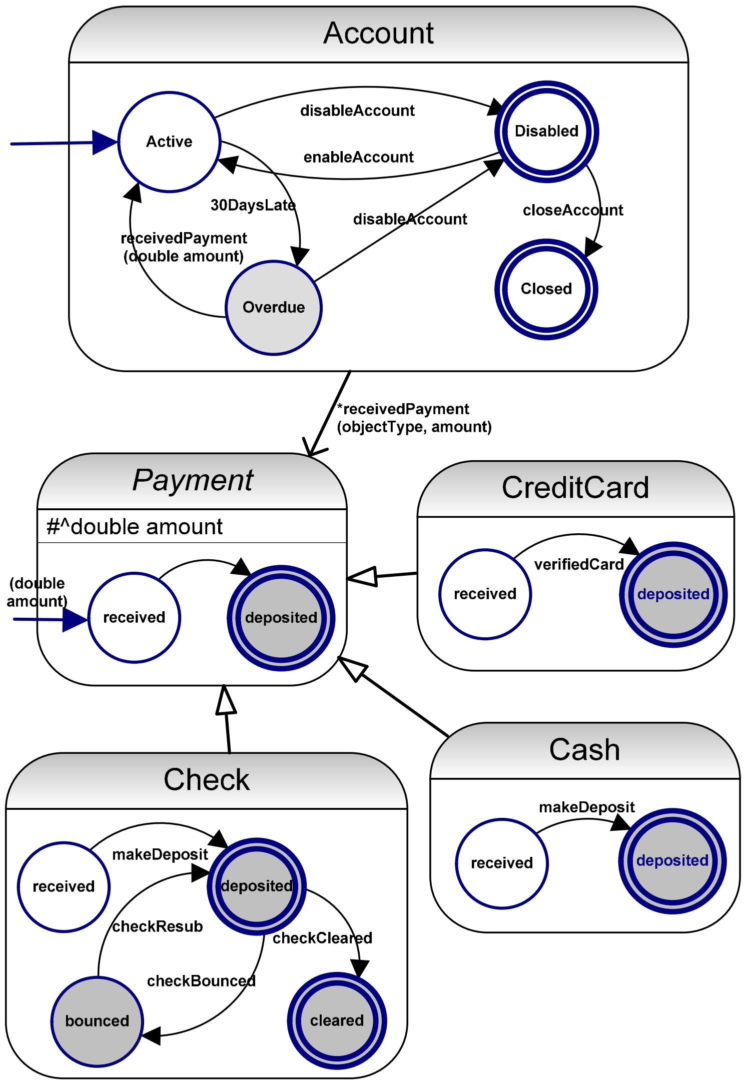


Figure 3.7 Basic Payment Process in FOIL

3.2.3 Inheritance

One of the hallmarks of object-oriented development is the concept of object inheritance. This inheritance is informally referred to as an "is a" relationship. Thus, if a class *Salmon* inherits from class *Fish*, it is because a Salmon "is a" Fish.

The main problem with inheritance as implemented in common modeling frameworks and programming languages is that it is solely concerned with structural conformity. It might be said that if class *Salmon* or *Trout* look like a *Fish* and acts like a *Fish* then it "is a" *Fish*. But, if inheritance only ensures structural conformity by definition a child's wind-up fish toy (class *WindupFish*) could actually be a *Fish*. Indeed, it looks like a *Fish* (attributes: fins, tail, etc.) and acts like a *Fish* (methods: swim, catch, etc.). But *WindupFish* is not a *Fish* primarily because the way in which it implements its methods is decidedly different.

FOIL reintroduces the concept of "behavioral inheritance" [74] where inheritance is defined by the structural and behavioral conformity of an object. Since, FOIL allows for the detailed modeling of the behavior of individual objects, it can be determined if the behavior of one class represents a subset of behavior of another. The formal details of how this works will be explained in section 5.2. Therefore, *WindupFish* class could not extend from *Fish* since the internal behavior of *Fish* would not be a subset of the behavior of *WindupFish*. On the other hand, a *Salmon* could definitely inherit from *Fish*. While the nuances of how a *Salmon* and a *Trout* swim could differ slightly; in general, the mechanism for swimming in a *Trout* and a *Salmon* are fundamentally the same because they both look, act and function internally like a *Fish*.

An example of inheritance in FOIL is represented in Figure 3.7. In this case, the *Payment* class is abstract but defines that every payment should have two states and should accept an amount for instantiation. The *deposited* state is active and thus performs some action, but this action is not abstract. The abstract nature of this class is that the transition between the *received* and *deposited* states is undefined. By modeling the abstract class, the designer is stating that there are two states, *received* and *deposited*, and there is a transition between them. The concrete details are left to the subclass. Thus, each subclass must have these two states and must have a transition between them.

3.2.4 Communication

FOIL can be used to model distributed systems with a centralized event manager. This does not change the fact that communication between classes must be done through defined relationships. In an object-oriented environment, communication between objects occurs when an event is fired by one object and received by another. This is analogous to a method call.

Figure 3.8 shows an example of a communication sent by the *Elevator* object to the *Door*. The *reachedFloor* event is propagated down the composition relationship. A light dotted line can be used to indicate the relationships that an event uses for communication. This example can therefore be interpreted to mean that the *Elevator* object calls the *reachedFloor* method of the *Door* object. The dotted line connecting a firing with a relationship is optional but is helpful in correlating events and affected transitions.

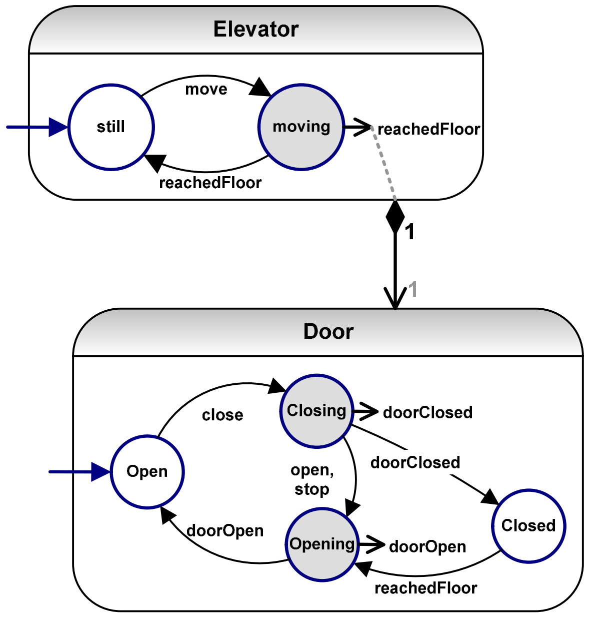


Figure 3.8 Simple Object Communication

3.3 Process Modeling

FOIL not only allows for the modeling of objects and their behavior, but it also provides a simple notation similar to the object notation for modeling high-level processes as well. A FOIL process can be checked, using FOIL algebra, against the object model to determine whether the given object model will, in fact, perform the defined process. The exact details of how this is accomplished described in chapter 7.

3.3.1 Process as Object

FOIL takes the approach that a process is an object of an abstract process engine. Since objects can be modeled with arbitrary levels of abstractions, it is reasonable that a process is an abstraction of a process execution engine. However, this means that a process in FOIL exists outside of the main object model and thus does not behave exactly like what would be expected of a modeled object. FOIL considers it important that, in a "pure" object-oriented framework, only objects in the model perform real work. The entire execution of process in FOIL is performed by the objects and their corresponding communications with each other and are moderated and controlled completely by the algebraic expressions they represent. The concept of objects "performing" process rather than process "using" or "regulating" objects, while not unique to FOIL, is an underlying principal of the language.

In order to maintain this fundamental nature of objects in FOIL, a FOIL process must comply with the following rules:

1. States in a FOIL process cannot perform work. Instead, active states in a process model represent a sub-process.
2. States in a FOIL process do not correspond to states in the object model since they are part of a totally different system: the process engine.

3. All Events both fired and received must exist in the object model. Object scope qualifiers in the process model may be used.

If these rules are followed, an object modeled can be checked to ensure that it will execute a given process. Figure 3.9 shows an example of a FOIL process *P* that can be used to verify that the system modeled by objects *X* and *Y* will perform work as expected. The firing of event *p* guarantees that object *Y* will be instantiated. Once this occurs, unique event *s* can not be accepted by the system until object *Y* transitions to state *H*. Thus, event *q* must always be received first, after *p* is fired, but before *r* is fired and *s* is received. This analysis clearly demonstrates that process *P* can be accomplished with this object system.

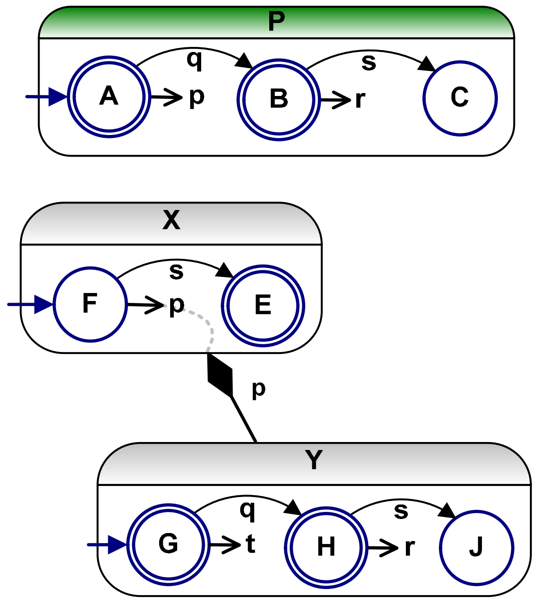


Figure 3.9 Simple FOIL Process Model

3.3.2 Process Nesting

Processes in FOIL can be arbitrarily nested. In FOIL, process nesting refers to the sequential replacement of a process state by another FOIL process. The notation for this nesting is done by marking a state in the process as active. An active state in a FOIL process diagram, as mentioned earlier, represents a sub-process. The term “active” here refers to the fact that another activity must be performed before this process may continue.

Figure 3.10 shows an example of how process nesting is represented in FOIL. Process P_1 has *B* marked as an active state. Thus, when arriving at state *B,* the process *B* will be verified with the object model by straight sequential substitution. Process P_2 is logically equivalent.

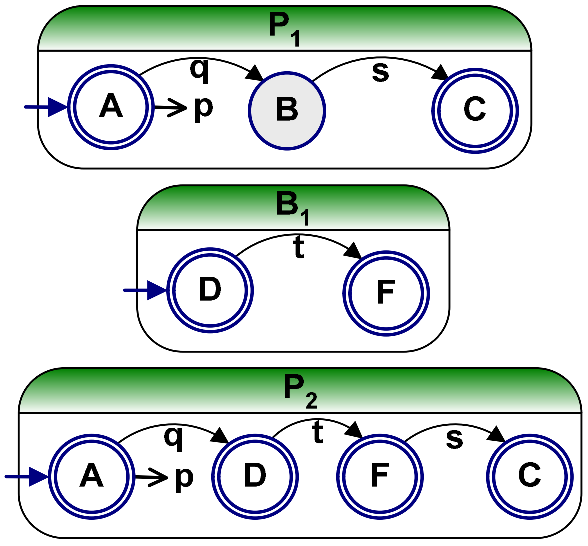


Figure 3.10 Process Nesting Equivalence

3.3.3 Process Spawning

In addition to nesting processes, FOIL supports the concept of process spawning. A process is spawned when an event occurs that will cause a new process to start. These two processes (the calling process and the new process) will then continue concurrently. Since a FOIL process model is primarily used for verification, this spawning allows objects to perhaps communicate in different ways while still performed their core process. Thus, concurrency in process modeling allows for more flexibility in the object model.

Figure 3.11 shows an example of how process spawning is modeled in FOIL. When process P_3 transitions to state *B* an *r* event is triggered. This event causes the creation of process B_3 which will continue concurrently with process P_3. At this point, either an *s* or a *t* event would be valid allowing the underlying object model to fire

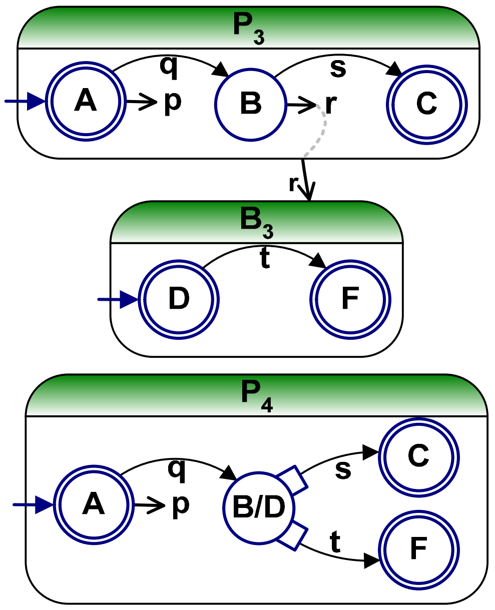


Figure 3.11 Process Spawning Equivalence

or receive these events in an arbitrary order. This is not true of object P_1 in Figure 3.10. In general, process spawning is a much looser validation of the underlying object model than process nesting. Process P_4 in Figure 3.11 is logically equivalent to the process system create by the interaction of process P_3 and process B_3.

3.4 Simple Elevator System

A simple elevator system is modeled below in both the Formal Object Interaction Language (FOIL) and the Unified Modeling Language (UML). As UML is both popular and familiar, this should aid in understanding the distinctive qualities of the FOIL model.

Figure 3.12 shows a simple elevator system as modeled with the Formal Object Interaction Language (FOIL). The relationships used in the model are the same as those used in the standard UML class diagram. However, in UML the communication between objects as a result of these relationships is unclear. The FOIL notation makes the communication requirements clear. This is an example of the behavioral information implicit in a FOIL model. Notice in Figure 3.12 how the elevator *Door* can be stopped by a *Passenger*, resulting in the *Door* reopening. The loop in the *ElevatorController* causes the *Door* to attempt to close again.

The *MasterController* in this diagram shows how concurrency is modeled. In this case, the master controller is a continuous listening object that will spawn a new thread of execution for every request received by the buttons. The next available *ElevatorController* sends a *nextFloor* event to the *MasterController*. The logic for which floor the controller will dispatch the elevator is determined by the active state *Queued*. In object-oriented implementation, the *go* event is really a method call that has a *floor* parameter. This is optional in the FOIL notation but is shown in the *go* event definition in the *ElevatorController* object.

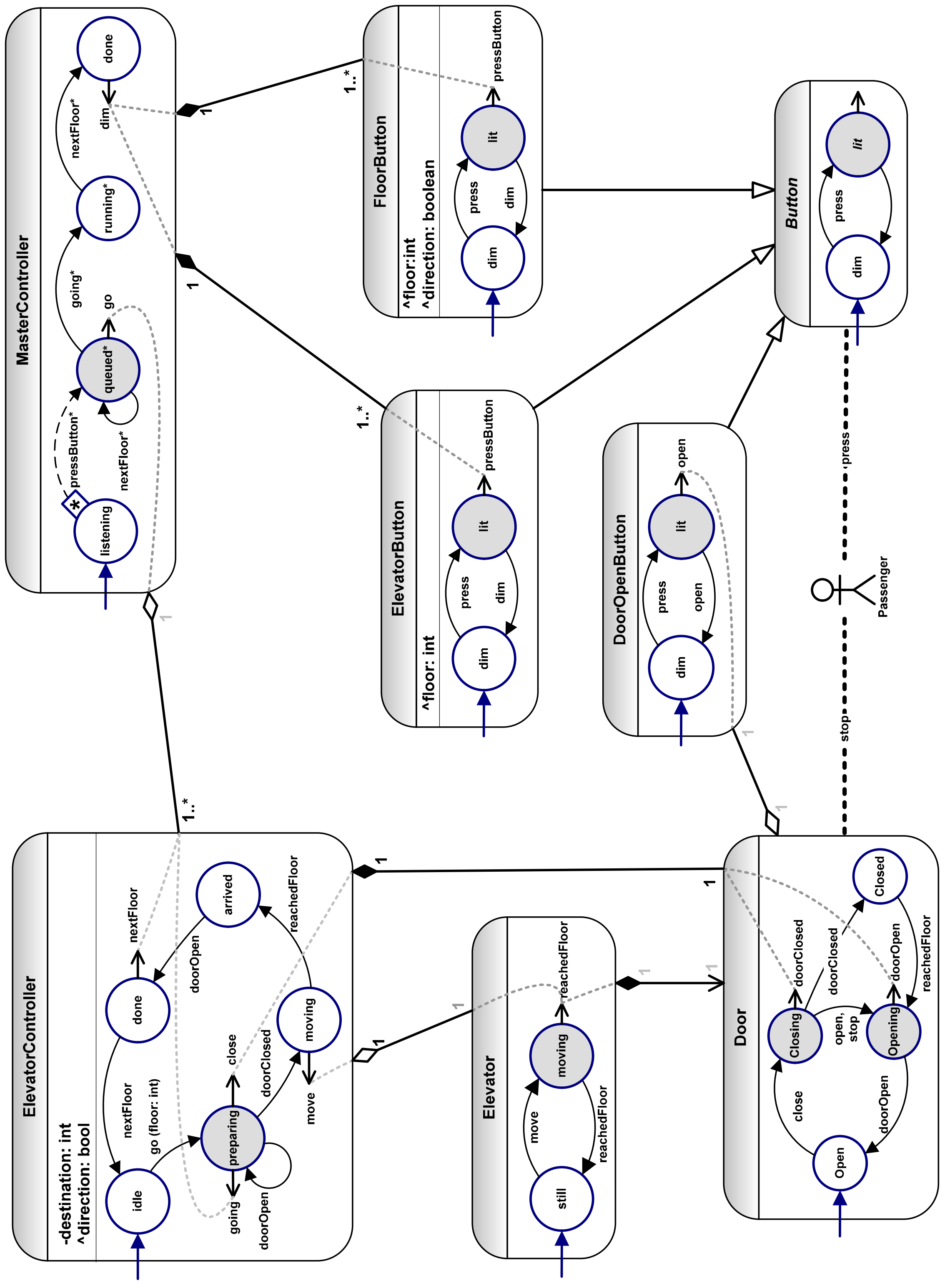
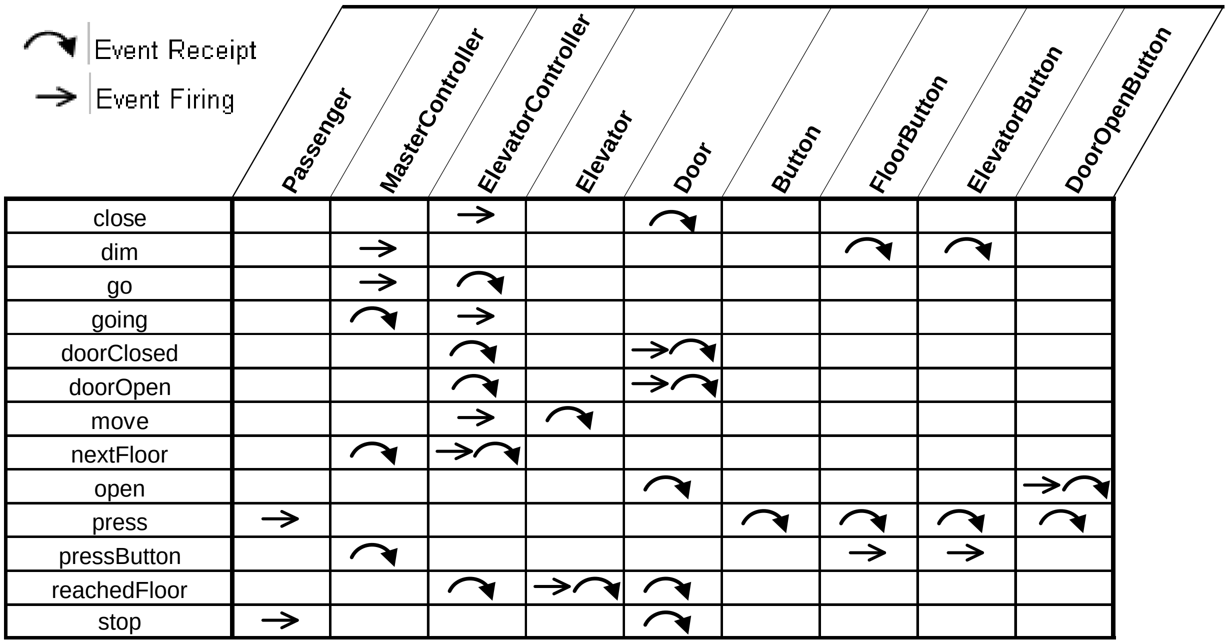


Figure 3.12 FOIL Elevator Example

The FOIL notation supports all of the relationships in the UML class diagram. Inheritance is extended to include "behavioral" inheritance, as can be shown by the abstract *Button* class. In this case, a button has a *dim* and a *lit* state and the *press* event will always cause transition from *dim* to *lit* regardless of the type of button. The implementation of the active state *lit* is not specified and must be implemented by the subclasses of *Button*. This is denoted by the *lit* state in italics.



↷	Event Receipt	
→	Event Firing	

	Passenger	MasterController	ElevatorController	Elevator	Door	Button	FloorButton	ElevatorButton	DoorOpenButton
close			→		↷				
dim		→					↷	↷	
go		→	↷						
going		↷	→						
doorClosed			↷		→↷				
doorOpen			↷		→↷				
move			→	↷					
nextFloor		↷	→↷						
open					↷				→↷
press	→					↷	↷	↷	↷
pressButton		↷					→	→	
reachedFloor			↷	→↷	↷				
stop	→				↷				

Figure 3.13 FOIL Diagram Event-Object Schedule

The FOIL model can also be augmented with a reference help called the Event-Object Schedule. Figure 3.13 shows the schedule for the elevator example in Figure 3.12. In this schedule, straight arrows indicate that the event is fired and the curved arrows indicate that the object accepts that event. This is a beneficial reference when trying to determine which objects are involved in event production and reaction.

Finally, Figure 3.14 shows the FOIL process diagram for the elevator model in Figure 3.12. This process model is composed of two processes that run concurrently: *Pick up Passenger* and *Drop off Passenger*. Each one is triggered by the *Passenger* pressing the appropriate button. Every time a Passenger presses a *FloorButton*, a new *Pick Up Passenger* process is created. The *Drop off Passenger* process is created whenever a *Passenger* presses an *ElevatorButton* and the *Pick up Passenger* process is in the *Loading* state. It should be relatively easy to see that the process as modeled in Figure 3.14 can be accomplished by the FOIL object model previously given in Figure 3.12.

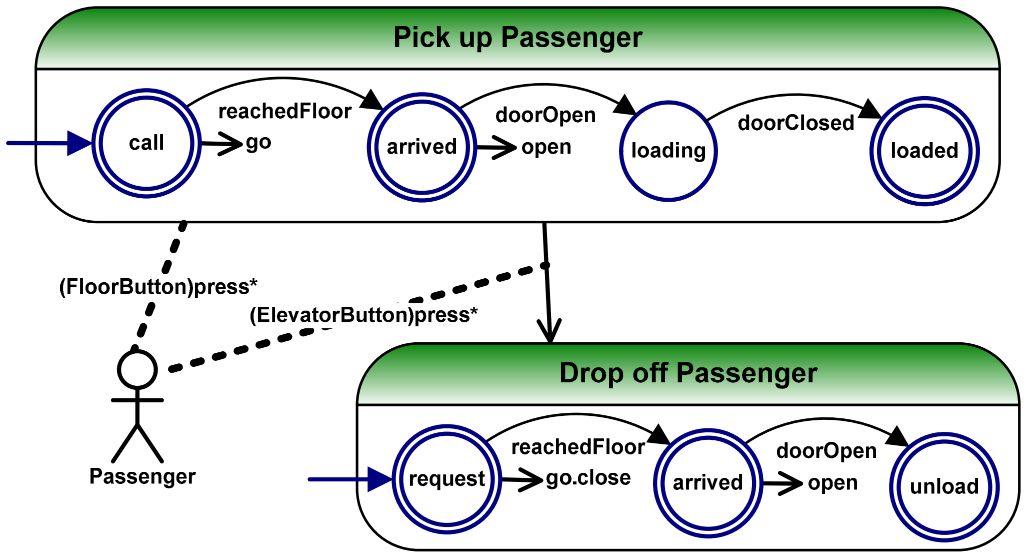


Figure 3.14 FOIL Elevator Process

3.4.1 UML Equivalent

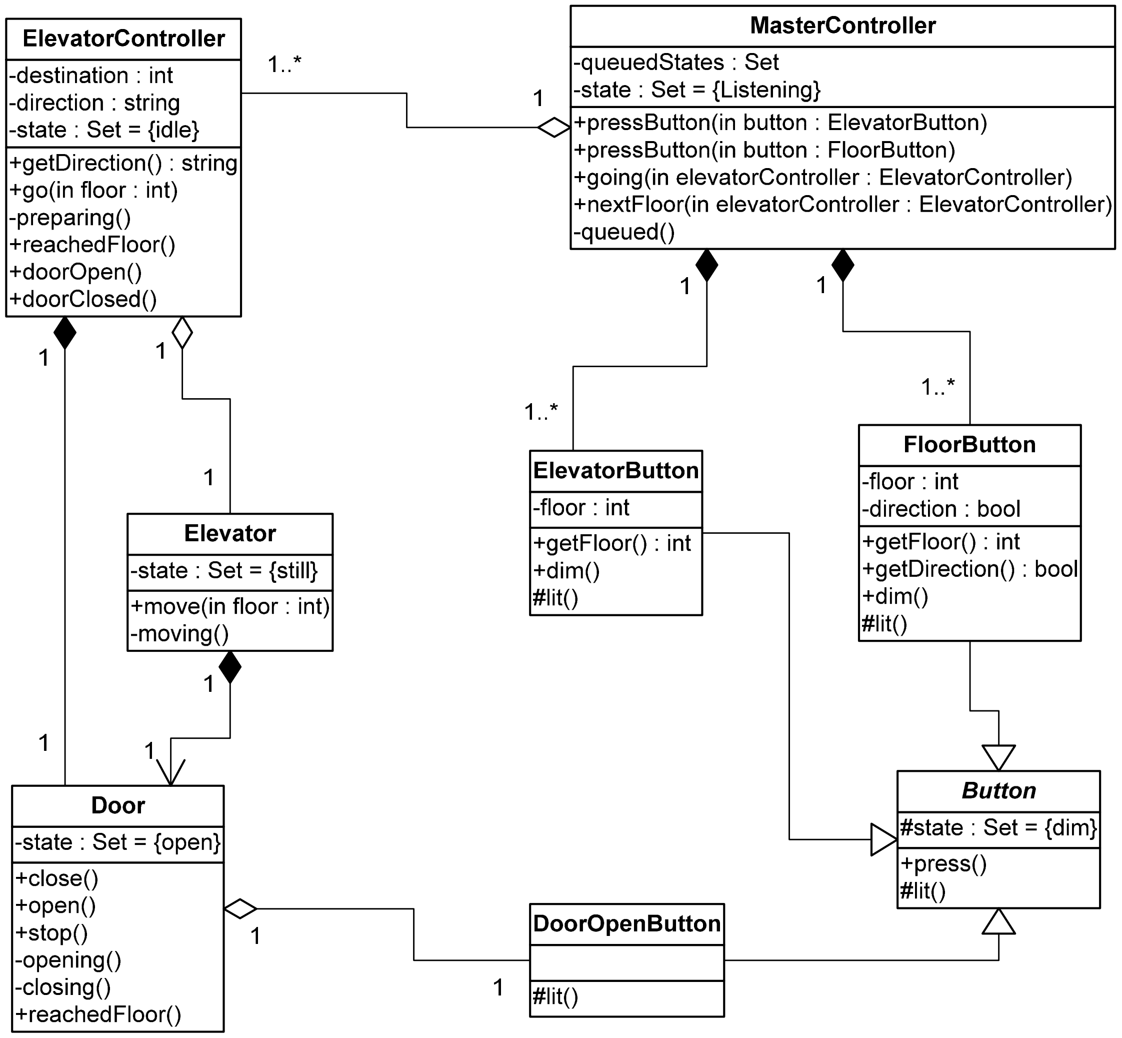


Figure 3.15 UML Class Diagram of Elevator

A simple FOIL model can be converted to a standard UML model. As more of the concurrency features of FOIL are used, the converted UML model becomes quite large as individual thread of execution must be explicitly modeled in UML. Figure 3.15 shows the UML class diagram of the equivalent model from Figure 3.12. Each active state in the FOIL model becomes a private method in a standard UML model. Likewise, any event which can be received becomes a public method. Read-only attributes are converted to private attributes with an appropriate *accessor* method. Public attributes – of which there are none in this model – can be converted as standard public attributes or private attributes with the appropriate get/set methods.

The detail in the equivalent class diagram is far less than the FOIL model as there is no indication as to the behavior of individual methods nor is there any indication as to how the objects interact. In UML this requires a separate diagram, of which there are several varieties. Figure 3.16 represents an equivalent sequence diagram for standard elevator operation as modeled by the FOIL diagram in Figure 3.12. The diagram in Figure 3.16 models an expected operation of a single elevator.

It should be noted that a sequence diagram is rarely suitable for specifying multiple scenarios. Modeling of the behavior of the *MasterController* or the scenario of an elevator door impediment would each require an additional diagram. Even after diagramming each scenario, additional UML state or collaboration diagrams would be required to specify the complete interaction between objects. Thus, this simple system would require approximately eight diagrams to display the same information as contained in the single FOIL model.

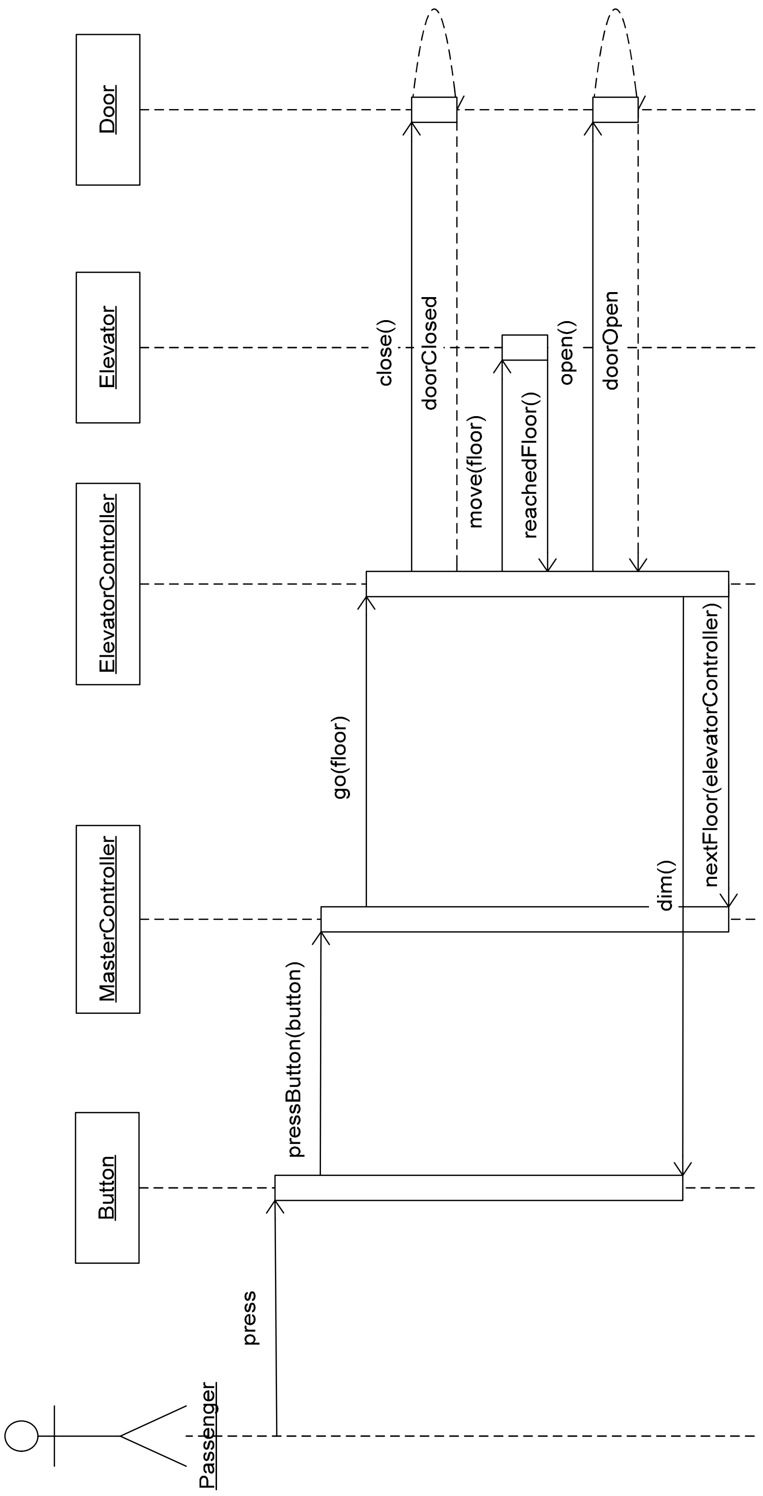


Figure 3.16 UML Sequence Diagram Equivalent

A UML activity diagram for this elevator example is hardly worth modeling. The process performed by an elevator is extremely simple and thus a UML activity diagram would consist of two boxes with a line between them. Additional notations may be made to the diagram. The FOIL process diagram actually presents the expected sequence of events when in operation. This could be done by creating a UML activity diagram with a very low process granularity where, for example, the door closing and opening would each be considered activities. In addition, the low level nature of such an activity diagram would totally defeat the purpose of an activity diagram which is to model what work the system is to perform from a high level perspective.

Finally, matching such a UML activity diagram with the class and sequence diagrams would be a manual process to be done by the designer. Part of this problem is caused by having multiple dissimilar diagrammatic notations to display the behavior of the objects and the system. This is exacerbated if UML state or collaboration diagrams are needed. Additionally, there is no formal or even standard mechanism, in place, for reconciling these multiple diagrams. FOIL uses a single notation to model the system structure, individual object behavior, object interaction, and high-level process. More importantly, the FOIL algebra provides a way to mathematically verify that the individual object behaviors are internally consistent and that high-level processes will reliably perform the desired work.

4. FOIL ALGEBRA

In addition to the graphical notation, the Formal Object Interaction Language (FOIL) has a direct representation as an algebraic expression called FOIL Algebra. This algebra gives FOIL a robust mechanism for ensuring model correctness both at design-time and run-time. FOIL algebra is a variant of the π-calculus originally designed by Robin Milner [7, 8] with additional axioms and theorems for manipulating object-oriented system execution. The π-calculus as a process algebra is solely concerned with names and as such it is overly abstract for the purposes of FOIL thus specific name types (such as events and states) have been added to the algebra for clarity. While every system in FOIL algebra can be abstracted into a pure π-calculus definition, the constraints placed on FOIL algebraic construction, manipulation, and reduction are in terms of the more specific FOIL naming semantics.

This chapter provides a theoretical discussion of the application of process algebra to the FOIL graphical model. First, algebraic expression for a system is constructed by converting each graphical element into individual terms and combining them. Second, the various algebraic laws and identities are discussed to enable manipulation of system expressions for use in model verification and run-time execution. Next, the system expressions are reduced using algebraic reduction with eligibility constraints. This chapter concludes with a demonstration of construction, manipulation and reduction of a sufficiently complex workflow pattern. This chapter is necessarily abstract; however, the following chapters will contain more real world examples.

4.1 Construction

Each notational element in FOIL has an algebraic equivalent; therefore, a system comprised solely of FOIL notational elements can be completely expressed using these algebraic equivalents. Through a process of substitution an expression for a complete system can be created.

4.1.1 Events and Operators

An event in FOIL represents a name in a π-calculus system that functions to change the state of the system. In a FOIL model, the primary unit of work is an Active State. The algebraic definition of a FOIL model is not concerned with the specific work being done, only the events required to start or end the performance of that work. The system definition must include all possible options for the sequence of events that are acceptable while allowing independent event sequences to carry on concurrently. As a convention, events are represented by a lower case letter.

Figure 4.1 shows the difference in algebraic notation for consuming or receiving an event verses producing or triggering an event. The bar notation over the *t* event indicates that it is fired not received.

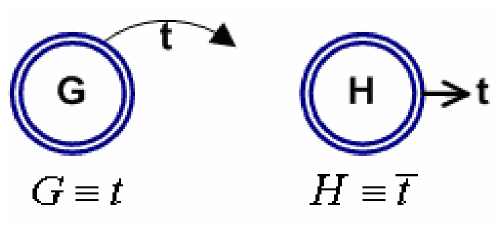
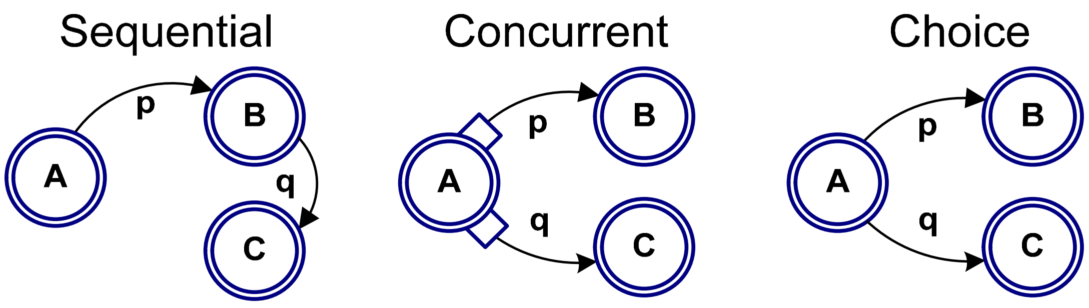


Figure 4.1 Algebraic Event Construction

State *G* is defined as transitioning upon the receipt of a *t* event while state *H* is defined as triggering a *t* event (post-trigger). FOIL uses these simple event expressions to represent complex system behaviors by using operators to define the temporal relationships between events.



$$A \equiv p.B \, and \; B \equiv q.C \quad \Rightarrow A \equiv p.q.C$$

$$A \equiv p.B \,|\, q.C$$

$$A \equiv p.B + q.C$$

Figure 4.2 Algebraic Operators

There are only three operators in the FOIL algebra: sequential, concurrent, and choice (see Figure 4.2). The sequential operator, represented by a dot or period, denotes two or more events which occur in a specified and sequential order. The concurrent operator, represented by a pipe, denotes two or more events which occur simultaneously. The choice operator, represented by a plus, denotes a choice among two or more events. All possible combinations of system operations as specified in FOIL can be completely expressed using these operators.

4.1.2 *Object Qualifiers*

There is some debate as to whether the use of object identifiers limits the flexibility in modeling object-oriented systems. However, in the case of FOIL, objects need to be able to respond to events that may be specifically designed for them. Without a mechanism for addressing a specific event to a specific object, this would not be possible. Thus, despite some drawbacks to this approach, it was decided that FOIL would use object identifiers. These identifiers can be prepended to an event term in the FOIL algebra to offer event specificity.

Figure 4.3 shows an example of a class *X* that is defined with a specific event *s* and a global event *p*. The *X* qualifier to the *s* event means that this *s* event is specific to an object instance of class *X*. Thus, when an *X* object is instantiated the expression is

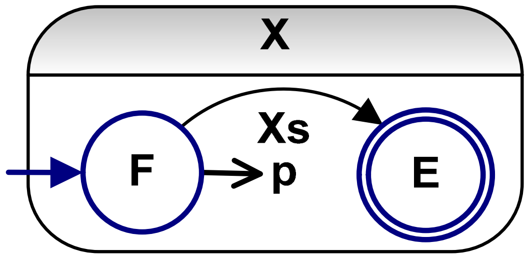


Figure 4.3 Object Qualifier

$$X_1 \equiv X_1F \equiv \overline{p}.X_1s.X_1E$$

The convention for this paper will be to sequentially number each instance of an object as its identifier but any object identifier scheme may be used.

4.1.3 State Representation

Each state of an object has an algebraic expression that represents its behavior. In that regard, state expressions are the building blocks of system definitions. The representation of passive states is rather trivial. As such, it has already been presented previously without much explanation. The expressions take on a fair amount of complexity, however, when active states are involved.

Figure 4.4 shows two examples that contain active states. There are two main problems in the algebraic representation of active states

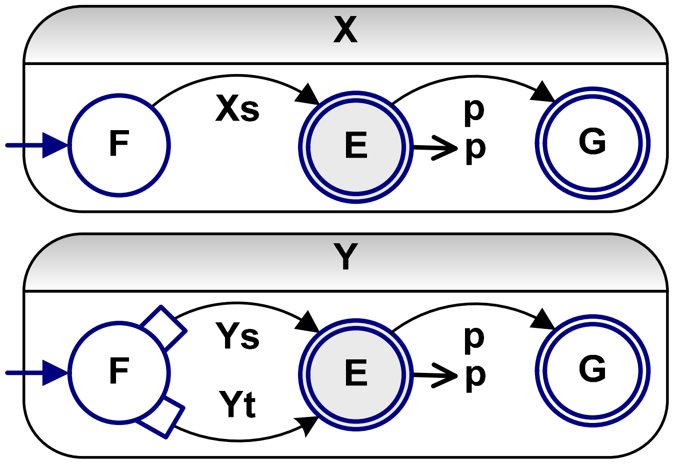


Figure 4.4 Active State Examples

demonstrated in these examples: 1) ensuring that the *p* event does not fire until the actions of *E* are complete, and 2) determining when the actions of *E* should begin. It is easy to see that neither of these problems have any consequence if state *E* is passive in either class. In order for the algebra to be robust and complete, there must be an event representation for handling active states.

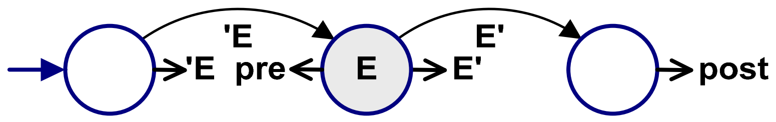


Figure 4.5 Active State Event Flow

Figure 4.5 shows the behavior in FOIL notation for the active state *E* in both objects in Figure 4.4. Of course, the FOIL notation could be drawn to show this behavior explicitly and, indeed, a diagramming tool could have this option. However, the simple shading of an active state retains simplicity in the overall diagram which could easily grow cumbersome if such behavior was explicit. The impact on the algebra of this substitution is significant. For instance, object *X* would now have the expression:

$$X \equiv F \quad F \equiv Xs.E \quad E \equiv \overline{`E}.`E.\overline{E'}.E'.\overline{p}.p.G$$
$$X \equiv Xs.\overline{`E}.`E.\overline{E'}.E'.\overline{p}.p.G$$

This complexity is stark when compared to the expression for Object X if state E were passive:

$$X \equiv F \quad F \equiv Xs.E \quad E \equiv \overline{p}.p.G$$
$$X \equiv Xs.\overline{p}.p.G$$

Likewise, object *Y* with its concurrency takes on a new character as well:

$$Y \equiv F \quad F \equiv Ys.E \mid Yt.E \quad E \equiv \overline{`E}.`E.\overline{E'}.E'.\overline{p}.p.G$$
$$Y \equiv Ys.\overline{`E}.`E.\overline{E'}.E'.\overline{p}.p.G \mid Yt.\overline{`E}.`E.\overline{E'}.E'.\overline{p}.p.G$$

The extra events fired and received with these expressions may seem rather redundant; however, the utility of this representation will become apparent later during the discussion of reductions on these expressions. In addition, it should be noted that the diagram of Figure 4.5 is only one representation of how active state behavior could be modeled. In this case, active states will only execute their actions when all threads of execution synchronize onto it (unless an input port is used). Additionally, no pre-firing events will fire until all threads have synchronized to the active state. By replacing the behavior of active states with a different state flow, the system as a whole would treat such situations differently. For example, an action could fire when the first thread reaches the state rather than waiting until synchronization occurs. For the remainder of this thesis, active states will be assumed to follow the behavior of Figure 4.5.

It may be necessary for an executable modeling system based on FOIL to know what state an object is in. This can be done through the use of a state event. This is a simple mechanism of firing an event when an object reaches a given state. It requires no additional notation but is implicit. With state events object *X* of Figure 4.4 would have the expression:

$$\begin{aligned} &X \equiv F \quad F \equiv Xs.E \quad E \equiv \overline{`E}.`E.\overline{E'}.E'.\overline{p}.p.G \\ &X \equiv \overline{F}.Xs.\overline{`E}.`E.\overline{E}.\overline{E'}.E'.\overline{p}.p.\overline{G} \end{aligned}$$

The addition of the *F*, *E* and *G* events serve to inform the system that object X has reached a those states. The *E* event occurs within the active state flow meaning that all pre-firing events must be accepted prior to being considered by the system as “arriving” at this state.

The firing of state events is completely optional. It is easy to see that such event firing does not inhibit the work of the system since no transition is dependent on such an event. It is conceivable that such events could be used by other objects to trigger additional transitions but this behavior can be modeled without such mechanisms. Additionally, it should be noted that without this mechanism, state *G* in object *X* and *Y* of Figure 4.4 has no expression unless termination is denoted by a *0* as is common in the standard π-calculus. It will be the convention of this thesis not to substitute states that have no expression.

4.1.4 Object Definition

Creating a FOIL algebraic expression of an object is a matter of substituting state expressions. This substitution of state terms is relatively trivial and has already been shown by the expressions created for Figure 4.4 and Figure 4.5. These simple cases did not have any iteration or loops. It is important that substitution only occur up to any loops or iteration. The reason for this restriction will become clear during the discussion on how these expressions are used during run-time operation of a system (i.e. reductions) and how models are verified. In addition, it is intuitive that if looping constructs are to be allowed (which they are) then substitution of terms would be infinite without at least an arbitrary stopping point. By having a clearly defined substitution stopping point, we maintain some qualities of the model which are useful.

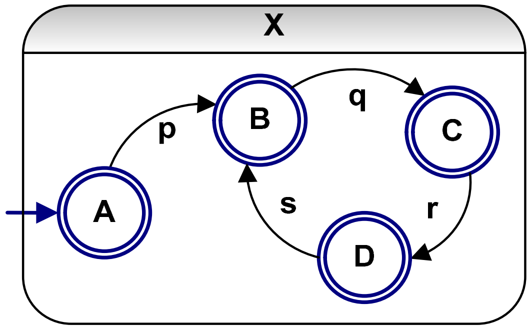


Figure 4.6 Object with Iteration

Figure 4.6 shows an example of a class definition that contains an iterative behavior. Terms are substituted in a depth-first manner using a simple depth-first search algorithm on the connected graph [76] represented by the behavior diagram. Substitution will cease whenever a back-edge is encountered thus eliminating any looping. The following shows the steps for building the expression for Figure 4.6:

1.	$X \equiv A \quad A \equiv p.B \quad B \equiv q.C \quad C \equiv r.D \quad D \equiv s.B$	*state expressions*
2.	$X \equiv p.B$	*substitute* A
3.	$X \equiv p.q.C$	*substitute* B
4.	$X \equiv p.q.r.D$	*substitute* C
5.	$X \equiv p.q.r.s.B$	*substitute* D *stop*

In a depth-first search of the graph represented by object *X*, the event *s* transition would be a back-edge. Thus, no substitutions take place beyond that transition until it is required in order to continue after a reduction. This does not mean that the same state will not be substituted twice. Figure 4.7 shows an example of where repeated substitution of the same state may occur.

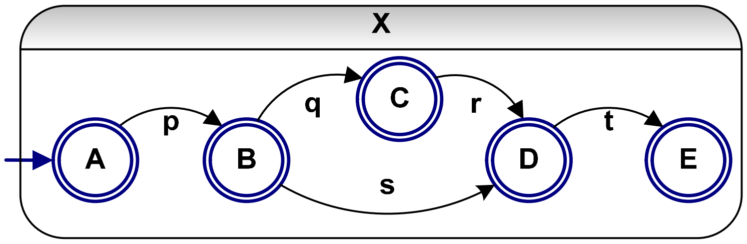


Figure 4.7 Repeated Substitution with No Loops

1.	$X \equiv A \quad A \equiv p.B \quad B \equiv s.D + q.C \quad C \equiv r.D \quad D \equiv t.E$	*state expressions*
2.	$X \equiv p.B$	*substitute* A
3.	$X \equiv p.(s.D + q.C)$	*substitute* B
4.	$X \equiv p.(s.t.E + q.C)$	*substitute* D
5.	$X \equiv p.(s.t.E + q.r.D)$	*substitute* C
6.	$X \equiv p.(s.t.E + q.r.t.E)$	*substitute* D *again*

State *D* in Figure 4.7 gets substituted twice during object expression construction. This is because the *s* event transition does not represent a back-edge during depth-first traversal (it is a cross-edge) and thus substitution should continue normally until a back-edge is encountered or no transitions are available (state *E*).

4.1.5 System Definition

The final step to construction of a system using FOIL algebra is the substitution of object expressions for the creation of a unified system expression. The substitution of object expressions generally occurs at run-time and directly correlates with object instantiation. Consider that an object-oriented program when first executed has no objects. Thus, the initial algebraic expression for a system would consist of the events that cause object instantiation from an outside source. Whether this outside source is a function, user or other system is unimportant.

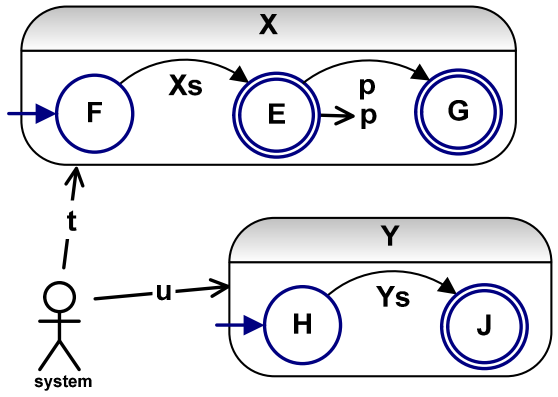


Figure 4.8 System Object Instantiation

Figure 4.8 shows an example of a complete system composed of two objects, neither of which exists prior to execution. Only when a *t* or *u* event is generated by the system will these objects be instantiated. Thus, this initial algebraic expression for this system is:

$$system \equiv t.X \mid u.Y$$

Since, objects *X* and *Y* have not been instantiated no substitution for these variables takes place. Only when an object term reaches the front of a concurrent expression during reduction will the substitution take place. However, the class expressions for objects *X* and *Y* can be predetermined prior to run-time to improve performance during object expression substitution.

4.2 Manipulation

The expressions created by the construction of a model using FOIL algebra are not very useful as created. Run-time execution and model verification place rules on the reductions that are allowed on a given system's expression. These rules would be overly complex and difficult to automate on raw expressions. Given this, expressions need to be rearranged such that they are more suitable for reduction operations and model verification.

4.2.1 Algebraic Identities

The identities associated with process algebra are fairly well known; however, FOIL takes a loose approach to equivalence. In addition, it is helpful to see how the identities function in FOIL algebraic notation rather than assuming that such notations are common knowledge. These identities are provided as axioms rather than providing rigorous proof since justification for these laws is fairly intuitive.

4.2.1.1 Distributive Law of Choice

Events fired or received before or after a choice can be distributed into the choice. Figure 4.9 shows a FOIL model of this law. Object *X* and *Y* have an equivalent behavior. In English, Object *X* would read, "Accept event *p* and then accept event *q* or accept event *r*." Object *Y*, on the other hand, reads, "Accept event *p* and then accept event *q* or accept event *p* and then accept event *r*." The logical equivalency of these two statements should be fairly intuitive. Object *Y* displays something akin to a differed choice, where two threads exist until a choice is actually made. However, since the destination of the "deferred" choice (state *B*) is the same, only a single thread need be produced during execution. This logical equivalence produces the axiomatic identity:

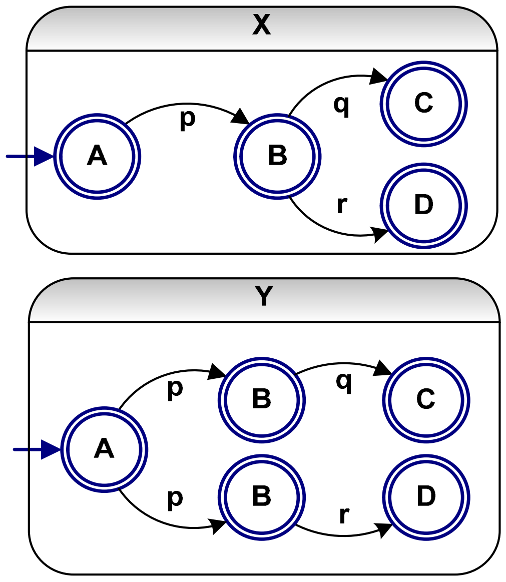


Figure 4.9 Distributive Law of Choice

$$X \equiv A \;\; and \;\; A \equiv p.B \;\; and \;\; B \equiv q.C + r.D$$
$$\Rightarrow X \equiv p.(q.C + r.D)$$
$$Y \equiv A \;\; and \;\; A \equiv p.B_1 + p.B_2 \;\; and \;\; B_1 \equiv q.C \;\; and \;\; B_2 \equiv r.D$$
$$\Rightarrow Y \equiv p.q.C + p.r.D$$
$$if \;\; X \equiv Y \;\; then$$
$$p.(q.C + r.D) \equiv p.q.C + p.r.D$$

4.2.1.2 Distributive Law of Concurrency

Sequential conditions required for the spawning of concurrent threads can be distributed to multiple threads. This identity is very similar to that for choice. Figure 4.10 shows an example of this Law. In English, object *X* would read, "Accept event *p* and then concurrently accept events *q* and *r*." Object *Y*, on the other hand, would read, "Accept event *p* and then *q* and concurrently accept event *p* and then *r*." Once again, the equivalence of these two statements should be intuitive. This produces the axiomatic identity:

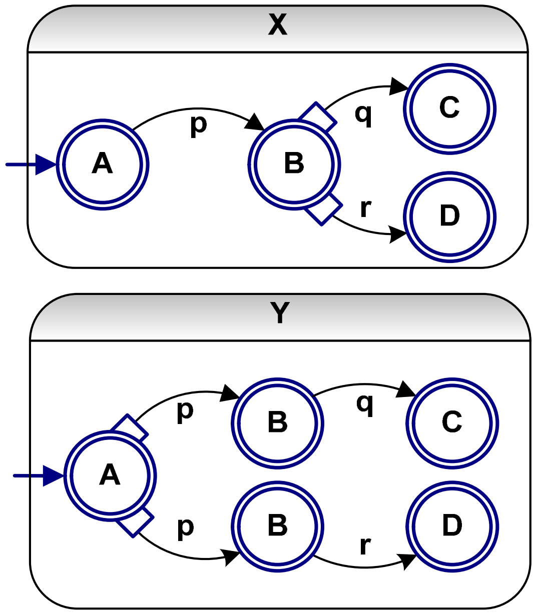


Figure 4.10 Distributive Law of Concurrency

$$X \equiv A \;\; and \;\; A \equiv p.B \;\; and \;\; B \equiv q.C \mid r.D$$
$$\Rightarrow X \equiv p.(q.C \mid r.D)$$
$$Y \equiv A \;\; and \;\; A \equiv p.B_1 \mid p.B_2 \;\; and \;\; B_1 \equiv q.C \;\; and \;\; B_2 \equiv r.D$$
$$\Rightarrow Y \equiv p.q.C \mid p.r.D$$
$$if \;\; X \equiv Y \;\; then$$
$$p.(q.C \mid r.D) \equiv p.q.C \mid p.r.D$$

4.2.1.3 Law of Redundancy

A choice between two identical sequential event expressions is not a choice. Likewise, a concurrency between two identical sequential event expressions is a single thread. It should be clear that to, “accept *p* and then *r* or accept *p* and then *r*,” is completely redundant and while it is worded as a choice between two actions there is really no choice at all. The same law holds true for concurrent relationships between event sequences.

$$p.r + p.r = p.r$$
$$p.r \mid p.r = p.r$$

4.2.1.4 Law of Concurrent Subsequence

If a sequential term in a concurrent expression is the order subsequence of another term in that same concurrent expression, then the first term may be eliminated. This law is closely connected to the reduction eligibility rule to be discussed later in this chapter. An example of this law is as follows:

$$p.q.r.s.t.u.v \mid q.s.u = p.q.r.s.t.u.v$$

4.2.1.5 Law of Nullability

If a sequential term of a concurrent expression begins with a non-event, then that expression is eliminated. If after construction or through the course of execution, all the terms of a concurrent expression begin with an event NOT being received, then that expression has no chance of execution. This particular law is based on the assumption that NOT making a choice is a passive event and would not be explicitly fired by the system. As such, an expression starting with such terms will never be reduced.

$$p.q\,|\,!r.s + !p.q\,|\,!r.s + p.q\,|\,r.s = p.q + p.q\,|\,r.s$$

4.2.1.6 Law of Contradiction

Two concurrent terms where any two events are sequentially transposed can be eliminated. Figure 4.11 shows an example of a simple contradiction. Since events *p* and *q* are unique, they can only be accepted once per iteration and thus to accept *p* would invalidate the bottom thread and likewise, to accept *q* would invalidate the top thread. This is an inherent contradiction. Such contradictions can be easily found by scanning the concurrent terms for transposed events, as in this example:

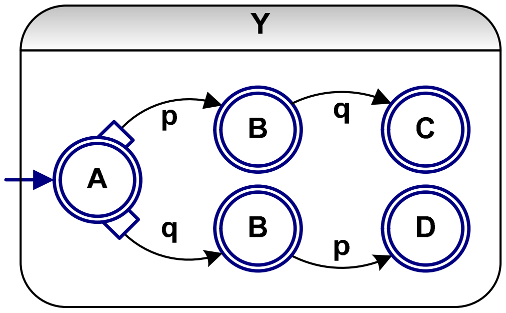


Figure 4.11 Law of Contradiction

$$p.q\,|\,q.p = 0$$

4.2.2 Algebraic Form

Using the algebraic identities described above, FOIL expressions can be rearranged to produce equivalent expressions that are useful for run-time execution and verification.

4.2.2.1 Choice-Action Form (CAF)

Any FOIL expression can be placed into a form where every possible sequence of events is handled. In effect, an expression in Choice-Action Form (CAF) is a choice among concurrent events. CAF is accomplished by fully distributing concurrency and choices using the distributive laws. Figure 4.12 shows an example of a simple

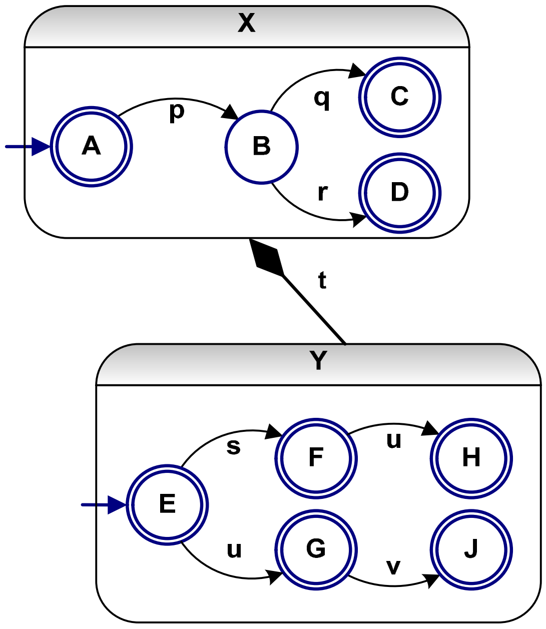


Figure 4.12 Algebraic Forms

object model which both concurrency and choices. The algebraic construction and manipulation to CAF is:

$$
\begin{array}{ll}
X \equiv A \quad A \equiv p.B \quad B \equiv t.(Y \mid (q.C + r.D)) + q.C + r.D & X \text{ states} \\
Y \equiv E \quad E \equiv s.F + u.G \quad F \equiv u.H \quad G \equiv v.J & Y \text{ states} \\
X \equiv p.(t.(Y \mid (q.C + r.D)) + q.C + r.D) & X \text{ construction} \\
Y \equiv s.u.H + u.v.J & Y \text{ construction} \\
X \equiv p.(t.((s.u.H + u.v.J) \mid (q.C + r.D)) + q.C + r.D) & \text{System construction} \\
X \equiv p.(t.(s.u.H + u.v.J) \mid t.(q.C + r.D) + q.C + r.D) & \\
X \equiv p.(t.s.u.H + t.u.v.J) \mid t.q.C + t.r.D) + q.C + r.D) & \\
X \equiv p.(t.s.u.H \mid t.q.C + t.s.u.H \mid t.r.D + t.u.v.J \mid t.q.C + t.u.v.J \mid t.r.D + q.C + r.D) & \\
X \overset{CAF}{\equiv} p.t.s.u.H \mid p.t.q.C + p.t.s.u.H \mid p.t.r.D + p.t.u.v.J \mid p.t.q.C & \\
\qquad + p.t.u.v.J \mid p.t.r.D + p.q.C + p.r.D &
\end{array}
$$

In this example, it is not necessary to substitute for object *Y* until a *t* event has been fired but doing so does not affect the execution of the model and serves to show the utility of CAF. The final expression in CAF is a complete list of all the possible concurrent outcomes for this system. In this form, it is extremely easy to use the remaining laws to eliminate terms. Additionally, CAF is used to determine whether two modeled objects are logically equivalent.

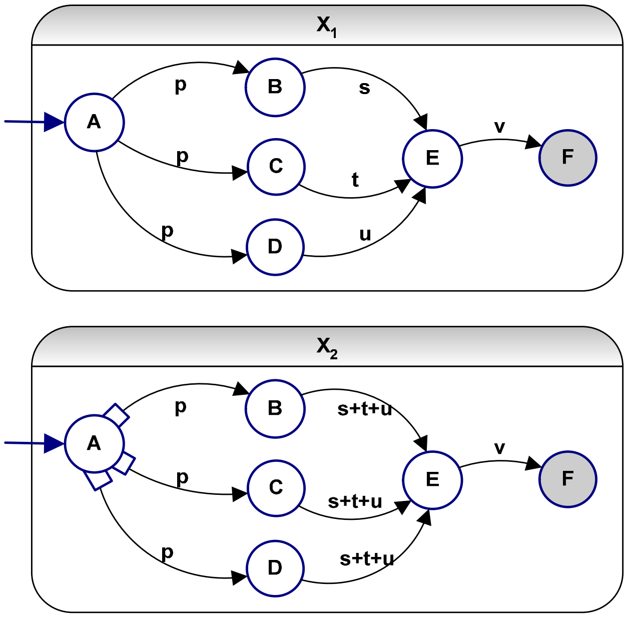


Figure 4.13 Choice-Concurrent Equivalence

The simple merge pattern allows for the construction of an interesting equivalency. Figure 4.13 shows two object behaviors that are equivalent. Object X_1 uses a deferred choice followed by a simple merge while object X_2 uses a parallel split followed by a synchronous merge. In both cases, states *B*, *C* and *D* are reached.

The difference is that object X_1 is waiting to determine where the single thread of execution exists while object X_2 has three separate threads of execution. Upon an s, t, or u event both behaviors will transition to state E once and only once. Object X_1 makes its choice while object X_2 merges its three threads into one. Algebraically, it can be shown that the two flows are the same.

$$
\begin{aligned}
&X_1 \equiv p.B + p.C + p.D \quad B \equiv s.E \quad C \equiv t.E \quad D \equiv u.E \quad E \equiv v.F \\
&X_1 \overset{CAF}{\equiv} p.s.v.F + p.t.v.F + p.u.v.F \\
&X_2 \equiv p.(B \mid C \mid D) \equiv p.B \mid p.C \mid p.D \quad B,C,D \equiv (s+t+u).E \quad E \equiv v.F \\
&X_2 \equiv p.(s+t+u).v.F \mid p.(s+t+u).vF \mid p.(s+t+u).v.F \\
&X_2 \equiv p.(s+t+u).v.F = (p.s + p.t + p.u).v.F \\
&X_2 \overset{CAF}{\equiv} p.s.v.F + p.t.v.F + p.u.v.F \\
&\Rightarrow X_2 = X_1
\end{aligned}
$$

The main drawback to CAF is that it exhibits the state explosion problem. For each optional choice used the number of possible action sequences increases by a factor of two. Thus, the growth rate of the algebraic expression is O(2^n) where n is the number of options. In object-oriented models that exhibit low coupling the size of the expressions are manageable since it is expected that the expression of any single object would be relatively small. However, in some models the size of the system expression would make run-time verification intractable.

4.2.2.2 Choice-Compressed Form (CCF)

The Choice-Compressed Form (CCF) is achieved by distributing all concurrent and sequential actions but delaying the distribution of choices until necessary for subsequent reductions. While CCF is not as easy to reduce as CAF, it does not exhibit the exponential growth rate. This means that CCF expressions will never grow too large for state-based analysis and run-time reductions. The Algebraic construction and manipulation into CCF of the system in Figure 4.12 is:

$$\begin{aligned}
X &\equiv p.(t.((s.u.H + u.v.J) \mid (q.C + r.D)) + q.C + r.D) && \textit{System construction} \\
X &\equiv p.((t.s.u.H + t.u.v.J) \mid (t.q.C + t.r.D) + q.C + r.D) \\
X &\equiv p.(t.s.u.H + t.u.v.J) \mid p.(t.q.C + t.r.D) + p.q.C + p.r.D \\
X &\overset{CCF}{\equiv} (p.t.s.u.H + p.t.u.v.J) \mid (p.t.q.C + p.t.r.D) + p.q.C + p.r.D
\end{aligned}$$

4.3 Reduction

Once any event is sent or an eligible event is received there is no reason to continue to denote it in the expression. The process of removing these terms is called a reduction. A FOIL algebraic expression is changed at run-time as a result of such reductions. The reduction process is as follows:

1. Determine Reduction Eligibility
2. Reduce the Expression
3. Fire Additional Events

4.3.1 Determine Reduction Eligibility

The first step in performing algebraic reductions is to determine whether or not the given event received is eligible. The following definition is provided with respect to FOIL algebra:

Eligibility – The system is in a state such that it is ready to process the event and the processing of said event will not place the system in a state from which it can no longer complete its work.

As an example, take the following expression:

$$X \equiv p.q.r \mid s.p.t$$

This expression represents a system that is performing two concurrent threads. It is clear that events *q*, *r*, and *t* are not eligible since the system is not in a state that is ready to receive them. It may not be so obvious that event *p* is also not eligible. The reason for this is that if unique event *p* were processed then the second concurrent term would be deadlocked since it also expects that same event *p* in the future. Another way of describing eligibility would be, "all concurrent actions that expect the event are ready to receive that event." Given this understanding, it is clear that the only eligible event is *s*.

Eligibility Rule: Given a system definition in Choice-Action Form (CAF) and the receiving of an event *b*, a choice is not eligible for reduction if event *b* exists anywhere other than the beginning of a concurrent expression. Event *b* is not eligible if there are no eligible choices.

Determining eligibility is easiest when a FOIL expression is placed in CAF. The diagram in Figure 4.14 shows an example of a multiple choice pattern for the state flow of Object *X*. Note in this case, that the receiving of event *q* before receiving event *p* will mean that *p* is no longer an option. The FOIL algebra expression for Object X in CAF is:

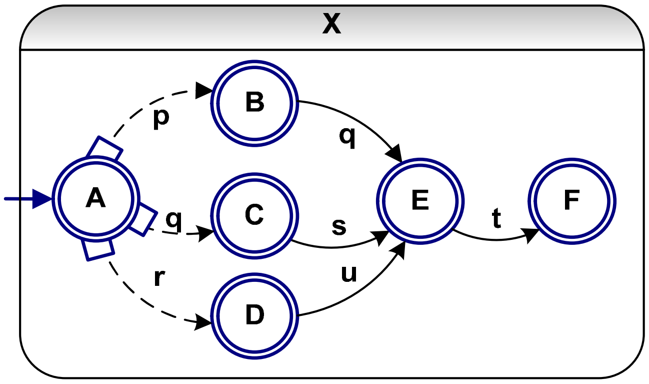


Figure 4.14 Multiple Choice Eligibility

$$X \equiv (p+!p).q.v.F \mid (q+!q).t.v.F \mid (r+!r).t.v.F$$

by Distributive Law of Choice

$$X \equiv (p.q.v.F+!p.q.v.F) \mid (q.t.v.F+!q.t.v.F) \mid (r.t.v.F+!r.t.v.F)$$

$$\begin{aligned}
&\textit{by Distributive Law of Concurrency}\\
&X \equiv (p.q.v.F \mid q.t.v.F + !p.q.v.F \mid q.t.v.F + p.q.v.F \mid !q.t.v.F\\
&\qquad + !p.q.v.F \mid !q.t.v.F) \mid (r.t.v.F + !r.t.v.F)\\
&X \equiv (p.q.v.F \mid q.t.v.F + !p.q.v.F \mid q.t.v.F + p.q.v.F \mid !q.t.v.F\\
&\qquad + !p.q.v.F \mid !q.t.v.F) \mid (r.t.v.F + !r.t.v.F)\\
&X \overset{CAF}{\equiv} p.q.v.F \mid q.t.v.F \mid r.t.v.F + !p.q.v.F \mid q.t.v.F \mid r.t.v.F +\\
&\qquad p.q.v.F \mid !q.t.v.F \mid r.t.v.F + !p.q.v.F \mid !q.t.v.F \mid r.t.v.F +\\
&\qquad p.q.v.F \mid q.t.v.F \mid !r.t.v.F + !p.q.v.F \mid q.t.v.F \mid !r.t.v.F +\\
&\qquad p.q.v.F \mid !q.t.v.F \mid !r.t.v.F + !p.q.v.F \mid !q.t.v.F \mid !r.t.v.F
\end{aligned}$$

$$\begin{aligned}
&\textit{by Law of Nullability}\\
&X \overset{CAF}{\equiv} p.q.v.F \mid q.t.v.F \mid r.t.v.F + q.t.v.F \mid r.t.v.F + p.q.v.F \mid r.t.v.F\\
&\qquad + r.t.v.F + p.q.v.F \mid q.t.v.F + q.t.v.F + p.q.v.F
\end{aligned}$$

Determining whether an event is ready to be accepted by the system is a simple matter of scanning the events at the beginning of each sequential term providing the set: {*p,q,r*}. After this, it can be determined whether each event in this set occurs anywhere other than in a concurrent term. In the first choice above, event *q* is not eligible since it occurs in the sequential expression *p.q.v.F*. Since event *q* is not in the front then this choice is ineligible. However, the event *q* remains an eligible event since there are other choices in the expression for which this event is eligible.

This example, however, clearly illustrates the state explosion problem created by using CAF. In this case, the initial construction of object *X* is already in CCF:

$$\begin{aligned}
&X \overset{CCF}{\equiv} (p + !p).q.v.F \mid (q + !q).t.v.F \mid (r + !r).t.v.F\\
&\textit{do not distribute through choices}
\end{aligned}$$

If the same eligibility rule outlined above is used on this expression in CCF, it would seem that event *q* is not eligible. The front terms of each expression will still indicate that events {*p*,*q*,*r*} are ready but modification of the rule to support CCF is required.

Eligibility Rule: Given a system definition in Choice-Compressed Form (CCF) and the receiving of an event *b*, a choice is not eligible if event *b* occurs anywhere other than the beginning of a concurrent expression and participates in any choice that does not contain a non-event. Event *b* is not eligible if there are no eligible choices.

Given this rule for CCF expressions, event *q* above is clearly eligible. It occurs downstream of a concurrent expression but does NOT participate in any choice that does not contain a non-event. It participates in the *(p+!p)* choice, but this contains a non-event. Thus, event *q* is eligible. It should be noted that the eligibility rules for CAF and CCF will always result in the same set of eligible events.

4.3.2 Reduce the Expression

Once an event is determined to be eligible, it is processed. This processing from an algebraic sense means that the system is no longer waiting on this event to occur. Thus, there is no longer any reason to denote this in the expression. In addition, while the event may have been eligible, individual choices within the system expression may not have been. Thus, these choices (having not been chosen) may be removed from the expression. Continuing with the example of Figure 4.14, the processing of the eligible event *p* on the expression in CAF would be:

$$\begin{aligned} X &\overset{CAF}{\equiv} p.q.v.F \mid q.t.v.F \mid r.t.v.F + q.t.v.F \mid r.t.v.F + p.q.v.F \mid r.t.v.F \\ &\quad +r.t.v.F + p.q.v.F \mid q.t.v.F + q.t.v.F + p.q.v.F \\ X &\xrightarrow{p} q.v.F \mid q.t.v.F \mid r.t.v.F + q.v.F \mid r.t.v.F + q.v.F \mid q.t.v.F + q.v.F \end{aligned}$$

Note that of the seven choices represented by the expression, only four of them were eligible for processing event *p*. The final expression for object *X* has removed event *p* from the front of each sequential term that participated in an eligible choice and eliminated all ineligible choices. The same object would reduce differently if event *q* were received:

$$
\begin{aligned}
X &\overset{CAF}{\equiv} p.q.v.F \mid q.t.v.F \mid r.t.v.F + q.t.v.F \mid r.t.v.F + p.q.v.F \mid r.t.v.F \\
&\quad + r.t.v.F + p.q.v.F \mid q.t.v.F + q.t.v.F + p.q.v.F \\
X &\xrightarrow{q} t.v.F \mid r.t.v.F + r.t.v.F + t.v.F
\end{aligned}
$$

In this reduction, there are only three eligible terms. The reduction eligibility rule eliminates the first and fifth choices even though these choices have a term that begins with this event.

Performing reductions in CCF is more difficult in that rather than eliminating whole terms, analysis can result in eliminating a portion of expressions.

$$
\begin{aligned}
X &\overset{CCF}{\equiv} (p+!p).q.v.F \mid (q+!q).t.v.F \mid (r+!r).t.v.F \\
X &\xrightarrow{p} q.v.F \mid (q+!q).t.v.F \mid (r+!r).t.v.F
\end{aligned}
$$

Processing event *p* results in reduction of the entire choice. Since, it is determined that indeed, one of those choices was reduced, the other choices were not and thus they can be eliminated as well. Processing of event *q* is even more complex:

$$
\begin{aligned}
X &\overset{CCF}{\equiv} (p+!p).q.v.F \mid (q+!q).t.v.F \mid (r+!r).t.v.F \\
X &\xrightarrow{q} t.v.F \mid (r+!r).t.v.F
\end{aligned}
$$

To understand this result, consider that event *q* was previously determined to be eligible by the CCF eligibility rule; however, if event *q* is accepted then none of the concurrent choices of the first term are eligible. This leads to the following CCF elimination rule:

Elimination Rule: Given a system definition in Choice-Compressed Form (CCF) and the receiving of an event *b*, if in a concurrent term event *b* is eligible merely because it participates in a non-event choice, then that concurrent term may be eliminated.

Object *X* in CCF has only one choice of three concurrent terms; however, this choice is only eligible to received event *q* because the first concurrent term, while having a downstream *q* event participates in a non-event choice. Thus, this term can be eliminated when the *q* event reduction is performed.

A reduction operation may mean that an object is created or that a loop has occurred. This is obvious during reduction when a state or object variable reaches the front of a term. Referring back to Figure 4.6, which shows a simple looping construct for object *X*. The execution of this system using FOIL algebraic reductions is:

$$
\begin{aligned}
&X \overset{CAF}{\equiv} p.q.r.s.B \xrightarrow{p} q.r.s.B \xrightarrow{q} r.s.B \xrightarrow{r} s.B \xrightarrow{s} B \\
&B \text{ is at the front so substitute} \\
&X \xrightarrow{p,q,r,s} q.r.s.B
\end{aligned}
$$

It may be convenient to number the iterations of events; this can be done with simple subscripts:

$$
\begin{aligned}
&X \overset{CAF}{\equiv} p_1.q_1.r_1.s_1.B_2 \xrightarrow{p} q_1.r_1.s_1.B_2 \xrightarrow{q} r_1.s_1.B_2 \xrightarrow{r} s_1.B_2 \xrightarrow{s} B_2 \\
&B \text{ is at the front so substitute} \\
&X \xrightarrow{p,q,r,s} q_2.r_2.s_2.B_3
\end{aligned}
$$

The event subscripts should not be confused with object identifiers that also use subscripts. Substitution of object variables, which occurs when one object instantiates another, is done in the same manner as state variables and is presented in the example at the end of this chapter. After substitution of variables it may be necessary to place the expression into CAF or CCF again.

4.3.3 Fire Additional Events

After completing the reduction operation, it may be that event firings move to the front of terms in the expression. If this is true, then they are immediately processed. Thus, event firings are always immediately removed from the terms. If multiple events reach the front simultaneously, this is only because they are participating in concurrent actions and thus the order of the event firings is unimportant. A simple queue is used to handle these multiple events. Optionally, any events fired that are ineligible can be moved to the back of the queue until only ineligible events remain. This can be used to ensure that events are not ineligible simply due to the order for which simultaneous events were fired. This option can present additional problems thus it may not be preferable. Such difficulties can be eliminated through better design of the model.

Figure 4.15 shows an example of a simple state flow for Object *Y*. In this example, events *p* and *q* are performed concurrently, thus *q* is eligible from the beginning; however, the system wants to guarantee that if event *p* is received first that event *q* is immediately fired.

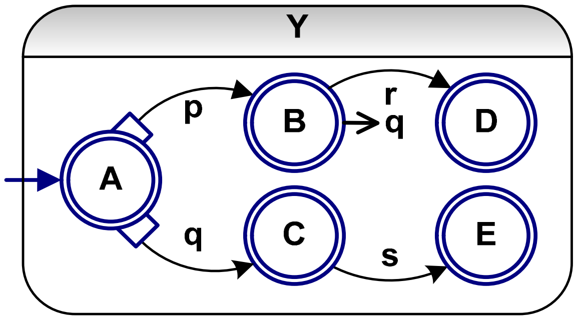


Figure 4.15 Event Firing Reduction

Algebraically, if event *p* is received first:

$$\begin{aligned} &X \overset{CAF}{\equiv} p.\bar{q}.r.D \mid q.s.E \\ &X \xrightarrow{p} \bar{q}.r.D \mid q.s.E \equiv r.D \mid q.s.E \\ &X \xrightarrow{q} r.D \mid s.E \end{aligned}$$

This example shows each step of the operation. It is permissible to simply show:

$$\begin{aligned} &X \overset{CAF}{\equiv} p.\bar{q}.r.D \mid q.s.E \\ &X \xrightarrow{p} r.D \mid s.E \end{aligned}$$

It is important to note that event firings do not affect the eligibility of a choice and thus do not affect the eligibility of an event. While there is a *q* event firing in the first concurrent term, it does not make *q* ineligible, but the later firing of event *q* in this example would be:

$$\begin{aligned} &X \overset{CAF}{\equiv} p.\bar{q}.r.D \mid q.s.E \\ &X \xrightarrow{q} p.\bar{q}.r.D \mid s.E \xrightarrow{p} \bar{q}.r.D \mid s.E \equiv r.D \mid s.E \xrightarrow{q} ineligible \end{aligned}$$

The ineligibility of a fired event does not make the originating event ineligible. An event firing is always immediately reduced. The result of the event on the system is immaterial to the eligibility of prior operations.

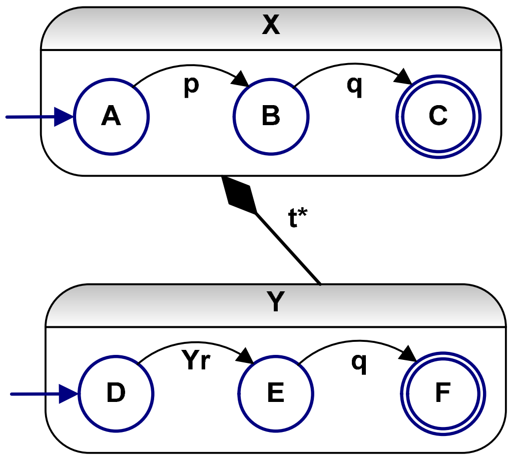


Figure 4.16 Object-Event Synchronization

4.4 Example

Figure 4.16 shows an example of a system modeled in FOIL. Object *X* is initially in state *A*. Because *A* is an accepting state object *X* can accept both *p* and *t* events. The *p* event will cause object *X* to transition to state *B*. The *t* event will cause object *X* to instantiate a new *Y* object which can subsequently begin accepting events. The asterisk indicates that multiple *Y* objects can be created by multiple *t* events being received as long as object *X* is in an accepting state.

The diagram in Figure 4.16 models a workflow pattern known as "multiple instance with no *a priori* runtime knowledge" [30]. This is one of the more complicated patterns in workflow management. The system does not know how many instances of object *Y* there will be. But, it has to make sure that all of those copies are in state *E* before accepting the *q* event. For example, the event sequence (*t,Yr,q*) would be undesirable as object *X* would still be in state *A*. Thus, while object *Y* is ready to receive the *q* event, object *X* is not ready. The problem could likewise be reversed with a sequence like (*t,p,q*). To complicate matters, the problem could be extended with an event sequence such as (t_1,Y_1r,t_2,p,q). In this case, there are two instances of *Y* but only one of them is prepared to accept the *q* event.

The following demonstrates the algebraic construction of the system in Figure 4.16 with object identifiers:

$$X_nA \equiv p.X_nB + t_i.(p.X_nB \mid A \mid Y) \quad X_nB \equiv q.C + t_i.(q.C \mid B \mid Y)$$
$$X_1 \equiv p.X_1B + t_1.(p.X_1B \mid A \mid Y)$$
$$X_1 \equiv p.(q.C + t_2.(q.C \mid B \mid Y)) + t_1.(p.(q.C + t_2.(q.C \mid B \mid Y)) \mid A \mid Y)$$
$$X_1 \equiv p.q.C + p.t_2.(q.C \mid B \mid Y) + t_1.p.(q.C + t_2.(q.C \mid B \mid Y)) \mid t_1.A \mid t_1.Y$$
$$X_1 \overset{CAF}{\equiv} p.q.C + p.t_2.q.C \mid p.t_2.B \mid p.t_2.Y + t_1.p.q.C + t_1.p.t_2.q.C \mid t_1.p.t_2.B \mid t_1.p.t_2.Y \mid t_1.A \mid t_1.Y$$

If instance number 1 of an *X* object received an *p* event, the following reduction would take place:

$$X_1 \overset{CAF}{\equiv} p.q.C + p.t_2.q.C \mid p.t_2.B \mid p.t_2.Y + t_1.p.q.C + t_1.p.t_2.q.C \mid t_1.p.t_2.B \mid t_1.p.t_2.Y \mid t_1.A \mid t_1.Y$$
$$\xrightarrow{p} q.C + t_2.q.C \mid t_2.B \mid t_2.Y$$

Note that two choices were completely removed because the *t* event was not received and these choices were eliminated. The remaining expressions were reduced by eliminating the *p* events from the remaining applicable expressions. The final definition now represents the state of the system after receiving event *p*. Some interesting things to note from this current definition are:

- Receiving an event *q* will now completely eliminate event *t* from the definition. This is logical since, if *q* is received, then any new *Y* object will never complete since *q* has already processed.
- Receiving an event *t* would place *B* at the front of a term. This would be expanded and the definition again placed into choice-action form (CAF).

As discussed earlier, if the system received and accepts the events (*t*, Y_1r, *q*) the system would be hung since the X_1 object is not in a state that can accept the *q* event even though the Y_1 instance is ready. Reduction of these events yields:

$$X_1 \overset{CAF}{\equiv} p.q.C + p.t_2.q.C \mid p.t_2.B \mid p.t_2.Y + t_1.p.q.C + t_1.p.t_2.q.C \mid t_1.p.t_2.B \mid t_1.p.t_2.Y \mid t_1.A \mid t_1.Y$$
$$\xrightarrow{t} p.q.C + p.t_2.q.C \mid p.t_2.B \mid p.t_2.Y \mid A \mid Y$$

This triggers the instantiation of object *Y*. Anytime a name reaches the front of a concurrent action and does not have a defined subscript, it is assumed that new object creation has occurred and the subscript is replaced with the next iteration of the object instance. Continuing with the reductions:

$$
\begin{aligned}
& X_1 \overset{CAF}{\equiv} p.q.C + p.t_2.q.C \mid p.t_2.B \mid p.t_2.Y + t_1.p.q.C + t_1.p.t_2.q.C \mid t_1.p.t_2.B \mid t_1.p.t_2.Y \mid t_1.A \mid t_1.Y \\
& \xrightarrow{t} p.q.C + p.t_2.q.C \mid p.t_2.B \mid p.t_2.Y \mid A \mid Y \\
& \textit{create object } Y \\
& \equiv A \mid Y_1 r.q.Y_1 F \\
& \textit{substitute } A \\
& \equiv (p.q.C + p.t_2.q.C \mid p.t_2.B \mid p.t_2.Y + t_1.p.q.C + t_1.p.t_2.q.C \mid t_1.p.t_2.B \mid t_1.p.t_2.Y \mid t_1.A \mid t_1.Y) \mid Y_1 r.q.Y_1 F \\
& \overset{CAF}{\equiv} p.q.C \mid Y_1 r.q.Y_1 F + p.t_2.q.C \mid p.t_2.B \mid p.t_2.Y \mid Y_1 r.q.Y_1 F + t_1.p.q.C \mid Y_1 r.q.Y_1 F + \\
& \qquad t_1.p.t_2.q.C \mid t_1.p.t_2.B \mid t_1.p.t_2.Y \mid t_1.A \mid t_1.Y \mid Y_1 r.q.Y_1 F \\
& \xrightarrow{Y_1 r} p.q.C \mid q.Y_1 F + p.t_2.q.C \mid p.t_2.B \mid p.t_2.Y \mid q.Y_1 F + t_1.p.q.C \mid q.Y_1 F + \\
& \qquad t_1.p.t_2.q.C \mid t_1.p.t_2.B \mid t_1.p.t_2.Y \mid t_1.A \mid t_1.Y \mid q.Y_1 F \\
& \xrightarrow{q} \textit{no eligible choices}
\end{aligned}
$$

A look at the final reduction demonstrates the utility of the eligibility rule. All four of the concurrent choices are ready to accept a *q* event and without the rule the reduction would proceed normally; however, all four choices have a *q* embedded in one of their concurrent components. The eligibility rule states that a choice is not eligible if the event occurs anywhere other than the beginning of a concurrent component. Based on this, none of these action choices are eligible and thus the event is not accepted. Correctly receiving a *p* event will make one of the concurrent terms *q* eligible, as follows:

$$
\begin{aligned}
& X_1 \overset{CAF}{\equiv} p.q.C + p.t_2.q.C \mid p.t_2.B \mid p.t_2.Y + t_1.p.q.C + t_1.p.t_2.q.C \mid t_1.p.t_2.B \mid t_1.p.t_2.Y \mid t_1.A \mid t_1.Y \\
& \xrightarrow{t} p.q.C \mid Y_1 r.q.Y_1 F + p.t_2.q.C \mid p.t_2.B \mid p.t_2.Y \mid Y_1 r.q.Y_1 F + t_1.p.q.C \mid Y_1 r.q.Y_1 F + \\
& \qquad t_1.p.t_2.q.C \mid t_1.p.t_2.B \mid t_1.p.t_2.Y \mid t_1.A \mid t_1.Y \mid Y_1 r.q.Y_1 F \\
& \xrightarrow{Y_1 r} p.q.C \mid q.Y_1 F + p.t_2.q.C \mid p.t_2.B \mid p.t_2.Y \mid q.Y_1 F + t_1.p.q.C \mid q.Y_1 F + \\
& \qquad t_1.p.t_2.q.C \mid t_1.p.t_2.B \mid t_1.p.t_2.Y \mid t_1.A \mid t_1.Y \mid q.Y_1 F \\
& \xrightarrow{p} q.C \mid q.Y_1 F + t_2.q.C \mid t_2.B \mid t_2.Y \mid q.Y_1 F \\
& \xrightarrow{q} C \mid Y_1 F
\end{aligned}
$$

The following example demonstrates that it does not matter whether the $Y_1 r$ event or the *p* event is received first as long as both of them are received before the *q*.

$$X_1 \overset{CAF}{\equiv} p.q.C + p.t_2.q.C \mid p.t_2.B \mid p.t_2.Y + t_1.p.q.C + t_1.p.t_2.q.C \mid t_1.p.t_2.B \mid t_1.p.t_2.Y \mid t_1.A \mid t_1.Y$$
$$\xrightarrow{t} p.q.C \mid Y_1r.q.Y_1F + p.t_2.q.C \mid p.t_2.B \mid p.t_2.Y \mid Y_1r.q.Y_1F + t_1.p.q.C \mid Y_1r.q.Y_1F + t_1.p.t_2.q.C \mid t_1.p.t_2.B \mid t_1.p.t_2.Y \mid t_1.A \mid t_1.Y \mid Y_1r.q.Y_1F$$
$$\xrightarrow{p} q.C \mid Y_1r.q.Y_1F + t_2.q.C \mid t_2.B \mid t_2.Y \mid Y_1r.q.Y_1F$$
$$\xrightarrow{q} \textit{not eligible}$$
$$\xrightarrow{Y_1r} q.C \mid q.Y_1F + t_2.q.C \mid t_2.B \mid t_2.Y \mid q.Y_1F$$
$$\xrightarrow{q} C \mid Y_1F$$

As a final example, if two instances of object *Y* are created by two separate *t* events, the eligibility rule ensures that all instances are synchronized before continuing.

$$X_1 \overset{CAF}{\equiv} p.q.C + p.t_2.q.C \mid p.t_2.B \mid p.t_2.Y + t_1.p.q.C + t_1.p.t_2.q.C \mid t_1.p.t_2.B \mid t_1.p.t_2.Y \mid t_1.A \mid t_1.Y$$
$$\xrightarrow{t} p.q.C \mid Y_1r.q.Y_1F + p.t_2.q.C \mid p.t_2.B \mid p.t_2.Y \mid Y_1r.q.Y_1F + t_1.p.q.C \mid Y_1r.q.Y_1F + t_1.p.t_2.q.C \mid t_1.p.t_2.B \mid t_1.p.t_2.Y \mid t_1.A \mid t_1.Y \mid Y_1r.q.Y_1F$$
$$\xrightarrow{p} q.C \mid Y_1r.q.Y_1F + t_2.q.C \mid t_2.B \mid t_2.Y \mid Y_1r.q.Y_1F$$
$$\xrightarrow{t} q.C \mid Y_1r.q.Y_1F + q.C \mid Y_2r.q.Y_2F \mid Y_1r.q.Y_1F + q.C \mid Y_2r.q.Y_2F \mid Y_1r.q.Y_1F \mid t_3.q.C \mid t_3.B \mid t_3.Y$$
$$\xrightarrow{q} \textit{not eligible}$$
$$\xrightarrow{Y_1r} q.C \mid q.Y_1F + q.C \mid Y_2r.q.Y_2F \mid q.Y_1F + q.C \mid Y_2r.q.Y_2F \mid q.Y_1F \mid t_3.q.C \mid t_3.B \mid t_3.Y$$
$$\xrightarrow{q} \textit{still not eligible}$$
$$\xrightarrow{Y_2r} q.C \mid q.Y_1F + q.C \mid q.Y_2F \mid q.Y_1F + q.C \mid q.Y_2F \mid q.Y_1F \mid t_3.q.C \mid t_3.B \mid t_3.Y$$
$$\xrightarrow{q} C \mid Y_1F + C \mid Y_2F \mid Y_1F$$

5. CONCURRENCY, INHERITANCE, AND MODEL VERIFICATION

The Formal Object Interaction Language (FOIL) shows its utility most effectively when used to model complex systems. In addition to its inherent support for concurrency and its conformity to an object-oriented paradigm, it can be used to verify certain attributes of a complete system, and to analyze individual objects and states.

5.1 Concurrency

The ability to model systems that can perform concurrent actions is becoming more important in an age of distributed systems. FOIL has a method for modeling such simultaneous actions through the mechanism of thread spawning. As the notation and algebra of FOIL have already been explained, an understanding of how some concurrent patterns are modeled will aid in the understanding of the expressive power of the FOIL model.

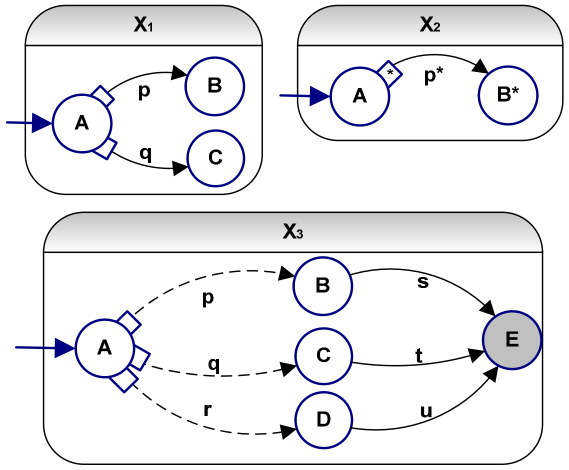


Figure 5.1 FOIL Thread Spawning

5.1.1 Spawning Threads

Spawning multiple threads of execution is done, primarily, by the use of the output port notational element. The output port indicates that the object will remain in its initial state until all output ports are satisfied. Figure 5.1 shows an example of a simple case where object X_1 will remain in state *A* until both a *p* and a *q* event are received. After the *p* event, object X_1 will be in state {*A*, *B*}. After the *q* event, object X_1 will be in state {*B, C*}. Thus, object X_1 begins its life in a single state but ends life in multiple states due to the thread spawning effect of the output ports.

Initially, this sounds like an easy concept, but there are many ways to model thread spawning, and in some of them the number of output transitions required for completing a state is either unknown or infinite. FOIL can handle all of these cases both by notation and by algebra.

Object X_3 in Figure 9 shows an example of a case where the number of output transitions to complete state *A* is unknown. In this case, any one of three events can be received while in state *A* but they are all optional. In this model, we must receive one of the events for the object to progress but we may receive multiple events which must be processed. Thus, the number of threads spawned is unknown at design-time. In fact, the number of threads required is not even known at run-time until the *E* state is reached by one or more threads. Thus, only when the *E'* event is received and all threads which have left *A* have reached *E*, will the object complete transition out of state *A*. The algebra clearly handles this case:

$$\begin{aligned}
X_3 &\equiv (p+!p).s.E' \,|\, (q+!q).t.E' \,|\, (r+!r).u.E' \\
X_3 &\overset{CAF}{\equiv} p.s.E'|q.t.E'|r.u.E' + p.s.E'|q.t.E'|!r.u.E' + p.s.E'|!q.t.E'|r.u.E' + \\
&\quad p.s.E'|!q.t.E'|!r.u.E' + !p.s.E'|q.t.E'|r.u.E' + !p.s.E'|q.t.E'|!r.u.E' + \\
&\quad !p.s.E'|!q.t.E'|r.u.E' + !p.s.E'|!q.t.E'|!r.u.E' \\
&\mathrm{Re}\,move\ \ Not-Event\ \ Actions
\end{aligned}$$

$$
\begin{aligned}
X_3 &\overset{CAF}{\equiv} p.s.E'|\,q.t.E'|\,r.u.E'+p.s.E'|\,q.t.E'+p.s.E'|\,r.u.E'+\\
&\quad p.s.E'+q.t.E'|\,r.u.E'+q.t.E'+r.u.E'\\
&\xrightarrow{p} s.E'|\,q.t.E'|\,r.u.E'+s.E'|\,q.t.E'+s.E'|\,r.u.E'+s.E'\\
&\xrightarrow{q} s.E'|\,t.E'|\,r.u.E'+s.E'|\,t.E'\\
&\xrightarrow{s} E'|\,t.E'|\,r.u.E'+E'|\,t.E'\\
&\xrightarrow{E'} ineligible\\
&\xrightarrow{t} E'|\,E'|\,r.u.E'+E'|\,E'\\
&\xrightarrow{E'} done
\end{aligned}
$$

Another case involves the concept that an object will never completely transition from a given state. This pattern can be used to model listening devices or objects that will infinitely react to events and process them. Object X_2 of Figure 5.1 shows an example of such a pattern. In this case, object X_2 will never fully transition out of state A. As each event p is received a new state B is created and processing continues. Thus, the initial state of object X_2 is $\{A\}$; after a p event, it becomes $\{A, B_1\}$; after another p event, it becomes $\{A, B_1, B_2\}$ and so on. The algebraic construction and operation clearly shows this behavior:

$$
\begin{aligned}
X_2 &\equiv p_1.(B_1'|\,A)\\
X_2 &\equiv p_1.B_1'|\,p_1.A\\
&\xrightarrow{p} B_1'|\,(p_2.B_2'|\,p_2.A)\\
&\xrightarrow{p} B_1'|\,B_2'|\,(p_3.B_3'|\,p_3.A)\\
&\xrightarrow{B_2'} B_1'|\,(p_3.B_3'|\,p_3.A)
\end{aligned}
$$

The *MasterController* object in Figure 3.12 of the elevator system is an example of this pattern in practical use.

5.1.2 Merging Threads

Perhaps an even more complicated situation that arises from modeling concurrency is how to merge multiple threads of execution. In some cases, Petri-nets fall short when it comes to this problem. For example, whether threads merge synchronously or asynchronously must be considered. Additionally, one must distinguish between a model merging and a thread merging.

Figure 5.2 shows three examples of identical object thread spawning; however, all of these cases merge differently. Object X_1 shows a standard synchronous merge; meaning that an object of type X_1 will not accept a *t* event unless both threads completely reach state *D*. Note that there is no specific notation for a synchronous merging of two behavioral threads. This is because the reduction eligibility rule automatically enforces this constraint.

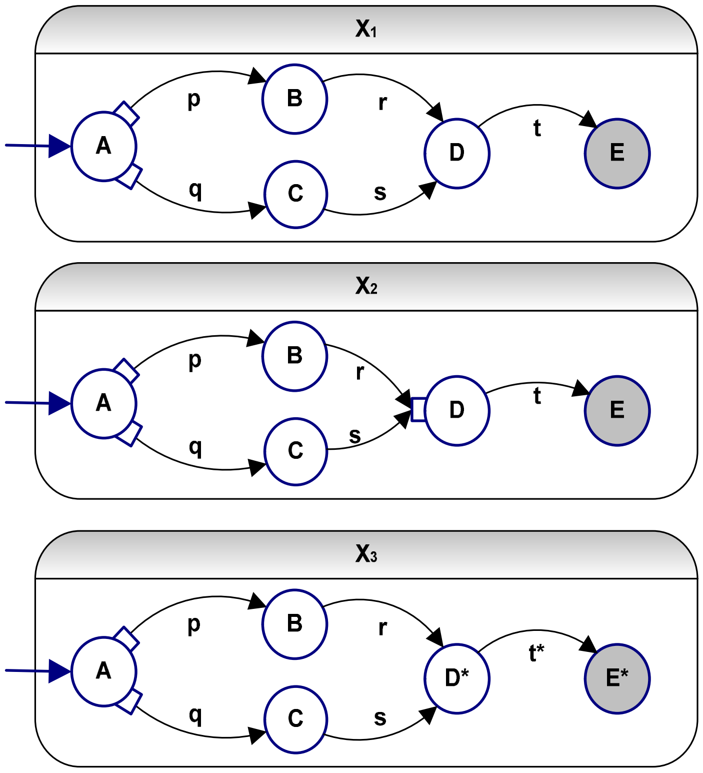


Figure 5.2 FOIL Thread Merging

$$
\begin{aligned}
&X_1 \equiv p.r.t.E' \,|\, q.s.t.E' \\
&\xrightarrow{q} p.r.t.E' \,|\, s.t.E' \\
&\xrightarrow{s} p.r.t.E' \,|\, t.E' \\
&\xrightarrow{t} \mathit{ineligible} \\
&\xrightarrow{p} r.t.E' \,|\, t.E' \\
&\xrightarrow{t} \mathit{ineligible} \\
&\xrightarrow{r} t.E' \,|\, t.E' \equiv t.E' \\
&\xrightarrow{t} E'
\end{aligned}
$$

Object X_2, on the other hand, models an asynchronous merge. In this case, the first thread reaching state *D* will be allowed to continue on with execution. The second thread will merge when it reaches state *D* regardless of the state of the first thread. The action of state *E* will not be executed twice even though it may be executed before both threads reach state *D*.

$$
\begin{array}{l}
X_2 \equiv A \qquad A \equiv p.B \mid q.C \\
B \equiv r.D + r \quad C \equiv s.D + s \quad D \equiv t.E' \\
X_2 \equiv p.(r.t.E' + r) \mid q.(s.t.E' + s) \\
X_2 \overset{caf}{\equiv} p.r.t.E' \mid q.s.t.E' + p.r.t.E' \mid q.s + p.r \mid q.s.t.E' + p.r \mid q.s \\
\xrightarrow{q} p.r.t.E' \mid s.t.E' + p.r.t.E' \mid s + p.r \mid s.t.E' + p.r \mid s \\
\xrightarrow{s} p.r.t.E' \mid t.E' + p.r.t.E' + p.r \mid t.E' + p.r \\
p.r \mid t.E' \textit{ is the only eligible term for } t \\
\xrightarrow{t} p.r \mid E' \\
\xrightarrow{p} r \mid E'
\end{array}
$$

Class X_3 is not a thread merging at all. It represents a model or multi merge. In this case, the two threads remain independent. This would be the same as having two state *D*'s and two state *E*'s. Thus, state *D* and *E* will each be executed twice, once by each thread. In order to distinguish them in the calculus, subscripts are used to represent different state instances in the instantiated object.

$$
\begin{array}{l}
X_3 \equiv A \qquad A \equiv p.B \mid q.C \\
B \equiv r.D_n \quad C \equiv s.D_n \quad D_n \equiv t_n.E_n' \\
X_3 \equiv p.r.t_1.E_1' \mid p.r.t_2.E_2'
\end{array}
$$

5.1.3 Active State Interrupt

To complete the representation of concurrency in FOIL, it is important to understand how active states perform their work. When a thread of execution arrives at an active state, all pre-fire events are transmitted (i.e. method calls are made) in a concurrent manner. This means that method calls in FOIL are assumed to run in their own thread on a sequential system. They are merely transmitted asynchronously in a distributed system. Once this is complete, the active state is free to perform its active state code.

The execution of the state's actions must also be performed in its own thread. The main thread will wait on this process while continuing to listen for events that may cause a transition to occur. Thus, the reception of an eligible event will result in immediate suspension of active state processing.

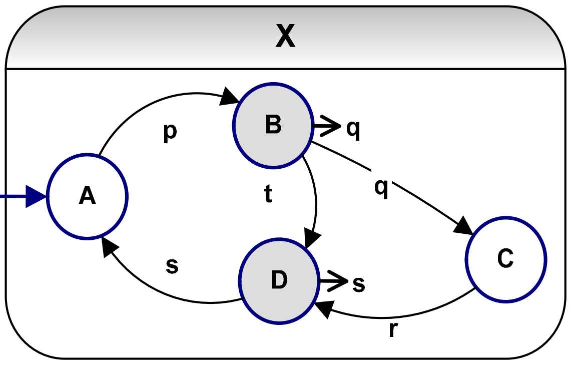


Figure 5.3 FOIL Active State Interrupt

Figure 5.3, a model of the *Door* object used by an elevator system (see Figure 3.12), is an example of how this mechanism is understood in FOIL. Object *X* starts in state *A*. Upon receiving a *p* event it will transition to *B* and begin executing B's active state code. Upon completion of *B*'s code (*B'*), it will fire a *q* event, which will cause it to transition to state *C*. However, if *B* receives a *t* event prior to completion of its code, it will transition to *D* and event *q* will never be fired. This is completely determined by the implied representation of active states. The behavior of active states as outlined here is based on the underlying representation outlined in 4.1.3 and the assumption that active state execution is current with other system operations. Given these assumptions the following reductions demonstrate the active state interrupt behavior of the model in Figure 5.3:

$$A \equiv p.B \quad B \equiv \overline{`B}.`B.B'.\overline{q} \mid (q.C + t.D) \quad C \equiv r.D \quad D \equiv \overline{`D}.`D.D'.\overline{s} \mid s.A$$
$$X \overset{CAF}{\equiv} p.\overline{`B}.`B.B'.\overline{q} \mid p.q.r.\overline{`D}.`D.D'.\overline{s} \mid p.q.r.s.A + p.\overline{`B}.`B.B'.\overline{q} \mid p.t.\overline{`D}.`D.D'.\overline{s} \mid p.t.s.A$$
$$\xrightarrow{p} B'.\overline{q} \mid q.r.\overline{`D}.`D.D'.\overline{s} \mid q.r.s.A + B'.\overline{q} \mid t.\overline{`D}.`D.D'.\overline{s} \mid t.s.A$$

If event *t* is received before *B* completes:

$$\xrightarrow{t} B'.\overline{q} \mid D'.\overline{s} \mid s.A$$

Active State *B* finishes but *q* is no longer eligible:

$$\xrightarrow{B'} D'.\overline{s} \mid s.A$$

In this example, *B* does actually complete processing even though the *t* event is received. Post-firings of the active state may still be processed. It is completely possible that by making different assumptions with regard to how active states behave that the system would perform differently. Likewise, if a constraint was made that events can only be accepted following completion of active state processing (i.e. sequential), then the *t* event would not be eligible until after the *B'* event is received. The following shows the algebra for the same sequence of events and the same active state representation but with the assumption of sequential process of interrupt events:

$$A \equiv p.B \quad B \equiv \overline{`B}.`B.B'.\overline{q}.(q.C + t.D) \quad C \equiv r.D \quad D \equiv \overline{`D}.`D.D'.\overline{s}.s.A$$
$$X \overset{CAF}{\equiv} p.\overline{`B}.`B.B'.\overline{q}.q.r.\overline{`D}.`D.D'.\overline{s}.s.A + p.\overline{`B}.`B.B'.\overline{q}.t.\overline{`D}.`D.D'.\overline{s}.s.A$$
$$\xrightarrow{p} p.\overline{`B}.`B.B'.\overline{q}.q.r.\overline{`D}.`D.D'.\overline{s}.s.A + p.\overline{`B}.`B.B'.\overline{q}.t.\overline{`D}.`D.D'.\overline{s}.s.A$$
$$\xrightarrow{t} \textit{not eligible}$$

The decision on how active states are treated could be made on an object or event a state level; however, FOIL currently has no notational variant to denote such treatment.

5.2 Inheritance

It is safe to say that FOIL could not be considered a truly object-oriented (OO) modeling language if it did not support inheritance. The code saving attribute of inheritance is one of the hallmarks of OO development. Two of the other attributes of OO development, abstraction and encapsulation, do not deviate from the traditional sense when expressed using FOIL. The other major attribute of OO programming is polymorphism and is primarily an implementation issue and does not impact the modeling of such systems in a specific way. Therefore, the specific mechanisms of polymorphism are not discussed in this thesis. It is safe to assume that, if it can be successfully demonstrated that inheritance is supported, the implementation of polymorphism is a programming-language-specific function and can be accomplished in a meaningful way when represented by a FOIL model.

5.2.1 Structural Inheritance

In typical object-oriented (OO) development, the term inheritance deals with the “is a” relationship of one object to another. For instance, a sparrow “is a” bird. While this relationship is intuitive, it may not be obvious that from a programming perspective, this inheritance relationship – sometimes referred to as generalization – only applies to the structural definition, or interface, of a class or object. FOIL does not contradict this notion.

Figure 5.4 is a more detailed FOIL model of the simple inheritance model of Figure 3.7. In order to demonstrate that FOIL models exhibit *interface conformity*, as in the typical definition of inheritance in OO development, the approach will be to convert the classes of this diagram into typical OO class definitions. This will prove that a FOIL model exhibits structural inheritance if:

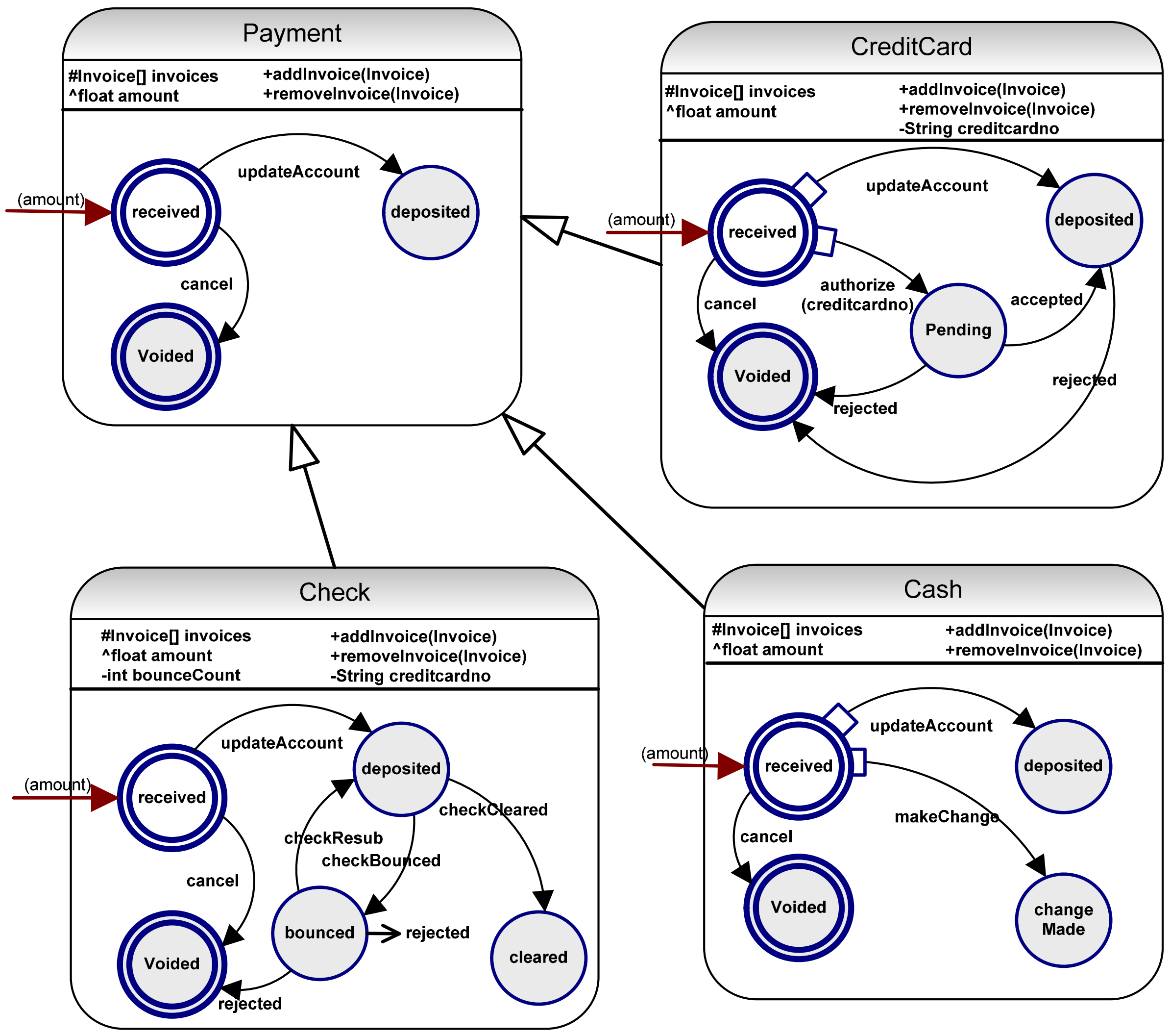


Figure 5.4 Structural and Behavioral Inheritance Example

- The conversion process is generic and repeatable for all such models.
- The conversion does not in any way add additional information to the model.
- The resulting conversion, while being less expressive than the original FOIL diagram, results in a valid OO class diagram.

The conversion of a FOIL model to a typical UML class diagram is relatively simple. Since FOIL offers additional information to a typical OO model, we simply extract from the FOIL model those methods and attributes which comprise the subset of information contained in the entire object. For example, FOIL implicitly tracks the state of the object and state tracking is non-deterministic and thus a state collection would have to be maintained in a typical OO language. However, since every such object in FOIL would have such a condition this would not aid in proving inheritance.

Every event in a FOIL model is received by the system as a whole and distributed to the object by some mechanism. This could be a distributed event service or an object to object call as is the case in OO development. Thus, each event could be viewed as a public method. Likewise, each active state performs work specific to that object and thus could be considered a protected method. A conversion of the read-only attributes as specified in FOIL to the appropriate protected attribute with a getter method for access would also have to be done.

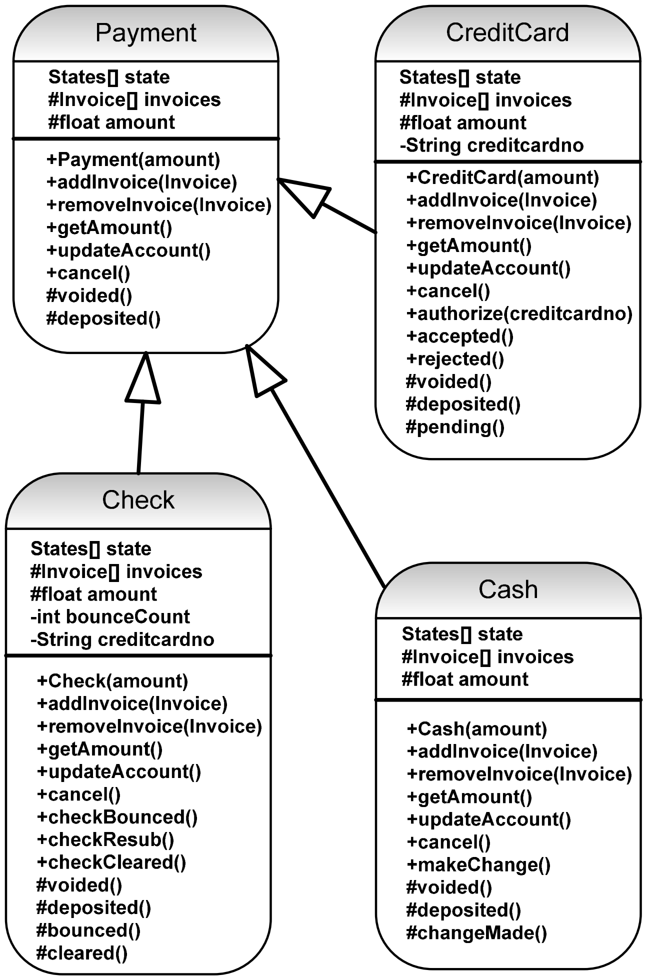


Figure 5.5 OO Equivalent of FOIL Inheritance

Combining the methods from these steps with the attributes and methods specified by FOIL in the traditional UML manner would result in the simple class diagram of Figure 5.5. It is evident that this resulting UML diagram is valid and since the method described above can be performed on any FOIL diagram then FOIL does conform to the industry-standard definition of inheritance.

In addition to proving that FOIL does provide for structural object-oriented inheritance, the resulting UML diagram provides proof that a FOIL diagram is far more expressive than its UML counterpart. In UML, the modeling required to provide the same level of behavioral detail would require numerous diagrams. In addition, this conversion process provides evidence regarding the intuitive nature of FOIL diagrams as compared with UML. While this evidence certainly does not constitute proof, it does suggest that such a claim may be plausible.

5.2.2 Behavioral Inheritance

While FOIL complies with the traditional notion of inheritance, it is difficult to see how this idea of inheritance makes implementation of polymorphism intuitive. Polymorphism means that one object can act like another and, in as far as one object can do all the things of another, this definition is completely satisfied by the concept of *interface conformity*. However, if the notion of polymorphism included that the object must behave the same way, then the concept of behavioral inheritance must be introduced.

Behavioral inheritance is not a new concept [70, 74]. It is easy to expand the idea of an "is a" relationship as being one where one object can do all the things that another can AND must do so in the same manner. Obviously, if an object does exactly the same thing in exactly the same way as another than those two objects are equivalent and there is no need for inheritance. However, an extension or override of behavior is allowed in the same was as an extension or override of an interface.

Given the fact that a class can extend or override the behavior of another, behavioral inheritance as a concept must be clearly defined. In FOIL, the informal definition of behavioral inheritance is:

Object *X* is said to be inherited from object *Y*, if it conforms to the same interface AND for all states in *Y* there are corresponding states in *X* such that the receipt of any event in *Y* will result in the same transition as that of *X*.

Formally, the behavior of an object is represented by a tuple:

$$O = \left\langle S, R, F, \theta \right\rangle$$

Where *S* is the set of states in *O*, *R* is the set of events received by *O*, *F* is the set of events fired by *O*, and *θ* is the set of transition functions performed by *O*. Formally, an object X inherits from Y if:

$$\theta_y \in \theta_x$$

Referring back to the example of Figure 5.4 extension of behavior can occur in one of three ways: sequential extension, concurrent extension, and choice extension. The *Check* class shows an example of sequential extension. The *Payment* class behavior is basically untouched in the *Check* class but where the *Payment* class would end the *Check* class has been extended to add additional states and transitions. The *Cash* class shows an example of concurrent extension. In this case, the terminating states of the parent class (*Payment*) remain the terminating states of the child but there are additional terminating states by way of concurrent actions. These two methods can be combined in the same object like the *CreditCard* class which is both a concurrent and sequential extension on the *Payment* behavior. Choice extension while not demonstrated in Figure 5.4 is similar to concurrent extension but is comprised of choices.

From a polymorphic perspective, choice and sequential extension provide some interesting side effects. For instance, if an object is treated as its inherited parent, some states in the object may not exist in the parent. In this case, the object is considered to be in the last state it was in that is in the set of states of the parent. For instance, if the *Check* object above is in the *cleared* state, then if it were treated as a generic *Payment*, it would be in the *deposited* state.

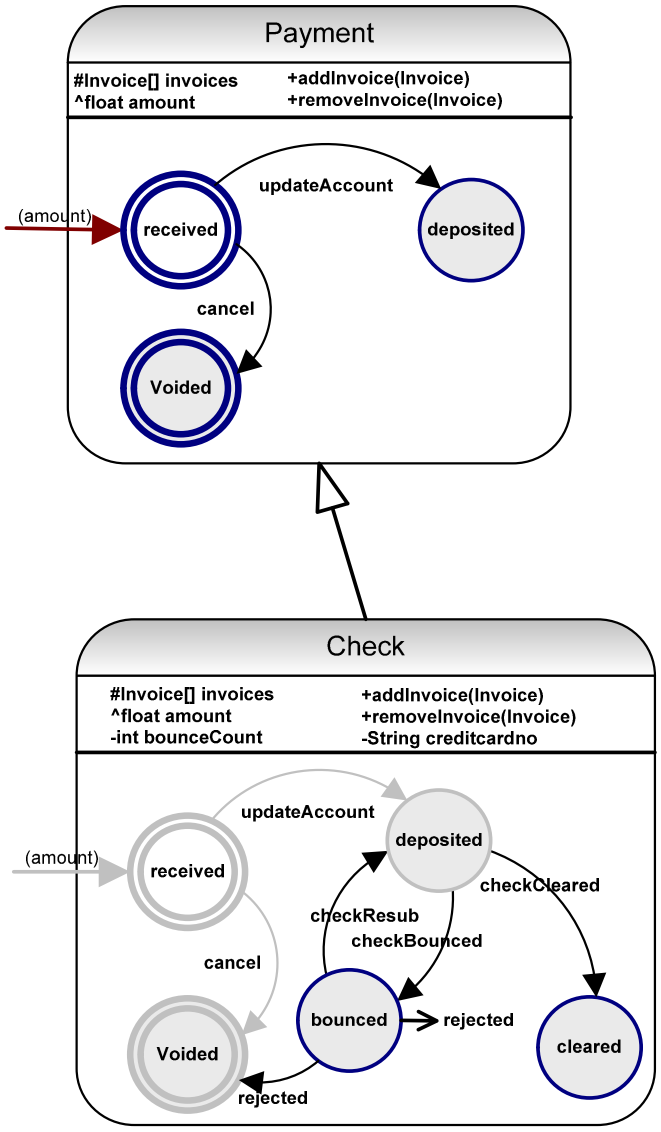


Figure 5.6 Alternate Behavioral Inheritance Notation

Ensuring proper behavioral inheritance notation is quit simple. Copying the behavioral specification of an object to another and then extending the behavior, adding concurrent actions or adding additional choices will result in a second object that can be said to inherit the behavior of the first. Figure 5.6 shows an optional way to denote the commonalities that may aid in clearly communicating this relationship.

The behavioral inheritance characteristic of FOIL can also be verified algebraically. This follows from the formal definition given previously.

An object *X* exhibits behavioral inheritance with respect to object *Y*, if for each sequential term of the FOIL algebraic expression for *Y* there is a corresponding expression in *X* that is a sequential superset.

As an example, consider again Figure 5.4. If the first letter of each state and event is used as an algebraic term, then the *Payment* class would be expressed as:

$$Payment \equiv R.u.\overline{\grave{D}}.\grave{D}.D'.D + R.c.\overline{\grave{V}}.\grave{V}.V'.V$$

The inherited class *Cash* would be:

$$Cash \equiv R.u.\overline{\grave{D}}.\grave{D}.D'.D \mid R.m.\overline{\grave{C}}.\grave{C}.C' + R.c.\overline{\grave{V}}.\grave{V}.V'.V$$

It should be obvious that each sequential term in the expression for the *Payment* object is contained within a selected term of the *Cash* object.

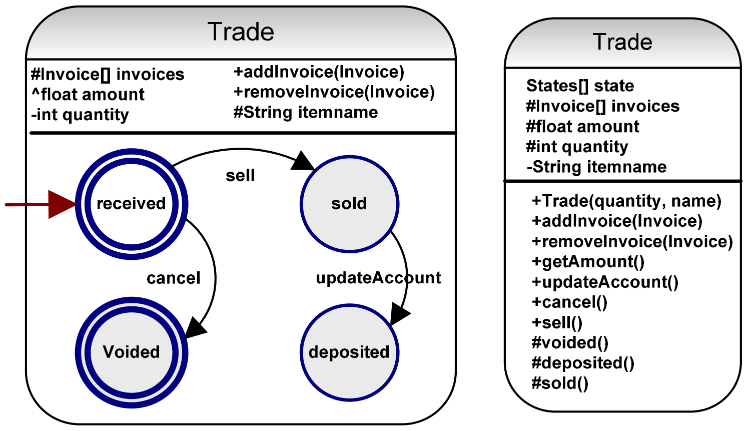


Figure 5.7 Structural Inheritance Only

It should be clear at this point, that the behavioral inheritance concept adds an additional constraint to an object in FOIL before it can be considered to be inherited from another. Figure 5.7 shows an example of an class which complies with the requirement of *interface conformity* as demonstrated by its corresponding UML class specification. Note that all of the attributes and methods of this *Trade* class do exist in the *Payment* class. Thus, by traditional thinking; the *Trade* class could be inherited from *Payment* class; however, from a FOIL perspective, it should be obvious that the behavioral specification of *Trade* does not match that of *Payment*. The algebra also bears this out:

$$\begin{gathered}Payment \equiv R.u.\grave{\overline{D}}.\grave{D}.D'.D + R.c.\overline{V}.V.V'.V \\ Trade \equiv R.s.\grave{\overline{S}}.\grave{S}.S'.S.u.\grave{\overline{D}}.\grave{D}.D'.D + R.c.\overline{V}.V.V'.V\end{gathered}$$

While the second term of these expressions match, the first terms do not. In addition, the first term of the *Payment* expression can not be found embedded in any term in the *Trade* expression. There is a common subsequence between these terms but this is not sufficient to fulfill the requirements for behavioral inheritance. This should be clear from the fact that in order for *Trade* to be inherited from *Payment*, the receipt of an *updateAccount* event while in state *Received* should result in a transition to state *deposited*, but clearly it does not.

5.3 Model Verification

Obviously, one of the major benefits of the Formal Object Interaction Language (FOIL) is the ability to validate models formally. This is done by a special form of state-based analysis using the FOIL algebra. Simple analysis of a FOIL system expression can reveal characteristics about the system as designed or the system during execution. While the extent of what can be learned using this method is less than that of other modeling approaches (such as Petri-nets), the information gleaned is consistent with that required for information system analysis.

5.3.1 *Inherent Inconsistency*

A simple sequential pattern can be used to represent an object behavior that is inconsistent. Figure 5.8 shows an object behavior that is inconsistent. This inconsistency is mainly derived from the fact that this model does not denote the *p* event as occurring multiple times. The system can not accept a *p* event since it will require it later but it

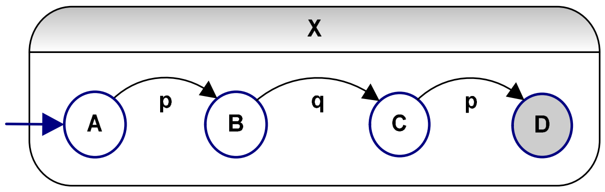


Figure 5.8 Inconsistent Sequential Behavior

can not get to the *C* state which requires it without accepting a *p* event. Thus, there is a contradiction. The simple algebraic representation for this system is:

$$X \overset{CAF}{\equiv} p.q.p.D$$

Clearly based on the Reduction Eligibility Rule, the only term in this expression is eliminated since it begins with a *p* event but has a *p* event embedded in it as well. This leaves object *X* with no valid events for which it may perform its behavior. Thus, object *X* can be said to have no behavior and thus it is no use as modeled. The term used in FOIL to describe this condition is "Inherently Inconsistent".

Figure 5.9 shows an example of the same *X* object but with the added notation that event *p* is allowed to occur multiple times. The algebraic construction now becomes:

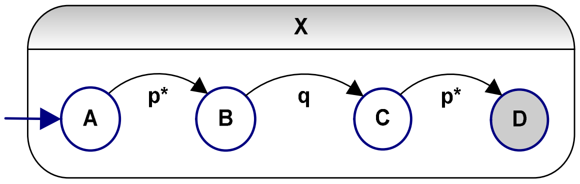


Figure 5.9 Sequential with Plural Events

$$X \overset{CAF}{\equiv} p_1.q.p_2.D$$

Each starred event is numbered upon expansive construction. Now it is clear that a *p* event will be processed if the occurrence of that valid event is numbered. Since event p_1 does not appear in the downstream sequence the Reduction Eligibility Rule is not violated. Thus, the behavior of object *X* expressed in Figure 5.9 is consistent.

5.3.2 Deadlocks

The ability to identify inherent inconsistencies in a model also allows for the detection simple deadlocks. Figure 5.10 shows an example of a simple deadlock. In this case, object *X* must be in state *C* before a *p* event will be accepted but it must be in state *B* before a *q* event will be accepted. Therefore, a deadlock condition exists. Algebraically Figure 5.10 would be constructed as:

$$A = p.B \mid q.C$$
$$B = q.D \text{ and } C = p.E$$
$$\Rightarrow A = p.q.D \mid q.p.E$$

It is easy to see that there are no eligible terms for reduction since all starting events are embedded in other concurrent action sequences. Thus, when an object is represented such that no eligible events exist, the algebra inherently detects the deadlock condition.

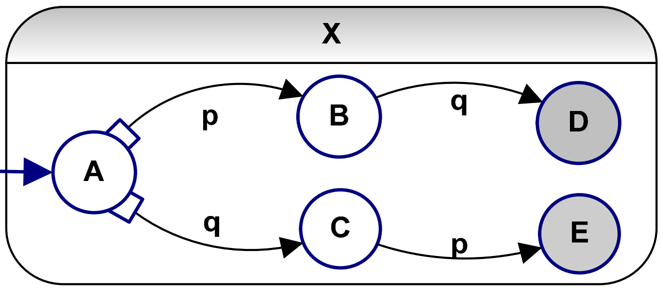


Figure 5.10 Simple Synchronization Deadlock

5.3.2.1 Deadlock Possibility

Figure 5.11 shows a deadlock scenario where object *W* and object *X* are sharing access to objects *Y* and *Z*. The algebraic expression for Figure 5.11 without state flow is:

$$S \overset{CAF}{\equiv} Wp.Wr.Ws.Wq.A \mid Xr.Xp.Xq.Xs.E \mid p.q.I \mid r.s.K$$

FOIL algebra can be used to find possible deadlocks. This is done by placing the model in CAF with only global event scope and determining what global events are eligible. Removing event scope in *S* produces:

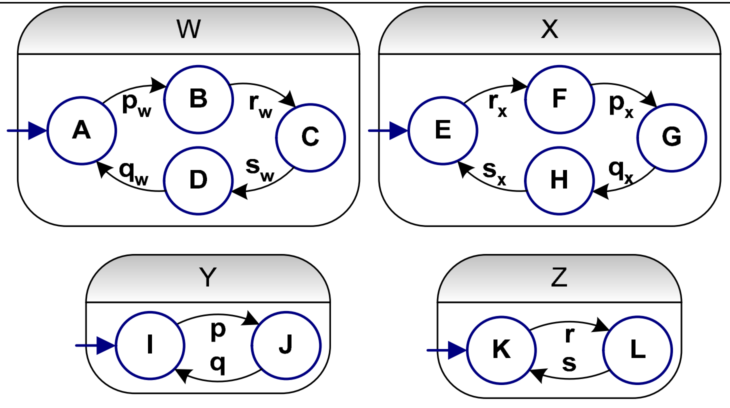


Figure 5.11 Deadlock Example

$$S \overset{GCAF}{\equiv} p.r.s.q.A \mid r.p.q.s.E \mid p.q.I \mid r.s.K$$

An attempt to determine the eligible events will result in an empty set since both *p* and *r* are embedded in other concurrent terms. Thus, this system can result in a deadlock.

5.3.2.2 Deadlock Occurrence

FOIL algebra also provides a mechanism to determine if a system is deadlocked. This is done similarly to deadlock avoidance but during the runtime reduction of events. It is easy to see in Figure 5.11 that a deadlock will result if a local *p* event is received for *W* and a local *r* event is received for *X*. The following reductions show this process:

$$\begin{aligned} S &\xrightarrow{Wp} Wr.Ws.Wq.Wp.B \mid Xr.Xp.Xq.Xs.E \mid q.p.J \mid r.s.K \\ &\xrightarrow{Xr} Wr.Ws.Wq.Wp.B \mid Xp.Xq.Xs.Xr.F \mid q.p.J \mid s.r.L \end{aligned}$$

Once again, an attempt to determine eligible events will result in an empty set meaning that the system can no longer accept any events. The system is deadlocked.

5.3.3 Reachability

Determining whether states are reachable after design or during run-time is nearly as simple as deadlock detection. Figure 5.12 shows an example of an object that has an unreachable state as designed as well as the potential for an unreachable state during execution. The algebraic expression for this object with partial state flow is:

$$X \equiv p.\overline{B}.B.(q.\overline{D}.D.(r.\overline{G}.G + t.\overline{F}.F) + s.\overline{E}.E) \mid s.\overline{C}.C.r.\overline{D}.D.(r.\overline{G}.G + t.\overline{F}.F)$$

$$\begin{aligned} X \overset{CAF}{\equiv}\ & p.\overline{B}.B.q.\overline{D}.D.r.\overline{G}.G \mid s.\overline{C}.C.r.\overline{D}.Dr.\overline{G}.G + p.\overline{B}.B.q.\overline{D}.D.r.\overline{G}.G \mid s.\overline{C}.C.r.\overline{D}.D.t.\overline{F}.F \\ & + p.\overline{B}.B.q.\overline{D}.D.t.\overline{F}.F \mid s.\overline{C}.C.r.\overline{D}.Dr.\overline{G}.G + p.\overline{B}.B.q.\overline{D}.D.t.\overline{F}.F \mid s.\overline{C}.C.r.\overline{D}.D.t.\overline{F}.F \\ & + p.\overline{B}.B.s.\overline{E}.E \mid s.\overline{C}.C.r.\overline{D}.Dr.\overline{G}.G + p.\overline{B}.B.s.\overline{E}.E \mid s.\overline{C}.C.r.\overline{D}.D.t.\overline{F}.F \end{aligned}$$

Removing inherently inconsistent terms produces:

$$X \equiv p.\overline{B}.B.q.\overline{D}.D.t.\overline{F}.F \mid s.\overline{C}.C.r.\overline{D}.D.t.\overline{F}.F + p.\overline{B}.B.s.\overline{E}.E \mid s.\overline{C}.C.r.\overline{D}.D.t.\overline{F}.F$$

There are only two concurrent terms remaining and states *A* and *G* are missing. State *A* is the current state of the object, thus state *G*, from the outset, is unreachable. This can be done during runtime as well. If the above system were to receive an *s* event, the reduction would be:

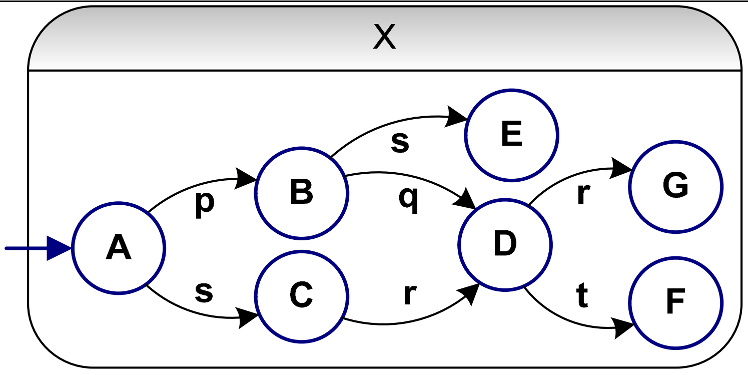


Figure 5.12 Reachability Analysis

$$X \xrightarrow{s} p.\overline{B}.B.q.\overline{D}.D.t.\overline{F}.F \mid r.\overline{D}.D.t.\overline{F}.F$$

Since the second term was ineligible, that choice was eliminated and only the single term remains. In addition, states *A*, *G*, and *E* (the system is currently in state *C*) are no longer in the expression, thus they are all unreachable as this point in execution.

5.4 Russian Philosopher Problem

One of the most popular problems in computer science, the Dining Philosopher Problem, is used to teach and demonstrate the problem of concurrency and resource dependency in computer systems. The problem poses that there are five philosophers sitting around a circular table. Each philosopher has a bowl of rice and a chopstick on their left. In order to eat the rice, each philosopher must pick up the chopstick on their left and their neighbors' chopstick on their right. Each philosopher is thinking independently and when he is done thinking he will eat. The goal is to design a system where no philosopher starves.

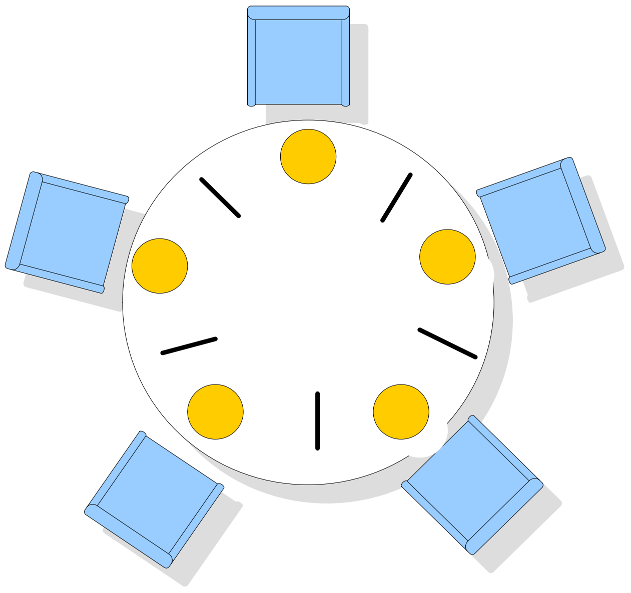


Figure 5.13 Dining Philosopher Problem

A typical solution to this problem is to have each philosopher, when done thinking, pick up the chopstick on his left, then pick up the chopstick on his right, and then eat. When finished, he will put down his left and then his right chopstick sequentially and start thinking again. If the philosopher can not pick up a chopstick because it is being used by another, then he must wait until the chopstick become available. The problem with this scenario occurs if all philosophers begin to eat at the same time. Each one picks up his left chopstick and thus there is no right chopstick for any of them. Thus, they all wait. There are several solutions available to solve this problem but it is not the goal of this paper to explore them.

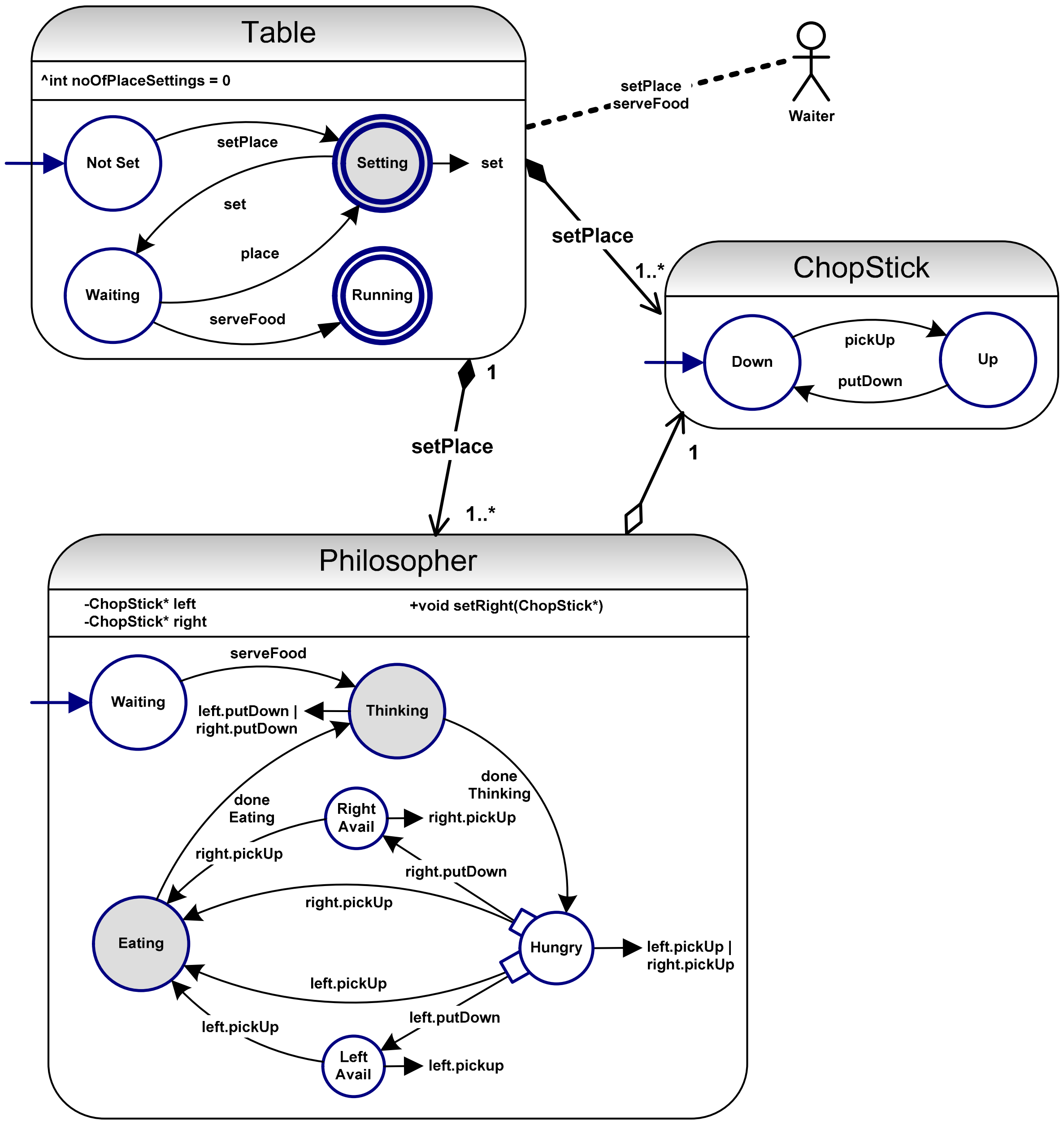


Figure 5.14 FOIL Dining Philosopher Model

Figure 5.14 shows a FOIL model for the Dining Philosopher problem. Immediately, it should be obvious that this model is different from traditional solutions. Since FOIL has support for concurrency, the picking up of chopsticks has been modeled as a concurrent action. To reiterate, if one of the benefits of OO modeling is that it most closely resembles the real world, then this model is more accurate, as most would agree that picking up both chopsticks at the same time is most likely how a person would do it. This deviation from the traditional model does not actually solve the deadlock problem; it merely makes it less likely.

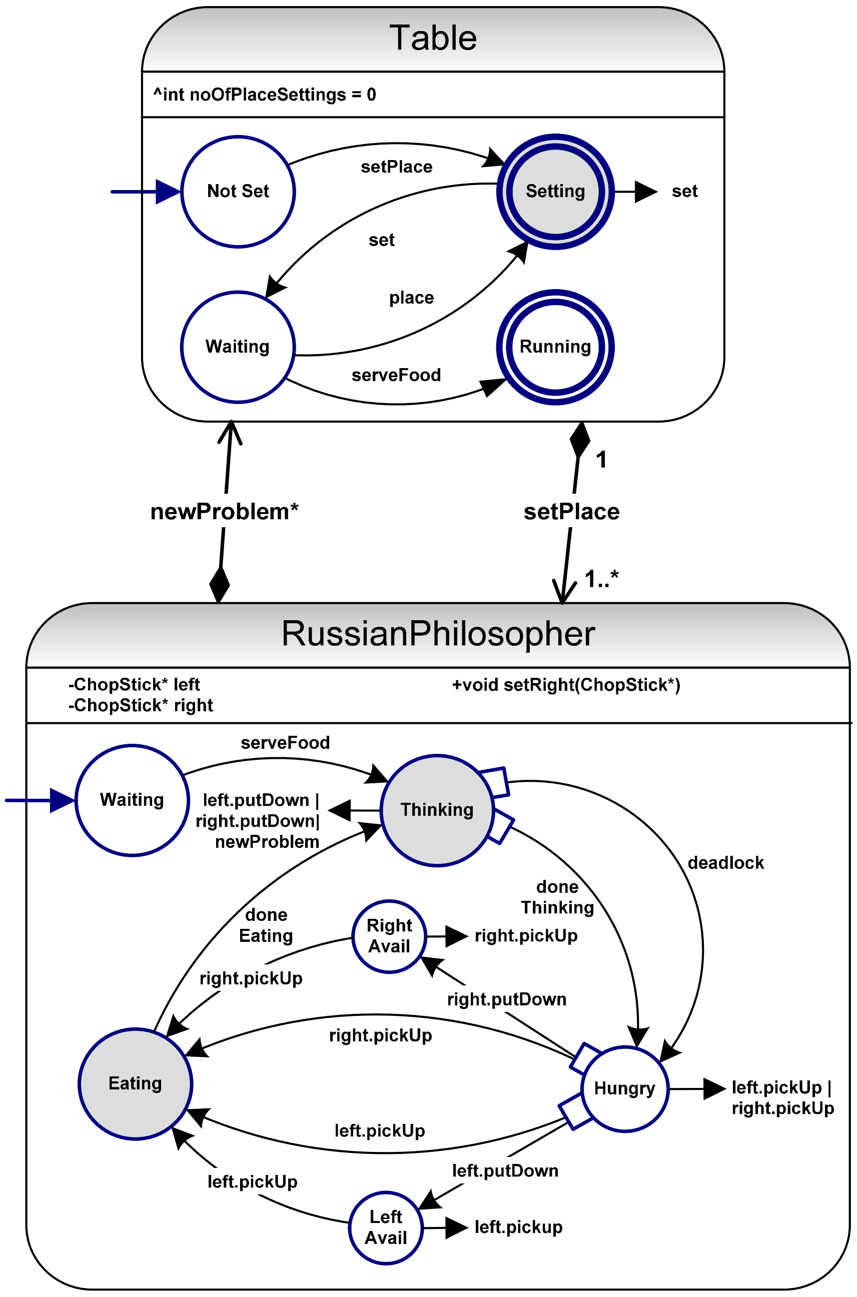


Figure 5.15 FOIL Russian Philosopher

The Russian Philosopher Problem is an extension of the classic Dining Philosopher Problem. This extension is used to add a level of hierarchy to the model. In the Russian Philosopher Problem each “Russian” philosopher is thinking of a Dining Philosopher problem. A Russian Philosopher eats only when the Dining Philosopher table deadlocks. It is simple to see that a Russian Philosopher “is a” Dining Philosopher. Figure 5.15 shows the Russian Philosopher class as modeled in FOIL. There are two places where concurrent extension is used to ensure both structural and behavioral inheritance: the *newProblem* event was added to fire concurrently when the *RussianPhilopher* is *doneEating* and complete transition to *Hungry* will not occur until both the *doneThinking* and *deadlock* events are received.

6. WORKFLOW PATTERNS

The Formal Object Interaction Language (FOIL) can model any system that can be modeled in UML, while providing more information about object behavior. In addition, it supports concurrency, resource dependency, and structural and behavioral inheritance. These models are verifiable through the FOIL algebra providing a formal underpinning much like Petri-nets. This makes FOIL a powerful modeling tool for object-oriented software development.

FOIL can also be used to model high-level processes. These processes can be verified using FOIL algebra to ensure that the underlying object model can perform the overarching process (see Chapter 7). However, modeling from an object or process perspective requires that any underlying framework be complete. The term "complete" refers to the ability to represent all known process or workflow patterns. The composition of a list of patterns is a well studied problem [31] and the current list of these patterns is generally considered to be complete. All complex processes or workflows can be composed of one or more patterns from this list.

This chapter outlines how every workflow pattern can be represented in FOIL both graphically and algebraically. When certain interesting run-time situations are presented by these patterns, an additional demonstration of how FOIL algebra handles such occurrences may be provided. All of the patterns shown use non-active states, unless the fact that states perform code, has an effect execution of the pattern. In some cases, the algebraic reductions will include the state indicators while, for simplicity, others may not.

6.1 Basic Control Patterns

The simplest class of patterns found for processing work deal with simple control. The basic control patterns address simple issues such as task processing in series or parallel and making choices about which tasks will be performed. Parallel processing in the basic sense is always considered to be synchronous.

6.1.1 Sequence

The simplest pattern found in standard workflow implementation is the sequence. In a sequence, the object proceeds from one state to another in a sequential fashion. In this case, an object will never be in multiple states and thus it is completely deterministic in nature.

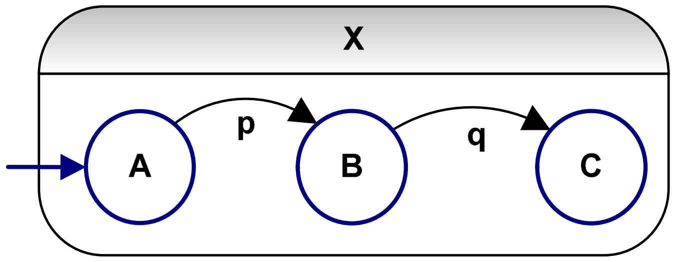


Figure 6.1 Sequence Pattern

Figure 6.1 shows an example of the sequence pattern. When object *X* is instantiated, it begins in state *A*. Upon the receipt of a *p* event designated for the *X* object, it will transition to state *B*. Upon the receipt of a designated *q* event, the *X* object will transition to state *C*. Once arriving at state *C*, no further behavior can be performed on the object making it eligible for deletion.

As might be expected the algebra for this pattern as well as the execution of the events outlined above is simple:

$$\begin{aligned} &X_A \equiv p.B.q.C \\ &\xrightarrow{p} X_B \equiv q.C \\ &\xrightarrow{q} X_c \end{aligned}$$

6.1.2 *Parallel Split*

In order to adequately express the behavior of an object, multiple threads of execution may be required. The parallel split represents a simple situation where multiple threads of execution are enacted. Thus, an object after a parallel split may be in multiple states simultaneously. This is analogous to *non-deterministic finite automata*.

Figure 6.2 shows the simplest example of the parallel split pattern. The output port ensures that object *X* will remain in state *A* until both a *p* and a *q* event have been received. Thus, when a *p* event is received object *X* will be in two states, that being state *A* and state *B*, simultaneously. If the threads were to continue from state *B* and *C* then each thread would execute concurrently.

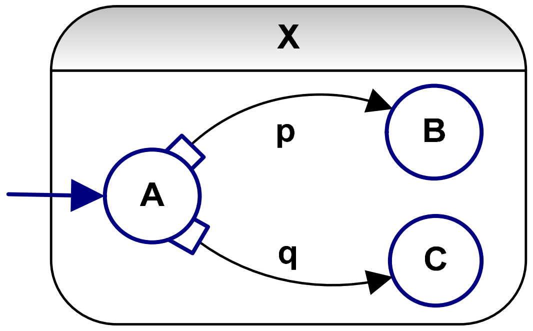


Figure 6.2 Parallel Split

The following is the object *X* expression construction:

$$X_A \equiv p.B \mid q.C$$

Thus, there are two concurrent action sequences that must be followed before the entire flow is complete. Note that a reduction upon receipt of event *p* would result in:

$$X_A \xrightarrow{p} B \mid q.C$$

6.1.3 *Synchronization*

Synchronization refers to the idea that one thread of execution must wait for a parallel process to reach a proper state before accepting the next event. This should not be confused with a merge (see 6.2.2) as in this case both threads of execution will continue independently. It merely suggests that each thread must be in a certain state before either thread can continue.

Figure 6.3 shows a diagrammatic example of an object behavior which requires synchronization. Note that event *q* shows up twice in the diagram. If these occurrences had been represented by a q* then no synchronization would be required since multiple *q* events would be expected. However, this was not done and thus only a single *q* event is expected. When an event *q* is received it is expected that object *X* will transition from state *B* to state *D* and concurrently transition from state *A* to state *C*; however, object *X* must be in state *B* already. Thus, an event *q* is not eligible unless an event *p* has already been received.

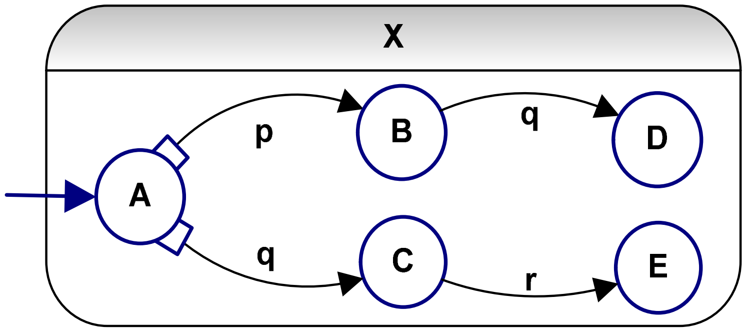


Figure 6.3 Synchronization Pattern

This demonstrates the robustness of the FOIL algebra and the utility of the reduction eligibility rule. Inherently, events that are assumed to occur once must be synchronized. This unique event synchronization is automatically enforced by the algebra. Figure 6.3 can be constructed as follows:

$$A \equiv p.B \mid q.C \qquad B = q.D \qquad C = r.E$$
$$X_A \overset{CAF}{\equiv} p.B.q.D \mid q.C.r.E$$

According to the reduction eligibility rule the only term in the expression for object *X* that is eligible for reduction is *p.B.q.D* since while the second concurrent action starts with a *q* it also appears embedded in the other concurrent action. It is rather simple to see that the *q* event does become eligible after a *p* event is received.

$$X_A \xrightarrow{p} X_{A|B} \equiv q.D \mid q.C.r.E$$

Now, the *q* event is eligible for reduction. Thus, the algebra by way of the reduction eligibility rule, enforces synchronization among unique events.

6.1.4 Exclusive Choice

This pattern represents a single choice between one or more transitions. This pattern can also be viewed as directing a particular thread of execution. No new threads of execution are produced during the execution of this pattern.

Figure 6.4 shows an example of the exclusive choice pattern. Note the absence of the output ports which result in additional threads of execution. Without output ports only the single thread that started object *X* in state *A* will be executed upon either a *p* or a *q* event. It is also important to understand that object completion does not require that all final states be reached. In the case of Figure 6.4, either state *B* or state *C* will be reached but not both; and, in either case, the object has finished its behavior.

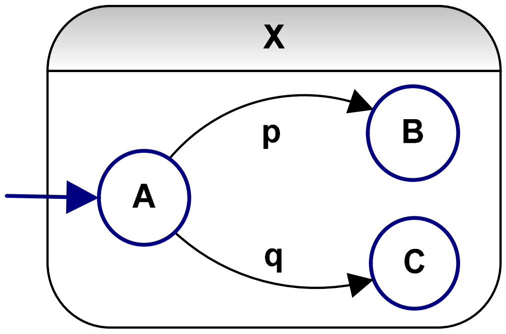


Figure 6.4 Exclusive Choice Pattern

$$X_A \equiv p.B + q.C$$
$$X_A \xrightarrow{p} X_B$$
$$X_A \xrightarrow{q} X_C$$

6.1.5 Simple Merge

The simple merge pattern represents the merging of one or more alternate paths. This should not be confused with the merging of threads of execution. In the case of the simple merge, there is only one thread of execution; however, the path of that execution merges with another alternate path.

Figure 6.5 shows an example of the simple merge pattern. The choice made at state *A* causes a single thread to move to either state *B* or *C*. Regardless of this choice, the path of the behavior will merge at state *D*. Once again, merging in this context does not indicate the joining of two concurrent threads of execution but merely refers to the merging of the path for a single thread.

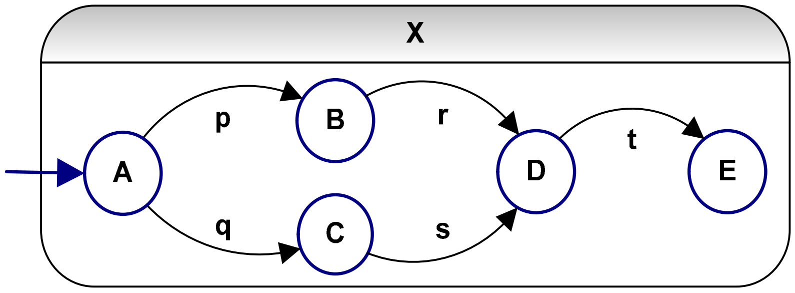


Figure 6.5 Simple Merge Pattern

The algebra for the simple merge is implicit in its construction and is straightforward.

$$A \equiv p.B + q.C \qquad B \equiv r.D \qquad C \equiv s.D \qquad D \equiv t.E$$
$$X_A \equiv p.B.r.D.t.E + q.C.s.D.t.E$$
$$X_A \xrightarrow{p.r} X_D$$
$$X_A \xrightarrow{q.s} X_D$$

The distributive law of choice can be applied to show that states D and E are only executed once.

$$X_A \equiv (p.B.r + q.C.s).D.t.E$$

6.2 Advanced Branching and Synchronization

The power of a modeling language is composed of its ability to model complex patterns while maintaining model simplicity. Many of the patterns in common use in object and process modeling can be composed of series of simple patterns; however, such compositions can grow exponentially resulting in a completely unusable model. Thus, it becomes necessary to ensure that there are simpler notations for more complex patterns.

6.2.1 Multiple Choice

The multiple choice pattern allows for the optional spawning of multiple threads of execution. In other words, it allows for choosing several execution paths from many alternatives.

Figure 6.6 shows an example of the multiple choice pattern along with its associated path merging. In this case, events *p*, *q* and *r* will all spawn a thread of execution but are optional. In this figure, object *X* will remain in state *A* as long as one of the events has not been received. Thus, Figure 6.6 will not complete unless all of the optional events are received. This can be overcome by adding a synchronizing event that will result in completion without receiving all events (see 6.2.3).

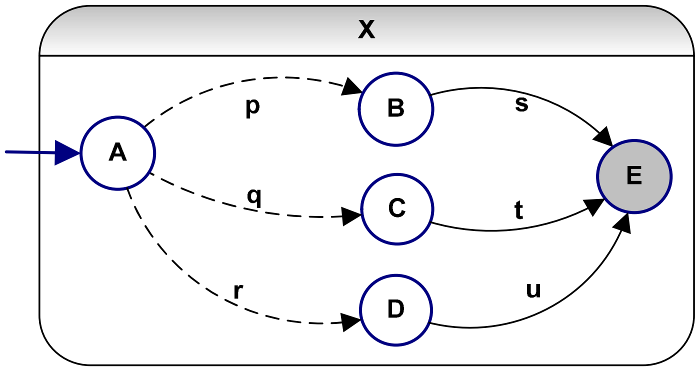


Figure 6.6 Multiple Choice Pattern

It is interesting to note that if all events are received this pattern is the same as the parallel split while if only one event is received it is the same as the exclusive choice. Thus, this construct allows for the range of possibilities between those two patterns inclusively. Additionally, the use of output ports for the transitions out of state *A* are optional since such ports would not change the behavior in any way. Thus, output ports may be added if the spawning of threads from this pattern is not clear.

It is relatively clear that the dotted line represents the possibility that an event may be received. Thus, it is necessary to have an annotation for not receiving an event. While *p* represents the occurrence of the *p* event, a *!p* represents the lack of a *p* event. In the algebra, this functions as a placeholder for manipulating the expressions since it is understood that a *!p* event will never be received. However, some implementations could send a *!p* event explicitly if it is determined that a *p* event will never be received. The algebra will handle this case as well.

Using this notation, the basic definition for the *p* event option of state *A* in is:

$$A \equiv p.B + !p$$

This is read simply as: *A* is defined as receiving an event *p* and acting like *B* or not receiving an event *p* at all. Understanding this, the complete definition of *X* is:

$$A \equiv (p.B + !p) \mid (q.C + !q) \mid (r.D + !r) \qquad B = s.E \qquad C = t.E \qquad D = u.E$$
$$X_A \equiv (p.B.s.E + !p) \mid (q.C.t.E + !q) \mid (r.D.u.E + !r)$$

Through the application of the distributive laws, this definition can be converted to choice-action form.

$$X_A \equiv (p.B.s.E + !p) \mid (q.C.t.E + !q) \mid (r.D.u.E + !r)$$
$$X_A \equiv (p.B.s.E \mid q.C.t.E + p.B.s.E \mid !q + !p \mid q.C.t.E + !p \mid !q) \mid (r.D.u.E + !r)$$
$$X_A \overset{CAF}{\equiv} p.B.s.E \mid q.C.t.E \mid r.D.u.E + p.B.s.E \mid !q \mid r.D.u.E + !p \mid q.C.t.E \mid r.D.u.E + !p \mid !q \mid r.D.u.E + p.B.s.E \mid q.C.t.E \mid !r + p.B.s.E \mid !q \mid !r + !p \mid q.C.t.E \mid !r + !p \mid !q \mid !r$$

This algebra clearly shows the state explosion problem that can be a result of the placing expressions in CAF. In implementation, the underlying system would be better off to place this expression in choice-compressed form (CCF) (see 4.2.2.2). Note that the last choice allows for no events to be received but this case must be executed

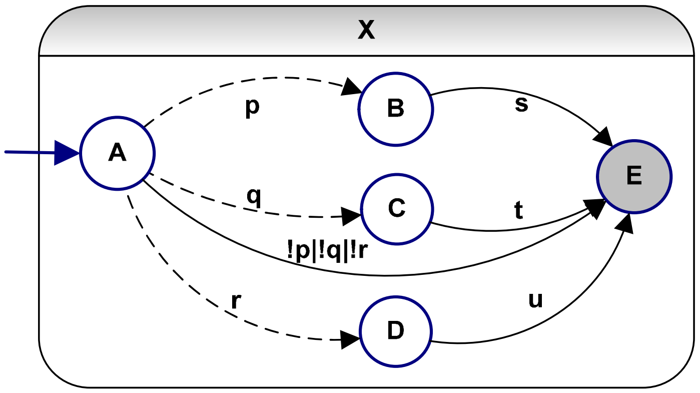


Figure 6.7 No options chosen but continue

explicitly by the firing of events: *!p*, *!q*, and *!r*. This makes sense because the absence of information is not sufficient for the object to determine that it should continue. Also, the final option will result in termination but will not result in arriving at state *E* as may be desired. In order to accomplish this, an additional option may be necessary as shown in Figure 6.7. In this case, the algebra becomes:

$$X_A \overset{CAF}{\equiv} p.B.s.E \mid q.C.t.E \mid r.D.u.E + p.B.s.E \mid !q \mid r.D.u.E + !p \mid q.C.t.E \mid r.D.u.E + !p \mid !q \mid r.D.u.E + \\ p.B.s.E \mid q.C.t.E \mid !r + p.B.s.E \mid !q \mid !r + !p \mid q.C.t.E \mid !r + !p.E \mid !q.E \mid !r.E$$

This allows for transition to state *E* if the system explicitly indicates that no choices will be made.

6.2.2 Synchronizing Merge

In this pattern multiple threads of execution are synchronized and then merged into a single thread of execution. This is distinguished from the simple merge pattern (see 6.1.5), where the paths are merged but only one thread exists, and the discriminator (see 6.2.4), where the threads of execution are merged but are not synchronized.

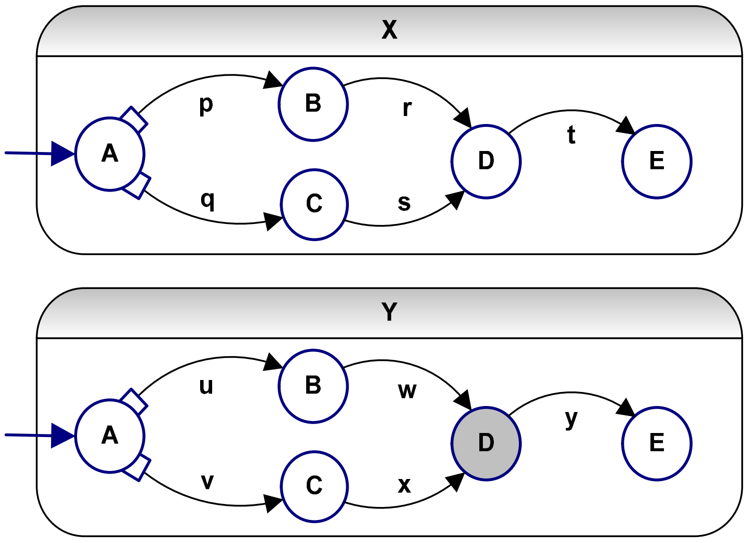


Figure 6.8 Synchronizing Merge Pattern

Figure 6.8 shows an example of the synchronizing merge. State *A* is a parallel split that causes multiple concurrent threads of execution to be spawned. No explicit diagrammatic notations are required to show the synchronous nature of the merge as the synchronizing of the threads occurs implicitly at state *D*. Since only one *t* event is expected, the system implicitly understands that all threads must reach state *D* prior to allowing that event.

The algebraic representation of Figure 6.8 is not much different than what has already been presented in the other patterns.

$$A \equiv p.B \mid q.C \qquad B = r.D \qquad C = s.D \qquad D = t.E$$
$$X_A \overset{CAF}{\equiv} p.B.r.D.t.E \mid q.C.s.D.t.E$$

The main mechanism for synchronous merging is the reduction eligibility rule. As an example, note how the following events affect the expression.

$$\begin{aligned}
&X_A \equiv p.B.r.D.t.E \mid q.C.s.D.t.E \\
&\xrightarrow{p} r.D.t.E \mid q.C.s.D.t.E \\
&\xrightarrow{r} t.E \mid q.C.s.D.t.E \\
&\xrightarrow{t} not\,an\,eligible\,term \\
&\xrightarrow{q} t.E \mid s.D.t.E \\
&\xrightarrow{t} not\,an\,eligible\,term \\
&\xrightarrow{s} t.E \mid t.E \equiv t.E \\
&\xrightarrow{t} E \Rightarrow finished
\end{aligned}$$

Thus, the reduction eligibility rule enforces synchronization. It is also interesting to note that the algebra without modification handles a situation where synchronization is optionally required. Figure 6.9 shows an example diagram of such a situation. In this case, a synchronizing merge will be required at state *D* until a choice is made at state *B*. If an event *t* is received prior to passing state *B* then it is assumed that *u* is the only valid event to transition out of state *B*. The following algebra, without state identifiers, demonstrates this property:

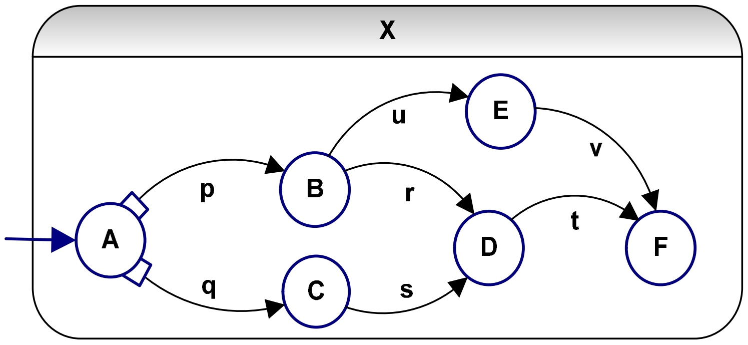


Figure 6.9 Optional Synchronizing Merge

$$
\begin{aligned}
&A \equiv p.B \mid q.C \quad B = u.F + r.D \quad C = s.D \quad D = t.E \quad F = v.E \\
&X_A \equiv p.(u.v.E + r.t.E) \mid q.s.t.E \\
&X_A \overset{CAF}{\equiv} p.u.v.E \mid q.s.t.E + p.r.t.E \mid q.s.t.E \\
&\xrightarrow{q} p.u.v.E \mid s.t.E + p.r.t.E \mid s.t.E \\
&\xrightarrow{p} u.v.E \mid s.t.E + r.t.E \mid s.t.E \\
&\xrightarrow{s} u.v.E \mid t.E + r.t.E \mid t.E
\end{aligned}
$$

At this point a decision will be made on the next valid event. If an event *u* or *t* is received then the first choice will be used. If an *r* event is received then the second choice will be used. This makes sense because the acceptance of the *t* event prior to synchronization precludes *r* as a valid choice out of state *B*.

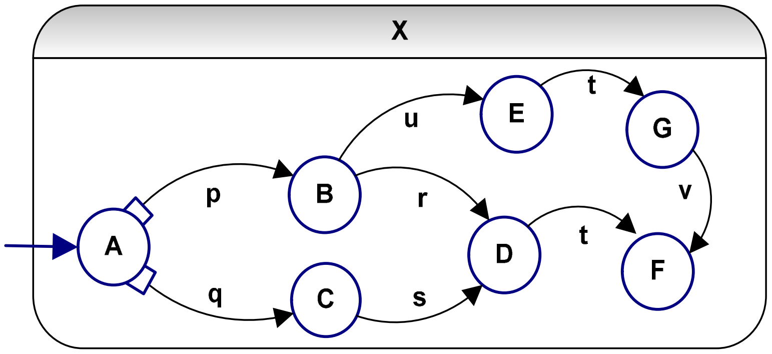


Figure 6.10 Forced Synchronizing Merge

If it is desirable to have the thread wait at state *D* until a choice is made at *B*, a simple use of the synchronization pattern can achieve this as shown in Figure 6.10. In this instance, *E* and *D* require the same event in order to merge at *F*. Thus, if state *C* transitions to *D* by event *s*, an event *t* will still be unaccepted until the thread through state *B* has made a choice and transitioned to either state *E*, where the synchronizing merge will occur at *F*, or state *D* where the synchronizing merge occurs right away. The algebra handles this case without modification and is not shown here.

6.2.3 Multiple Merge

This pattern means that many execution paths are merged without synchronization and multiple threads continue to exist. This does not represent a merging of execution threads but a merging of the path multiple threads will follow. For this reason, this pattern is often referred to as a "Path" merge.

Figure 6.11 shows an example of the multiple-merge pattern. This diagram looks identical to that of the synchronous merge with the exception of the asterisk notation on states *D* and *E* and on event *t*. This asterisk is a multiplicity indicator. Thus, a state marked with an asterisk refers to the fact multiple instances of this state may exist. Likewise, an event marked with an asterisk means that multiple events of this type may be expected.

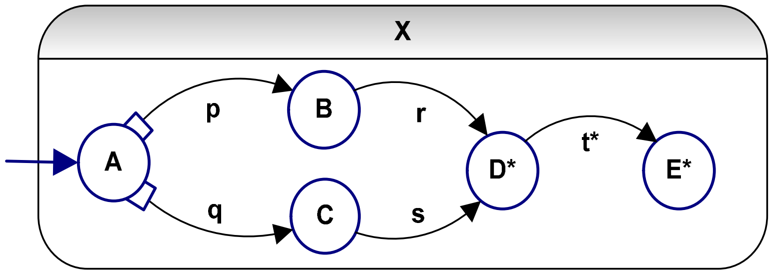


Figure 6.11 Multiple Merge Pattern

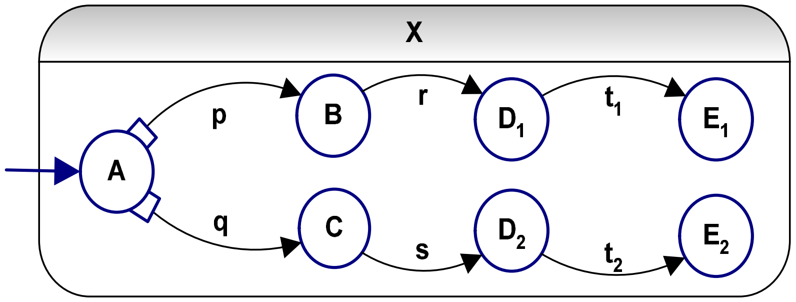


Figure 6.12 Multiple Merge Pattern Alternate Look

Thus, Figure 6.12 indicates that the multiple threads of execution spawned at state *A* will continue even after their paths have merged at state *D*. Since there are multiple state *D*s and the expectation that multiple *t* events will be received then the threads of execution are independent and thus no synchronization is necessary. Figure 6.12 shows another way of looking at the same pattern that may make the function of the asterisks clear. This version of the pattern makes it clear that multiple threads will continue to exist independently but that the same path will be followed by both threads.

During the construction of the algebraic representation of the model in Figure 6.11, each starred item is numbered sequentially as each instance in encountered during expansion. Thus, the algebra for Figure 6.11 is:

$$A = p.B \mid q.C \quad B = r.D_1 \quad C = s.D_2 \quad D_1 = t_1.E_1 \quad D_2 = t_2.E_2$$
$$X_A \equiv p.r.t_1.E_1 \mid q.s.t_2.E_2$$

It is important when using this notation to remember that the numbers do not represent any relation to the actual sequence that these states or events will be reached or received. If either the t_1 and t_2 reductions are eligible when a t event arrives a reduction will occur on that instance. If both are eligible then only one of the instances will be reduced, the choice of which is unimportant. The following sequence of events demonstrates this point.

$$\begin{aligned}
&X_A \equiv p.r.t_1.E_1 \mid q.s.t_2.E_2 \\
&\xrightarrow{p} r.t_1.E_1 \mid q.s.t_2.E_2 \\
&\xrightarrow{q,s} r.t_1.E_1 \mid t_2.E_2 \\
&\xrightarrow{t} r.t_1.E_1 \mid E_2
\end{aligned}$$

The *t* event in this example results in reduction of the second concurrent term despite the fact that this is the first *t* event received but the second term is marked with a subscript of two. The subscript notation is important because, without it, the *t* event would not be accepted at all as it would violate the reduction eligibility rule. Thus, the proper construction of the algebraic notation using subscripts for the starred items results in an expression which can be reduced without any modification to the reduction rules. While it is encouraged that the numbering of starred items be sequential, in actuality the numbering carries no semantic meaning and thus could be arbitrary as long as no two instances have the same subscript.

6.2.4 Discriminator

This pattern is the merging of threads of execution, not a merging of paths. Thus, multiple threads become one thread of execution. The difference is that this merging can be done asynchronously. Therefore, execution of states after the merge is not stopped until the other thread catches up.

Figure 6.13 shows an example of the discriminator pattern. In this example, if a *p* and *r* event is received, the subsequent receipt of event *t* will still be accepted event though the

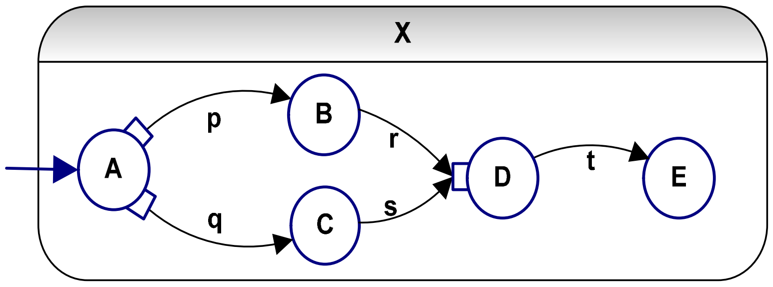


Figure 6.13 Discriminator Pattern

other thread of execution never even reached state *C*.

Algebraically the function of the input port is similar to that of the optional transitions as shown in Figure 6.8. Each one represents a possibility. Thus, state *B* would be represented as:

$$B \equiv r.D + r.0$$

The use of the *.0* term is introduced to explicitly show that a thread will terminate. It is not always necessary to show these, as all final states implicitly have this element. Thus state *D* in Figure 6.13 could really be shown as *t.0* or possibly *t.E.0*, but such explicitness in the algebra would only serve to raise the complexity without improving comprehension. However, in this instance, it is desirable to show the termination since it is not at a final state. Thus, this expression can be read as, "*B* is defined as the receiving of event *r* and then acting like *D* or receiving of an event *r* and then terminating." Given this understanding, the full definition of object *X* in choice-action form is:

$$A \equiv p.B \mid q.C \qquad D = t.E \qquad B = (r.D + r.0) \qquad C = (s.D + s.0)$$
$$X_A \equiv p.(r.t.E + r.0) \mid q.(s.t.E + s.0) = (p.r.t.E + p.r.0) \mid (q.s.t.E + q.s.0)$$
$$X_A \overset{CAF}{\equiv} p.r.t.E \mid q.s.t.E + p.r.t.E \mid q.s.0 + p.r.0 \mid q.s.t.E + p.r.0 \mid q.s.0$$

The final term in this definition was dropped since it is completely encompassed by the other terms. Thus, there is no option to accept events *p*, *r* and events *q*, *s* and then terminate completely as other choices have yet to be resolved. The following demonstrates the behavior with events *p*, *q*, *s*, and *t*:

$$\begin{aligned}
&X_A \overset{CAF}{\equiv} p.r.t.E \mid q.s.t.E + p.r.t.E \mid q.s.0 + p.r.0 \mid q.s.t.E \\
&\xrightarrow{p} r.t.E \mid q.s.t.E + r.t.E \mid q.s.0 + r.0 \mid q.s.t.E \\
&\xrightarrow{q} r.t.E \mid s.t.E + r.t.E \mid s.0 + r.0 \mid s.t.E \\
&\xrightarrow{s} r.t.E \mid t.E + r.0 \mid t.E \\
&\xrightarrow{t} r.0 \mid E
\end{aligned}$$

The reductions of events *p* and *q* are trivial. The reduction for event *s* is also trivial except to note that the second term loses one of its terms since the terminating .0 is reached and the remaining concurrent action can be dropped by the Law of Redundancy (see 4.2.1.3). The most interesting reduction is the *t* event. The first choice has a *t* embedded in its first concurrent action so it violates the reduction eligibility rule. The only acceptable choice is the second term. After reduction it is clear that state *E* has been reached but that completion of the behavior can not occur until the *r* event has been received. Thus, an asynchronous merge has occurred assuring that state *E* will not be executed twice.

6.2.5 N-out-of-M Join

An alternate way of modeling the discriminator pattern of Figure 6.13 is to treat an input port as a form of sequential interleaving. In the case of Figure 6.13, state *B* would be interpreted as having a simple transition to state *D*, since it has an input port. It then becomes the responsibility of state *D* to ensure that the input port is satisfied. Thus, the algebra of Figure 6.13 would have a construction of:

$$\begin{aligned}
&A \equiv p.B \mid q.C \quad B \equiv r.D \quad C \equiv s.D \quad D \equiv t.E + 0 \\
&X \equiv p.r.(t.E + 0) \mid q.s.(t.E + 0) \\
&X \equiv (p.r.t.E + p.r.0) \mid (q.s.t.E + q.s.0) \\
&X \overset{CAF}{\equiv} p.r.t.E \mid q.s.t.E + p.r.0 \mid q.s.t.E + p.r.t.E \mid q.s.0 + p.r.0 \mid q.s.0
\end{aligned}$$

Since this alternate representation is logically equivalent, it is no surprise that the final CAF expression is identical. This is not the preferred way of modeling the discriminator pattern because it looks backward into the model, which is not done on any other occasions.

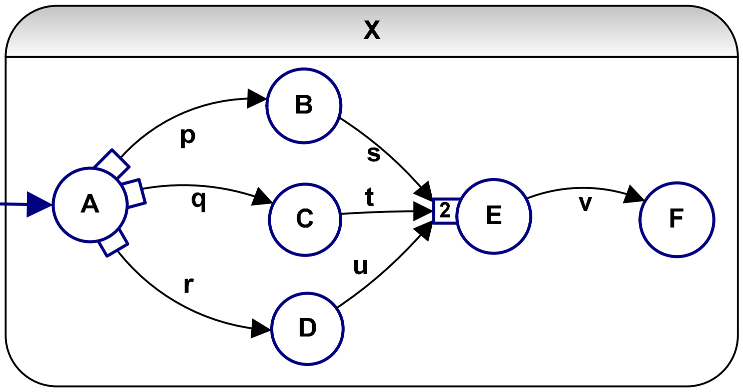


Figure 6.14 N out of M Join

This is the only way, however, of handling the N out of M join pattern. Figure 6.14 shows a simple example of this pattern. This example models the situation where event *v* is not to be accepted until at least two of the three threads have transitioned to state *D*. This pattern can be viewed logically as a complex discriminator with an interleaved condition required for thread continuation. In order to combine these ideas, the alternate form of the discriminator algebra is to be used:

$$A \equiv p.B \mid q.C \mid r.D \quad B \equiv s.E \quad C \equiv t.E \quad D \equiv u.E \quad E \equiv (s+t+u).v.F+0$$

$$X \equiv p.s.((s+t+u).v.F+0) \mid q.t.((s+t+u).v.F+0) \mid r.u.((s+t+u).v.F+0)$$

$$X \equiv p.s.(s.v.F+t.v.F+u.v.F+0) \mid q.t.(s.v.F+t.v.F+ u.v.F+0) \mid r.u.(s.v.F+t.v.F+u.v.F+0)$$

$$X \overset{CAF}{\equiv} p.s.0 \mid q.t.s.v.F \mid r.u.s.v.F + p.s.0 \mid q.t.u.v.F \mid r.u.s.v.F + p.s.t.v.F \mid q.t.0 \mid r.u.s.v.F + p.s.0 \mid q.t.0 \mid r.u.s.v.F + p.s.0 \mid q.t.s.v.F \mid r.u.t.v.F + p.s.t.v.F \mid q.t.0 \mid r.u.t.v.F + p.s.u.v.F \mid q.t.0 \mid r.u.t.v.F + p.s.0 \mid q.t.0 \mid r.u.t.v.F + p.s.u.v.F \mid q.t.s.v.F \mid r.u.0 + p.s.0 \mid q.t.s.v.F \mid r.u.0 + p.s.t.v.F \mid q.t.u.v.F \mid r.u.0 + p.s.u.v.F \mid q.t.u.v.F \mid r.u.0 + p.s.0 \mid q.t.u.v.F \mid r.u.0 + p.s.t.v.F \mid q.t.0 \mid r.u.0 + p.s.u.v.F \mid q.t.0 \mid r.u.0 + p.s.0 \mid q.t.0 \mid r.u.0$$

The expression *(s+t+u)* at the beginning of state *E* is repeated *n-1* times, where *n* is the number represented in the input port notation. Thus, if the input port were to have no number or an explicit one in its notation, then this term would be omitted; or, the construction for the discriminator pattern could be used (see 6.2.4). If the input port notation contained an asterisk, then it would have to wait for all thread to converge before continuing, which is a synchronous join (see 6.2.2). Thus, this pattern covers the range of possibilities between these two patterns.

The following algebraic reductions demonstrate how this pattern would function during run-time:

$$\begin{aligned}
X \xrightarrow{p} {} & s.0 \mid q.t.s.v.F \mid r.u.s.v.F + s.0 \mid q.t.u.v.F \mid r.u.s.v.F + s.t.v.F \mid q.t.0 \mid r.u.s.v.F + \\
& s.0 \mid q.t.0 \mid r.u.s.v.F + s.0 \mid q.t.s.v.F \mid r.u.t.v.F + s.t.v.F \mid q.t.0 \mid r.u.t.v.F + \\
& s.u.v.F \mid q.t.0 \mid r.u.t.v.F + s.0 \mid q.t.0 \mid r.u.t.v.F + \\
& s.u.v.F \mid q.t.s.v.F \mid r.u.0 + s.0 \mid q.t.s.v.F \mid r.u.0 + s.t.v.F \mid q.t.u.v.F \mid r.u.0 + \\
& s.u.v.F \mid q.t.u.v.F \mid r.u.0 + \\
& s.0 \mid q.t.u.v.F \mid r.u.0 + s.t.v.F \mid q.t.0 \mid r.u.0 + s.u.v.F \mid q.t.0 \mid r.u.0 + s.0 \mid q.t.0 \mid r.u.0 \\
\xrightarrow{s} {} & t.v.F \mid q.t.0 \mid r.u.t.v.F + u.v.F \mid q.t.0 \mid r.u.t.v.F + q.t.0 \mid r.u.t.v.F + \\
& t.v.F \mid q.t.u.v.F \mid r.u.0 + u.v.F \mid q.t.u.v.F \mid r.u.0 + q.t.u.v.F \mid r.u.0 + \\
& t.v.F \mid q.t.0 \mid r.u.0 + u.v.F \mid q.t.0 \mid r.u.0 + q.t.0 \mid r.u.0 \\
\xrightarrow{q} {} & t.v.F \mid t.0 \mid r.u.t.v.F + u.v.F \mid t.0 \mid r.u.t.v.F + t.0 \mid r.u.t.v.F + t.v.F \mid t.u.v.F \mid r.u.0 + \\
& u.v.F \mid t.u.v.F \mid r.u.0 + t.u.v.F \mid r.u.0 + t.v.F \mid t.0 \mid r.u.0 + u.v.F \mid t.0 \mid r.u.0 + t.0 \mid r.u.0 \\
\xrightarrow{t} {} & u.v.F \mid r.u.0 + u.v.F \mid u.v.F \mid r.u.0 + u.v.F \mid r.u.0 + v.F \mid r.u.0 + u.v.F \mid r.u.0 + r.u.0 \\
\xrightarrow{v} {} & F \mid r.u.0
\end{aligned}$$

6.3 Structural Patterns

These patterns involve the structural aspects of process control flow, not the structural aspects of objects. As such, it is similar to control flow statements or activities found in modern programming languages. FOIL has little trouble representing these patterns.

6.3.1 *Arbitrary Cycles*

This is a basic looping construct. This pattern is primarily supported by the manner in which the expression is constructed in FOIL. Recall that a unique event is defined as occurring only once per iteration. Thus, by looping a unique event can occur multiple times. Figure 6.15 shows an example of an arbitrary cycle. Event *p* is a unique event and thus occurs only once per iteration. Iteration, in this example, is triggered by an event *r* while in state *B*. Algebraically, the fact that substitution of terms is done only when unexpanded state terms reach the front of an expression is what allows for this behavior:

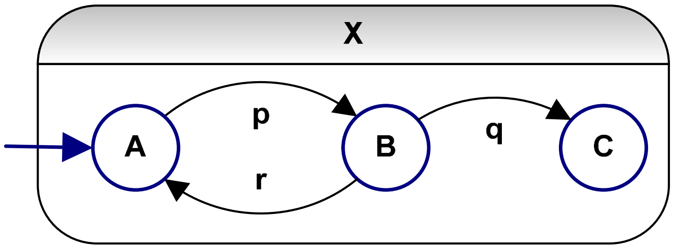


Figure 6.15 Arbitrary Cycle

$$A \equiv p.B \qquad B \equiv q.C + r.A$$
$$X_A \equiv p.(q.C + r.A)$$
$$X_A \overset{CAF}{\equiv} p.B.q.C + p.B.r.A$$
$$\xrightarrow{p} X_B \equiv q.C + r.A$$
$$\xrightarrow{r} X_A \equiv p.B.q.C + p.B.r.A$$

6.3.2 Implicit Termination

This pattern represents a system, process or object implicitly terminating when there is nothing left to do. This pattern is so intuitive that is have been used throughout this these with little explanation. Figure 6.16 shows two examples of this pattern. Analytically, an object or process is said to terminate when all remaining states are unreachable (see 5.3.3). Algebraically, implicit termination can be explicitly represented:

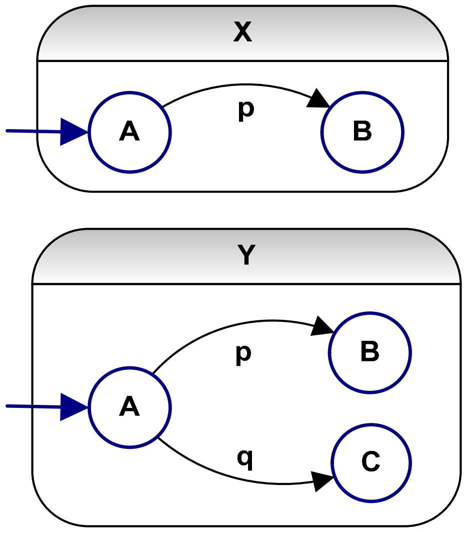


Figure 6.16 Implicit Termination

$$\begin{aligned} &X_A \equiv p.B \qquad X_B \equiv 0 \\ &X_A \equiv p.B.0 \\ &\xrightarrow{p} X_B \equiv 0 \Rightarrow \text{terminated} \end{aligned}$$

This explicitness is usually not necessary but can be helpful in understanding the behavior. For example, note that termination of an object does not necessarily mean that all states have been touched or all final states have been reached:

$$\begin{aligned} &Y_A \equiv p.B + r.C \qquad X_B \equiv 0 \qquad X_C \equiv 0 \\ &Y_A \equiv p.B.0 + r.C.0 \\ &\xrightarrow{r} X_C \equiv 0 \Rightarrow \text{terminated} \end{aligned}$$

Finally, implicit termination does not just refer to a process or object but could refer to a single thread of execution. Algebraically, such representation will always be explicit while graphically it may not. For an example of this refer the discriminator pattern (see 6.2.4).

6.4 Patterns Involving Multiple Instances

An object-oriented modeling language would hardly be useful without the ability to create and manage multiple instances. Interestingly, this same characteristic is used in process modeling to denote multiple copies of a process that run concurrently. With the ability of FOIL to model concurrency, objects can be distributed on multiple systems allowing for each copy of an object to run independently. The following is a review of the main workflow patterns involving multiple instances.

6.4.1 MI without Synchronization

This pattern involves the ability to create multiple instances of objects without requiring synchronization at a future time. In this sense, it is the simplest of the multiple instance patterns. The use of asterisks on event handlers provides a notational indicator that an event may be received multiple times. When used on a relationship between objects, it indicates that an event received by one object will result in the instantiation of another.

Figure 6.17 shows an example of this pattern. When an event *t* is received and object *X* is in an accepting states (states *A* and *B*), then a new instance of object *Y* will be created. Since object *X* and object *Y* have no events in common there is no need for future synchronization. If a global event *r* was received and there were two instance of *Y* then a synchronization condition might result, but if all events for instance of object *Y* are locally specified, no synchronization will occur in this system. The algebraic construction is:

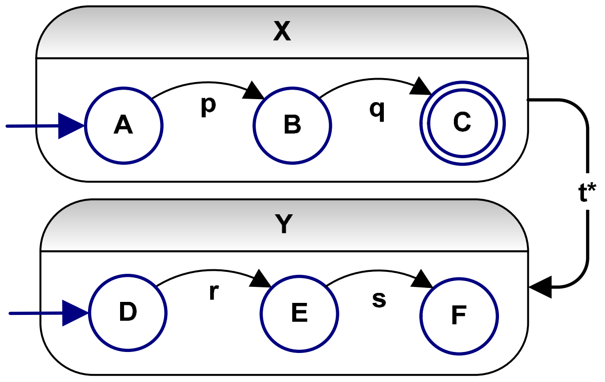


Figure 6.17 MI without Synchronization

$$X_A \equiv p.X_B + t_1.(p.X_B \mid X_A \mid Y) \quad X_B \equiv q.X_C + t_2.(q.X_C \mid X_B \mid Y) \quad X_C = 0$$
$$X \equiv p.(q.X_C + t_2.(q.X_C \mid X_B \mid Y)) + t_1.((q.X_C + t_2.(q.X_C \mid X_B \mid Y)) \mid X_A \mid Y)$$
$$X \equiv p.q.X_C + p.t_2.q.X_C \mid p.t_2.X_B \mid p.t_2.Y + t_1.q.X_C \mid t_1.X_A \mid t_1.Y +$$
$$t_1.t_2.q.X_C \mid t_1.t_2.X_B \mid t_1.t_2.Y \mid t_1.X_A \mid t_1.Y$$

Note that the receiving of an event *t* results in expansion of the expression to include *Y* and an iteration of state *A* of object *X*:

$$Y_D \equiv r.Y_E \quad Y_E \equiv s.Y_F \quad Y_F \equiv 0$$
$$Y \equiv r.s.F$$
$$X \xrightarrow{t} q.X_C \mid X_A \mid Y + t_2.q.X_C \mid t_2.X_B \mid t_2.Y \mid X_A \mid Y$$
$$\equiv q.X_C \mid X_A \mid Y_1 r.Y_1 s.Y_1 F + t_2.q.X_C \mid t_2.X_B \mid t_2.Y \mid X_A \mid Y_1 r.Y_1 s.Y_1 F$$

The expansion of X_A will result in redundancies which can be eliminated based on previous laws; however, this expansion is not shown here as it is a long and relatively trivial exercise.

6.4.2 MI with Priori Design Time Knowledge

This pattern involves the creation of multiple instances where the number of objects created is known at design time. FOIL allows, in addition to the asterisk, the placement of a number to represent the number of times that an event is acceptable. Figure 6.18 shows an example of this pattern. The relationship between class *X* and class *Y* indicates that exactly two instances of object *Y* will be instantiated. Since, all of the states in object *X* are accepting, the exact time of their creation is unknown.

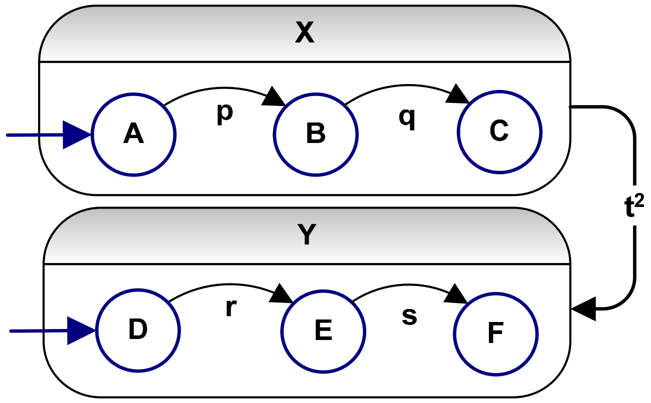


Figure 6.18 MI with *Priori* Design Time Knowledge

The algebraic construction requires that all event *t* results be pre-expanded the specified number of times. The creation of an arbitrary loop resulting from a *t* event would allow for an unbounded number of *Y* objects (see 6.4.1), which is clearly not the intent. Given this, the algebraic construction for each state is:

$$\begin{aligned}
X_A &\equiv p.X_B + t_1.(p.X_B \mid Y_1 r.Y_1 s.Y_1 F) + t_1.(t_2.(p.X_B \mid Y_2 r.Y_2 s.Y_2 F) \mid p.X_B \mid Y_1 r.Y_1 s.Y_1 F) \\
X_B &\equiv q.X_C + t_1.(q.X_C \mid Y_1 r.Y_1 s.Y_1 F) + t_1.(t_2.(q.X_C \mid Y_2 r.Y_2 s.Y_2 F) \mid q.X_C \mid Y_1 r.Y_1 s.Y_1 F) \\
X_C &\equiv t_1.(Y_1 r.Y_1 s.Y_1 F) + t_1.(t_2.(Y_2 r.Y_2 s.Y_2 F) \mid Y_1 r.Y_1 s.Y_1 F)
\end{aligned}$$

Creating full expression and applying the various laws would actually result in:

$$X \equiv p.q.X_C \mid t_1.Y_1 r.Y_1 s.Y_1 F \mid t_2.Y_2 r.Y_2 s.Y_2 F$$

This massive reduction in the size of the expression occurs because all states in object *X* are accepting and thus the creation of the two *Y* objects can occur at any time concurrently with normal behavior of object *X*. In some cases, this behavior may not be desirable.

A more complicated case occurs when state *B* is in a non-accepting state. Thus, there are two instances of object *Y* required but they must be created in one of three ways: both while in state *A*, both while in state *C*, or one in each of states *A* and *C*. Figure 6.19 shows an example of such a case.

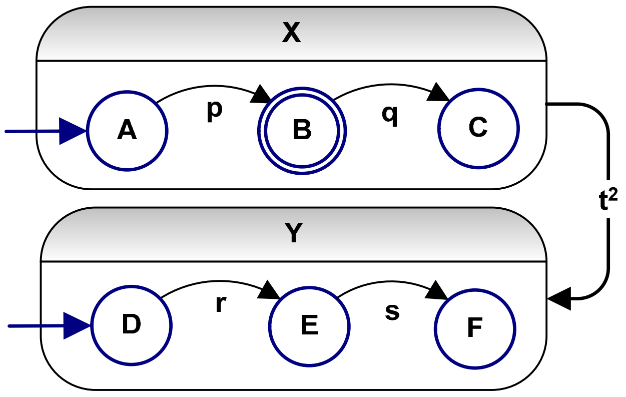


Figure 6.19 MI Creation Restriction

The algebra in this case does not simplify as nicely as the previous.

$$
\begin{aligned}
&X_A \equiv p.X_B + t_1.(p.X_B \mid Y_1 r.Y_1 s.Y_1 F) + t_1.(t_2.(p.X_B \mid Y_2 r.Y_2 s.Y_2 F) \mid p.X_B \mid Y_1 r.Y_1 s.Y_1 F) \\
&X_B \equiv q.X_C \\
&X_C \equiv t_1.(Y_1 r.Y_1 s.Y_1 F \mid C) + t_1.(t_2.(Y_2 r.Y_2 s.Y_2 F \mid C) \mid Y_1 r.Y_1 s.Y_1 F \mid C) + C \\
&X_A \overset{CAF}{\equiv} p.X_B + t_1.p.X_B \mid t_1.Y_1 r.Y_1 s.Y_1 F + t_1.t_2.p.X_B \mid t_1.t_2.Y_2 r.Y_2 s.Y_2 F \mid t_1.p.X_B \mid t_1.Y_1 r.Y_1 s.Y_1 F \\
&X_B \overset{CAF}{\equiv} q.X_C \\
&X_C \equiv t_1.Y_1 r.Y_1 s.Y_1 F \mid t_1.C + t_1.t_2.Y_2 r.Y_2 s.Y_2 F \mid t_1.t_2.C \mid t_1.Y_1 r.Y_1 s.Y_1 F \mid t_1.C
\end{aligned}
$$

Substituting terms and expanding all expressions using the distributive laws:

$$
\begin{gathered}
X \equiv p.q.t_1.Y_1 r.Y_1 s.Y_1 F \mid p.q.t_1.C + p.q.t_1.t_2.Y_2 r.Y_2 s.Y_2 F \mid p.q.t_1.t_2.C \mid p.q.t_1.Y_1 r.Y_1 s.Y_1 F \mid t_1.C \\
+ p.q.C + t_1.p.q.C \mid t_1.Y_1 r.Y_1 s.Y_1 F + t_1.t_2.Y_2 r.Y_2 s.Y_2 F \mid t_1.p.q.C \mid t_1.Y_1 r.Y_1 s.Y_1 F
\end{gathered}
$$

The expanded expression in choice-action form has many terms which are inherently inconsistent or violate one of laws. These concurrent terms are eliminated from the expression to produce a simplified and final CAF expression. This final expression shows that there are actually five choices, not just the three outlined previously. While it is true that there are only three ways to create the two *Y* objects, it is clear from the algebra that this system only limits the number of *Y* objects to three. Inspection of the algebra shows that there is a possibility that zero or one event *t* will be received. So, in this case, the design-time specification of two acceptable *t* events is merely a constraint on the creation of new *Y* objects.

Also, it is obvious from the verbose algebra, that there is still not a defined moment in which the *Y* objects will be created. If it is desired to ensure that exactly two *Y* objects will be created and that they will be created at a certain time, then a different diagram is

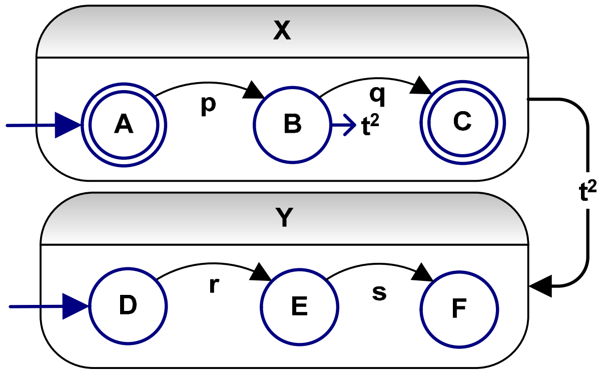


Figure 6.20 MI with Increased Determinism

required, such as Figure 6.20.

6.4.3 MI with Priori Runtime Knowledge

The pattern represents a condition in which the number of objects that will be instantiated for a particular class is not known at design time. In FOIL this particular pattern is actually easier to model than the design-time scenario. Figure 6.21 shows an example of this pattern where at some point prior to state *B*, the number of *t* events that will be fired after state *B* executes is determined. This causes the creation of a fixed number of instances of object *Y* but the exact number is known at some time during execution but not at design time.

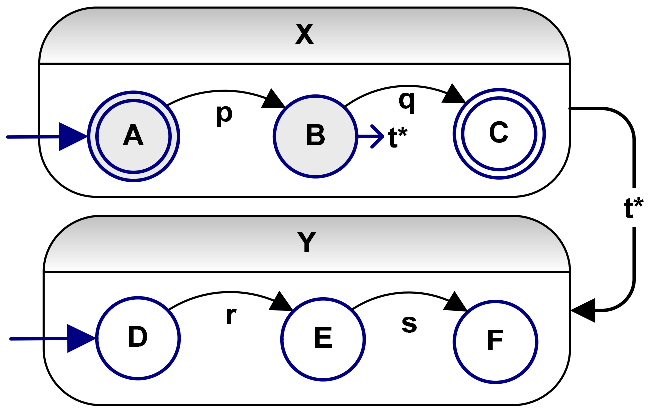


Figure 6.21 MI with *Priori* Runtime Knowledge

$$X_A \equiv p.X_B \qquad X_B \equiv \overline{t*}.(q.X_C + t_1.(q.X_C \mid X_B \mid Y)) \qquad X_C = C$$
$$X \equiv p.(\overline{t*}.(q.C + t_1.(q.C \mid X_B \mid Y)))$$
$$X \overset{CAF}{\equiv} p.\overline{t*}.q.C + p.\overline{t*}.t_1.q.C \mid p.\overline{t*}.t_1.X_B \mid p.\overline{t*}.t_1.Y$$

Initially the only eligible event is *p*, but after reception, an indeterminate number of *t* events will be fired. It is fairly easy to see that each event will result in a new *Y* object. The recursion occurs with the substitution of the X_B term.

6.4.4 MI with no Priori Runtime Knowledge

This pattern results from the system being unaware of exactly how many objects will be instantiated both at design-time and at run-time. This is most likely caused by the system responding to outside events. Since the FOIL modeling language is an event driven approach this particular pattern is extremely simple. Figure 6.22 shows a graphical example of this pattern in FOIL. The algebra is likewise relatively simple:

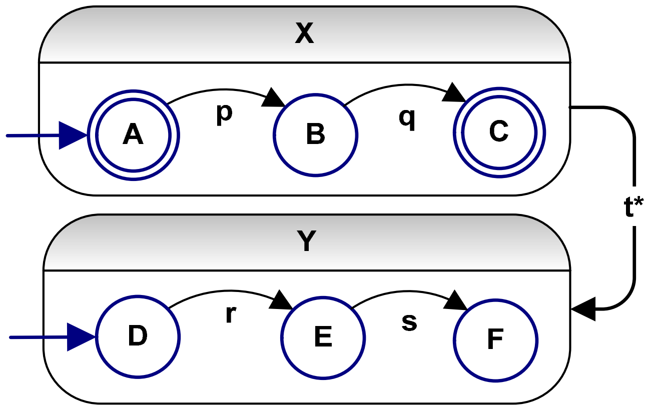


Figure 6.22 MI with no *Priori* Runtime Knowledge

$$X_A \equiv p.X_B \quad X_B \equiv q.X_C + t_1.(q.X_C \mid X_B \mid Y) \quad X_C = C$$
$$X \equiv p.(q.C + t_1.(q.C \mid X_B \mid Y))$$
$$X \overset{CAF}{\equiv} p.q.C + p.t_1.q.C \mid p.t_1.X_B \mid p.t_1.Y$$

This pattern is frequently used in a context of a listening device that will infinitely respond to events. In fact, this pattern has actually already been previously demonstrated with the *MasterController* class in the elevator example (see 3.4).

6.5 State-Based Patterns

This group of patterns is based on the idea that control flow is impacted by system state. In other words, if the system is in a particular state it will force or restrict various choices. Since, FOIL is, at its core, a state-driven modeling language, these patterns are not especially challenging to implement or follow. The only exception is, possibly, interleaved routing which requires special notation to avoid the model growing to an unusable size.

6.5.1 Deferred Choice

The pattern represents that ability of a system to respond to a choice that may not be immediately apparent, but will be determined by future events. FOIL actually depends on this truth in order to allow for the Distributive Law of Choice (see 4.2.1.1). Class *X* of Figure 6.23 shows a simple example of this pattern. The *p* event will result in a transition to either state *B* or state *C*. The absence of any output ports means that only one path can be chosen but the correct transition can not be determined until a subsequent event is received. If event *q* is received than the path to state *B* is chosen. Conversely, state *C* is chosen if the next eligible event received is event *r*. In the simple case, the algebra shows that object *X* would coexist in states *B* or *C* until another event is received.

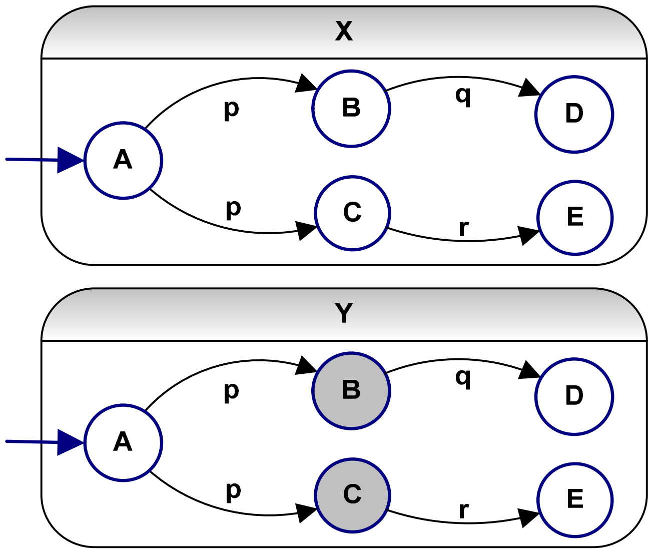


Figure 6.23 Deferred Choice

$$X_A \equiv p.X_B + p.X_C \qquad X_B \equiv B.q.D \qquad X_C \equiv C.r.E$$
$$X \overset{CAF}{\equiv} p.B.q.D + p.C.r.E$$
$$X_A \xrightarrow{p} X_{B+C} \equiv q.D + r.E$$

This particular situation creates difficulties algebraically if state *B* and/or *C* is an active state. While it may be desirable to have both states execute their code and have one thread terminate, this is usually not the intended behavior. Object *Y* of Figure 6.23 shows such an example. In this case, a simple rule can be applied to prevent such occurrences. It is logical to assume that the state execution can not be started by two different choices; hence the algebra can be converted to bring the next event forward in the expression to allow for that choice to be made first.

$$Y_A \equiv p.Y_B + p.Y_C \quad Y_B \equiv \overline{`B}.`B.B'.B.q.D \quad Y_C \equiv \overline{`C}.`C.C'.C.r.E$$
$$Y \overset{CAF}{\equiv} p.\overline{`B}.`B.B'.B.q.D + p.\overline{`C}.`C.C'.C.r.E$$
$$X_A \xrightarrow{p} \overline{`B}.`B.B'.B.q.D + \overline{`C}.`C.C'.C.r.E$$

Since each choice is supposed to fire an execution event to start processing this would result in a race condition as the first event to be received would eliminate the remaining term. In addition to being total unacceptable, it is not logical for concurrent events to fire when no concurrency is warranted. Thus, by moving the next eligible term to the front of each offending expression the decision is postponed.

$$X_A \xrightarrow{p} q.\overline{`B}.`B.B'.B.D + r.\overline{`C}.`C.C'.C.E$$

6.5.2 Interleaved Routing

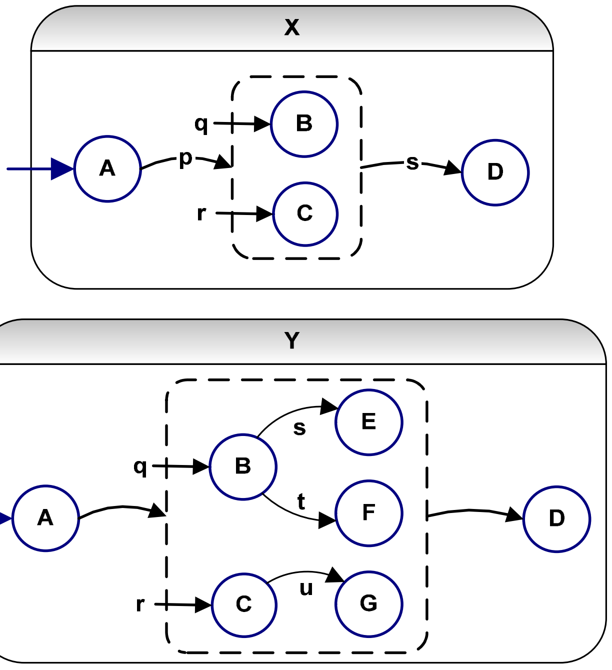


Figure 6.24 Interleaved Routing

This pattern is concerned with sequential operation of multiple control flows, but in no predetermined order. In other words, two or more flows need to be executed but they can not be executed at the same time. The order of execution is unimportant. Figure 6.24 shows a FOIL diagram of a simple interleaved routing situation. Once object *X* receives an event *p*, then either the *q* or *r* events will be exclusively allowed. The algebra constructed by a series of sequential choices where the interleaved construct is considered its own state *I*:

$$\begin{aligned}
&A \equiv p.I \quad I \equiv (q.B + r.C).(q.B + r.C).s.D \\
&X \equiv p.(q.B + r.C).(q.B + r.C).s.D \\
&X \equiv p.q.B.q.B.s.D + p.r.C.q.B.s.D + p.q.B.r.C.s.D + p.r.C.r.C.s.D \\
&X \overset{CAF}{\equiv} p.r.C.q.B.s.D + p.q.B.r.C.s.D
\end{aligned}$$

Substitution of state *B* or *C* in the above example can be expanded to include any independent flow. If the control flows cross in any way, or if they have a dependency on one another, then the algebraic expression would completely cancel out. This would indicate that such a pattern would not function. Object *Y* in the above figure demonstrates a slightly more complicated object control flow with some notational variations.

A transition without an event could be considered to be an automatic transition. In most cases, this is not desirable as such a construct just adds notational complexity without adding any meaning. Object *Y* in Figure 6.24, however, would like to execute two independent sequences one at a time but does not need a starting event to indicate that it wishes to start such a process. In this case, transitioning into or out of an interleaved construct is implicit as the algebra indicates:

$$\begin{aligned}
&A \equiv I \quad I \equiv (q.B + r.C).(q.B + r.C).D \quad B \equiv s.E + t.F \quad C \equiv u.G \\
&Y \equiv (q.(s.E + t.F) + r.u.G).(q.(s.E + t.F) + r.u.G).D \\
&Y \equiv (q.s.E + q.t.F + r.u.G).(q.s.E.D + q.t.F.D + r.u.G.D) \\
&Y \overset{CAF}{\equiv} r.u.G.q.s.E.D + r.u.G.q.t.F.D + q.s.E.r.u.G.D + q.t.F.r.u.G.D
\end{aligned}$$

6.5.3 Milestone

The milestone pattern involves other objects or processes waiting until another has reached a particular event has occurred. Synchronization of objects or processes may or may not occur with this pattern. This is because if all flows are, in fact, waiting on the milestone to be reached, then it is logical to say that when the milestone is reached the flows will be synchronized. If, however, the milestone is reached before affected flows are waiting then no synchronization occurs.

Figure 6.25 shows an example of this pattern. In this example, the assumption is that objects *X*, *Y*, and *Z* are all instantiated and currently in their starting states. The milestone occurs at state *B* of object *X*. Object *Y* is not allowed to proceed past state *F* and object *Z* is not allowed to proceed past state *J* until an event *x* has been received. This event is immediately fired by object *X* upon arriving at state *B*. The use of a concurrent thread in modeling this pattern ensures that synchronization has to occur at Y_F and Z_J respectively, but that event *x* may be received at any time. This example does not prohibit *x* from being fired from outside object *X*, however, such constraints can be applied through the use of event scope if desired.

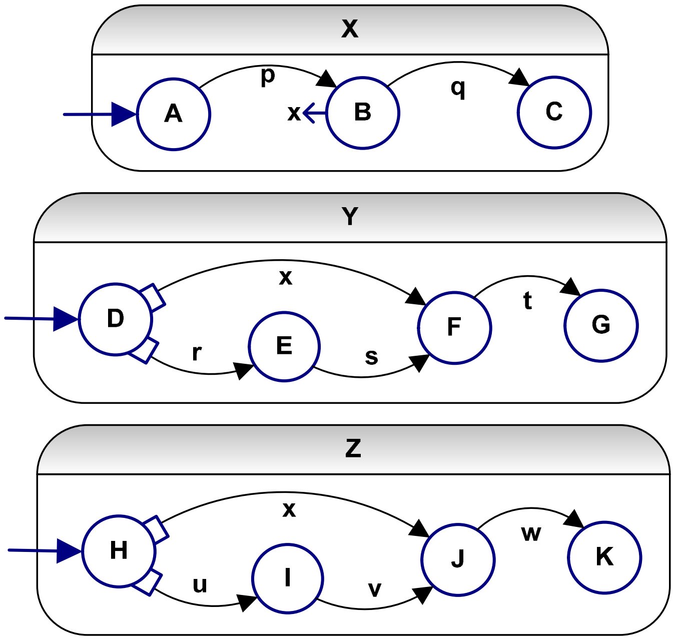


Figure 6.25 Milestone

The following is the algebraic construction with state notation of the system in Figure 6.25:

$$X \overset{CAF}{\equiv} A.p.\bar{x}.B.q.C \qquad Y \overset{CAF}{\equiv} D.r.E.s.F.t.G \mid D.x.F.t.G \qquad Z \overset{CAF}{\equiv} H.u.I.v.J \mid H.x.J.w.K$$

Since these objects each execute independently, the system expression would be:

$$S \overset{CAF}{\equiv} A.p.\bar{x}.B.q.C \mid D.r.E.s.F.t.G \mid D.x.F.t.G \mid H.u.I.v.J \mid H.x.J.w.K$$

The following demonstrates this pattern during run-time:

$$\begin{aligned}
S_{ADH} &\equiv p.\bar{x}.B.q.C \mid r.E.s.F.t.G \mid x.F.t.G \mid u.I.v.J \mid x.J.w.K \\
&\xrightarrow{r} p.\bar{x}.B.q.C \mid s.F.t.G \mid x.F.t.G \mid u.I.v.J \mid x.J.w.K \\
&\xrightarrow{s} p.\bar{x}.B.q.C \mid F.t.G \mid x.F.t.G \mid u.I.v.J \mid x.J.w.K
\end{aligned}$$

At this point during execution, object *Y* can not completely arrive at state *F* since there is still a state *F* term in a concurrent expression. It is clear that object *Y* can not continue until an *x* is received. This will move the remaining *F* state to the front of its concurrent term making it eligible for reduction. Once object *X* arrives at state *B* all restrictions on the objects are removed.

$$\begin{aligned}
&\xrightarrow{p} \bar{x}.B.q.C \mid F.t.G \mid x.F.t.G \mid u.I.v.J \mid x.J.w.K \\
&\xrightarrow{x} B.q.C \mid t.G \mid u.I.v.J \mid J.w.K
\end{aligned}$$

When *x* is fired, object *Y* arrives at state *F* and can continue with processing. Object *Z* was practically unaffected by the receipt of event *x*. It no longer has to synchronize with object *X*. Thus, in this scenario, object *Y* synchronizes at the milestone and object *Z* never does.

6.6 Cancellation Patterns

These patterns, while important, are among the simplest in workflow processing. These patterns involve causing a process or series of processes to stop execution.

6.6.1 Cancel Activity

The cancel activity pattern is simply ensuring that the receipt of an event will cause all processing of an object to cease. It is simple to see how FOIL could implement such a pattern. Figure 6.26 shows an example of this pattern using an optional notation to indicate that the thread terminates. There is no need to actually label the destination state for cancellation; however, in practice this would likely be desired to give an underlying implementation an indication of what state an object is in. Obviously, the modeling must ensure that a cancellation event terminates all concurrent threads of an object regardless of what state the object is in. This concept, while logically simple, can result in a very busy diagram. An alternate notation indicating that all states in the diagram have a choice to transition to the cancelled state could be used but is not provided here.

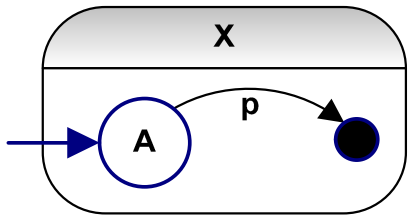


Figure 6.26 Cancel Activity

6.6.2 Cancel Case

This pattern is really just an extension on the previous pattern and ensures that a cancellation causes a group of related objects or processes to all terminate concurrently. Once again, this pattern is no challenge to the FOIL algebra; however, it may be a notational challenge if explicitly modeled, since every state of every object or process involved would require a transition to a cancelled state. In addition, the ability to restart a canceled case can be easy or difficult depending on whether the modeler wants to always restart at the beginning of a process or, instead, desires a restart from the previous object state.

7. PROCESS ANALYSIS

The Formal Object Interaction Language (FOIL), as has already been discussed is capable of modeling process flows as well as object diagrams. FOIL does this using primarily the same notational elements for both models. In addition, both models have a common underlying mathematical representation. Given that two models have an algebraic representation, it is logical that if there exists any intersection in the events received by these models, certain mathematical operations may offer insight into their interaction.

7.1 Process Achievability

The concept of process achievability is centered on the idea that a process "can" be completed given a particular object model. This does not indicate that a process "will" be completed. Since any FOIL object model can be effectively canceled at any time, it can be argued that there is never any guaranty that a process will complete; however, this is not considered as part of the definition:

> A process is said to be "achievable" if during the pursuit of local completion of object workflow on a corresponding object model, a given process has the potential to complete.

Determining the achievability of a process is a useful metric. It can be used to reject object models that can not perform a certain process. In addition, if it is desirable to ensure that a process "will" always complete, achievability metrics can be used to determine the modifications to the object model that are necessary.

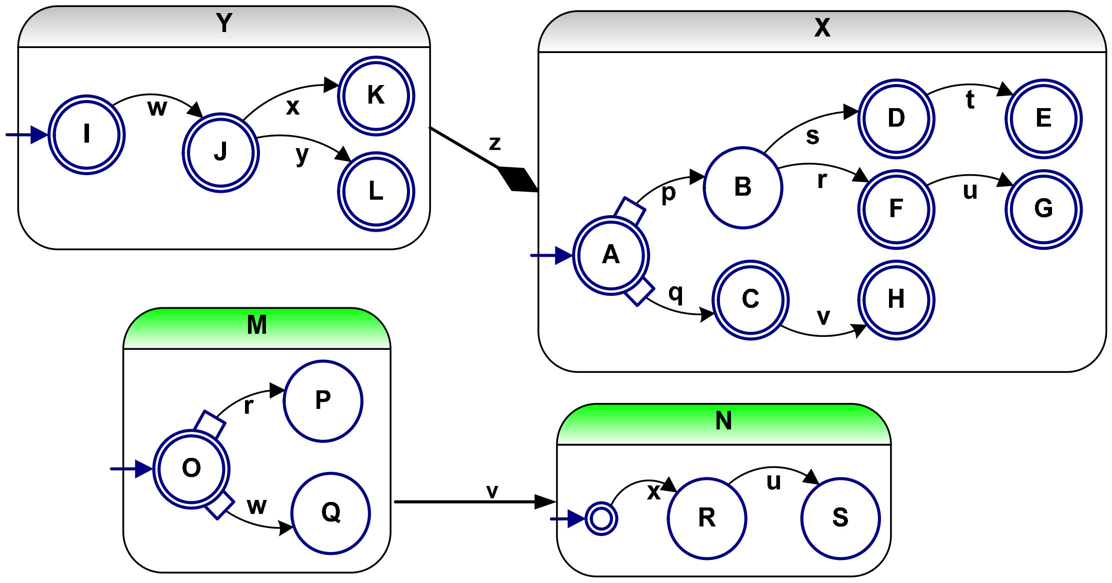


Figure 7.1 Process Achievability

The technique for determining achievability involves a look-ahead simulation of a process on the object model. Figure 7.1 shows a simple FOIL object model (objects *X* and *Y*) and a corresponding FOIL process model (processes *M* and *N*). The algebraic representation of objects *X* and *Y* in choice-action form are:

$$\begin{aligned} X &\overset{CAF}{\equiv} A.p.B.s.D.t.E \mid A.q.C.v.H + A.p.B.r.F.u.G \mid A.q.C.v.H + \\ &\quad A.p.B.z.s.D.t.E \mid A.p.B.z.B_i \mid A.p.B.z.Y_i \mid A.q.C.v.H + \\ &\quad A.p.B.z.r.F.u.G \mid A.p.B.z.B_i \mid A.p.B.z.Y \mid A.q.C.v.H \\ Y &\overset{CAF}{\equiv} I.w.J.x.K + I.w.J.y.L \end{aligned}$$

The algebraic representation of processes *M* and *N* in choice-action form are:

$$\begin{aligned} M &\overset{CAF}{\equiv} O.r.P.v.N_1 \mid O.w.Q.v.N_1 \\ N &\overset{CAF}{\equiv} x.R.u.S \end{aligned}$$

The algorithm for determining achievability uses a simple backtracking technique applied to the process and object expressions. Each eligible process event is placed in a process event stack *U*. The first event is removed from the stack and a search is done to determine which choice terms in the object expression have the event. This event is referred to as the “search” event. Each eligible term in the object expression is assigned a weight proportional to the depth at which the search event occurs and pushed onto a “choice” stack *V* in descending order. The term with the lowest weight is then simulated by popping stack *V* and firing events up to and including the search event. Then, all eligible process events are placed in the stack *U* and the process is repeated. If after each iteration, the process expression is reduced to *0* then the process is achievable. If the both stacks *U* and *V* become empty prior reducing the process expression to *0*, then the process is not achievable.

Using the example of Figure 7.1, the following demonstration is given to determine if the process of *M* and *N* is achievable with the object *X* and *Y*:

$$M \overset{CAF}{\equiv} O.r.P.v.N_1 \mid O.w.Q.v.N_1$$
$$M \xrightarrow{start} r.P.v.N_1 \mid w.Q.v.N_1$$

At this point in process *M* the eligible events are *r* and *w*. These are pushed onto the stack *Z*.

$$U \rhd \{w, r\}$$

Event *w* is popped from the stack and a search is done on the object expression *X* to determine which choice contains an event *w*. This is intuitive since in order for the process to complete, an event *w* must be accepted at some time during the object model workflow.

$$w \bigcap X \equiv \{\}$$

The search for an event *w* in the object event model results in an empty set of terms. Thus, event *w* is not a valid choice and the next term is popped from the stack, in this case *r*.

$$\begin{aligned} V \rhd \{&A.p.B.r.F.u.G \mid A.q.C.v.H : 4, \\ &A.p.B.z.r.F.u.G \mid A.p.B.z.B_i \mid A.p.B.z.Y \mid A.q.C.v.H : 5\} \end{aligned}$$

The search for event *r* in the object expression results in a set of two possible choices for execution. These choices are assigned a weight based on the first appearance of event *r* in their expressions. In this example, there are two choices eligible. The first choice has event *r* appearing as the fourth term, while in the second expression event *r* is the fifth term. A simulation is then run on the object expression by reducing the expression with all events necessary to reduce event *r* starting with the lowest weight choice.

$$\begin{aligned} &V \rhd \{A.p.B.z.r.F.u.G \mid A.p.B.z.B_i \mid A.p.B.z.Y \mid A.q.C.v.H : 5\} \\ &choice \equiv A.p.B.r.F.u.G \mid A.q.C.v.H \\ &X \xrightarrow{create} p.B.s.D.t.E \mid q.C.v.H + p.B.r.F.u.G \mid q.C.v.H + \\ &\qquad p.B.z.s.D.t.E \mid p.B.z.B_i \mid p.B.z.Y_i \mid q.C.v.H + \\ &\qquad p.B.z.r.F.u.G \mid p.B.z.B_i \mid p.B.z.Y \mid q.C.v.H \\ &X \xrightarrow{p} s.D.t.E \mid q.C.v.H + r.F.u.G \mid q.C.v.H + z.s.D.t.E \mid z.B_i \mid z.Y_i \mid q.C.v.H + \\ &\qquad z.r.F.u.G \mid z.B_i \mid z.Y \mid q.C.v.H \\ &X \xrightarrow{r} u.G \mid q.C.v.H \equiv X_F \end{aligned}$$

After the reductions, the process expression is reduced with event *r* as follows:

$$M \equiv v.N_1 \mid w.Q.v.N_1$$

The next eligible events are pushed onto stack *U*. Event *V* violates the eligibility rule and thus is not pushed onto stack *U*.

$$U \rhd \{w\}$$

It is clear that event w in stack U can not be processed with the object expression in its current state. Thus, stack U becomes empty and stack V must be popped to attempt the second choice.

$$\begin{aligned}
&V \rhd \{\} \\
&choice \equiv A.p.B.z.r.F.u.G \mid A.p.B.z.B_i \mid A.p.B.z.Y \mid A.q.C.v.H \\
&X \xrightarrow{create} p.B.s.D.t.E \mid q.C.v.H + p.B.r.F.u.G \mid q.C.v.H + \\
&\qquad p.B.z.s.D.t.E \mid p.B.z.B_i \mid p.B.z.Y_i \mid q.C.v.H + \\
&\qquad p.B.z.r.F.u.G \mid p.B.z.B_i \mid p.B.z.Y \mid q.C.v.H \\
&X \xrightarrow{p} s.D.t.E \mid q.C.v.H + r.F.u.G \mid q.C.v.H + z.s.D.t.E \mid z.B_i \mid z.Y_i \mid q.C.v.H + \\
&\qquad z.r.F.u.G \mid z.B_i \mid z.Y \mid q.C.v.H \\
&X \xrightarrow{z} r.F.u.G \mid s.D.t.E \mid w.J.x.K \mid q.C.v.H + r.F.u.G \mid s.D.t.E \mid w.J.y.L \mid q.C.v.H \\
&\qquad + r.F.u.G \mid s.D.t.E \mid w.J.x.K \mid q.C.v.H + r.F.u.G \mid s.D.t.E \mid w.J.y.L \mid q.C.v.H \\
&X \xrightarrow{r} u.G \mid s.D.t.E \mid w.J.x.K \mid q.C.v.H + u.G \mid s.D.t.E \mid w.J.y.L \mid q.C.v.H \\
&\qquad + u.G \mid s.D.t.E \mid w.J.x.K \mid q.C.v.H + u.G \mid s.D.t.E \mid w.J.y.L \mid q.C.v.H
\end{aligned}$$

As before, the next eligible process events are placed on the stack:

$$U \rhd \{w\}$$

Event w appears in every choice in exactly the same place, so the order in which these choices are pushed onto stack V is unimportant.

$$\begin{aligned}
&V \rhd \{u.G \mid s.D.t.E \mid w.J.x.K \mid q.C.v.H : 3, \\
&\qquad u.G \mid s.D.t.E \mid w.J.y.L \mid q.C.v.H : 3, \\
&\qquad u.G \mid s.D.t.E \mid w.J.x.K \mid q.C.v.H : 3, \\
&\qquad u.G \mid s.D.t.E \mid w.J.y.L \mid q.C.v.H : 3\} \\
&X \xrightarrow{w} u.G \mid s.D.t.E \mid x.K \mid q.C.v.H + u.G \mid s.D.t.E \mid y.L \mid q.C.v.H \\
&\qquad + u.G \mid s.D.t.E \mid x.K \mid q.C.v.H + u.G \mid s.D.t.E \mid y.L \mid q.C.v.H
\end{aligned}$$

The process expression is now:

$$\begin{aligned}
&M \equiv v.N_1 \mid v.N_1 \\
&\Rightarrow U \rhd \{v\}
\end{aligned}$$

This process continues as follows:

$$
\begin{aligned}
V \rhd \{&u.G \mid s.D.t.E \mid x.K \mid q.C.v.H : 3, \qquad && u.G \mid s.D.t.E \mid y.L \mid q.C.v.H : 3 \\
&u.G \mid s.D.t.E \mid x.K \mid q.C.v.H : 3, && u.G \mid s.D.t.E \mid y.L \mid q.C.v.H : 3 \\
&u.G \mid s.D.t.E \mid w.J.y.L \mid q.C.v.H : 3, && u.G \mid s.D.t.E \mid w.J.x.K \mid q.C.v.H : 3, \\
&u.G \mid s.D.t.E \mid w.J.y.L \mid q.C.v.H : 3\}
\end{aligned}
$$

$$X \xrightarrow{q,v} u.G \mid s.D.t.E \mid x.K + u.G \mid s.D.t.E \mid y.L + u.G \mid s.D.t.E \mid x.K + u.G \mid s.D.t.E \mid y.L$$

For simplicity, stack V is not shown in these last steps; however, it should be noted that stack V will continue to grow with each step.

$$
\begin{aligned}
&M \equiv x.R.u.S \mid x.R.u.S \equiv x.R.u.S \\
&U \rhd \{x\} \\
&X \xrightarrow{x} u.G \mid s.D.t.E + u.G \mid s.D.t.E \equiv u.G \mid s.D.t.E \\
&M \equiv u.S \\
&U \rhd \{u\} \\
&X \xrightarrow{u} s.D.t.E \\
&M \equiv 0
\end{aligned}
$$

The process expression M completes and thus this process is achievable with the object model given by X and Y. The same process which is achievable with the object model of Figure 7.1 can be non-achievable with a different object model. Figure 7.2 shows an example of an object model that would not be achievable with the previously defined process model M and N. While it may not be obvious at first glance, intuitively it is simple to see that events r and u are mutually exclusive in object X. Thus, since the

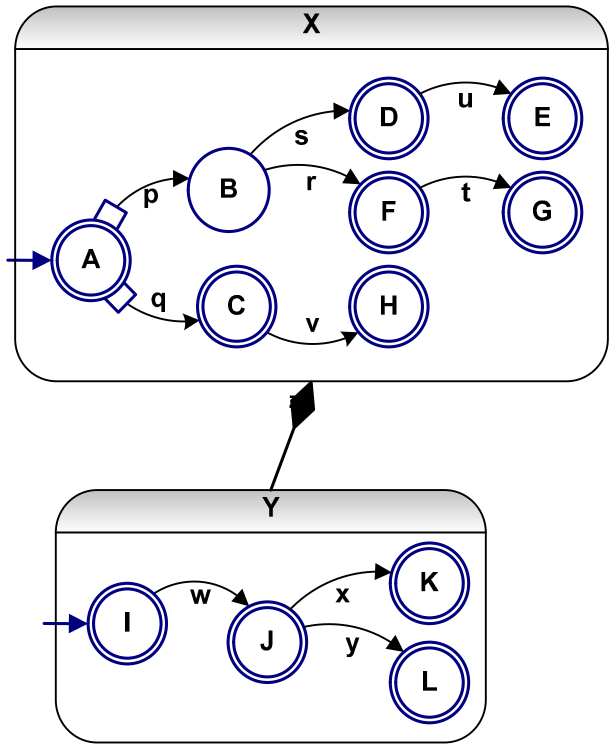


Figure 7.2 Defunct Object Model

combined process of *M* and *N* required both events, this object model can not be used to achieve the process. This is referred to as a “defunct” object model.

7.2 Process Determinism

Another concept of importance related to that of achievability is that of determinism. A process is said to be deterministic if for every conceivable event sequence that results in object model workflow completion, the process is guaranteed to complete. Recall that achievability says that a process “can” complete given a specific object model. Determinism means that a process “will” complete. The proof that a process is deterministic is two fold. First, prove that the process is “completely” achievable. Second, prove that for every control flow path in the object model the sequence of events is in keeping with that in the process model.

7.2.1 Determining Complete Achievability

A process *P* is said to have “complete achievability” with respect to an object model *O* if for every path to completion in *O*, process *P* is achievable.

A slight modification of the object model in Figure 7.1 is shown in Figure 7.3. It is not completely obvious that with this object model, the process of *M* and *N* of Figure 7.1 can be completed regardless of the path. Classification of a process as wholly achievable is done by performing the achievability algorithm as described in 7.1 with two modifications: 1) Record all choices that lead to an achievable result and place in a set θ, and 2) do not discontinue the algorithm when achievability is proven, but instead continue until all stacks are empty.

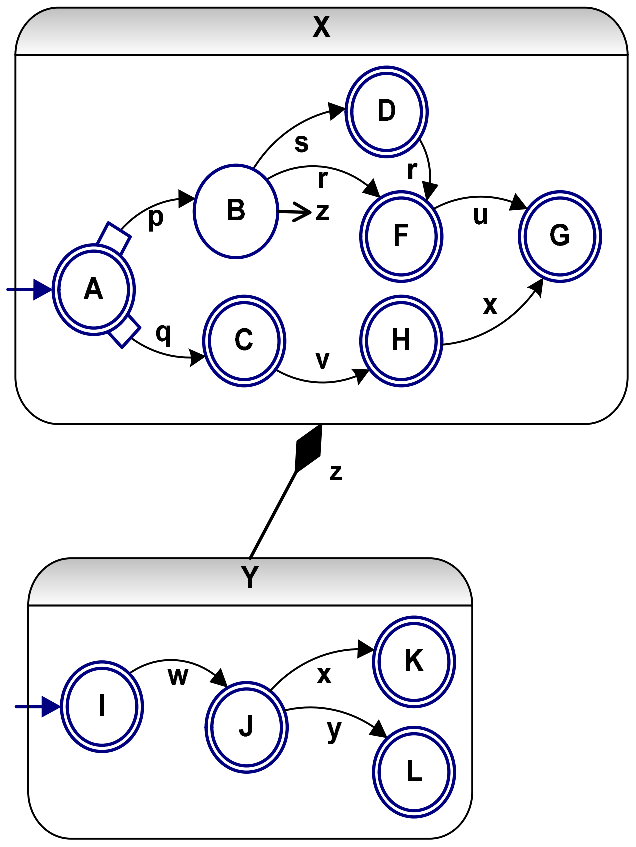


Figure 7.3 Completely Achievable Process

Given an object model *O* and a process model *P*, let *S* be the set of all paths through the object system *O* and let *θ* be the set of all eligible paths through the object model *O* that achieve the process *P*. Thus, process *P* is said to be achievable if:

$$\theta \subseteq S \text{ and } \theta \neq \{\}$$

Process *P* is said to be completely achievable if:

$$\theta \cap S = S \text{ and } S \neq \{\}$$

Thus, after full completion of the achievability algorithm, if the total set of eligible choices that will result in completion of the process is the same as the set of all choices to complete the object model, then the process is completely achievable. The logic is simple: if all choices "can" complete the process, then there are no choices that "can not" complete the process. Thus, the process can always be completed.

7.2.2 Determining Process Determinism

This still does not prove that a process model is deterministic. In order to complete the proof, it must be shown that the process is completely achievable and that for every eligible sequence of events the process will be completed. While the model in Figure 7.3 is completely achievable with respect to the process model of Figure 7.1, it is not deterministic. While every path can result in completing the process of Figure 7.1, note that the process model requires that event *w* be received prior to event *v*. The object model does not enforce this constraint. Thus, if an event *v* were received before event *w*, the object model would continue reductions normally but the process would no longer be valid.

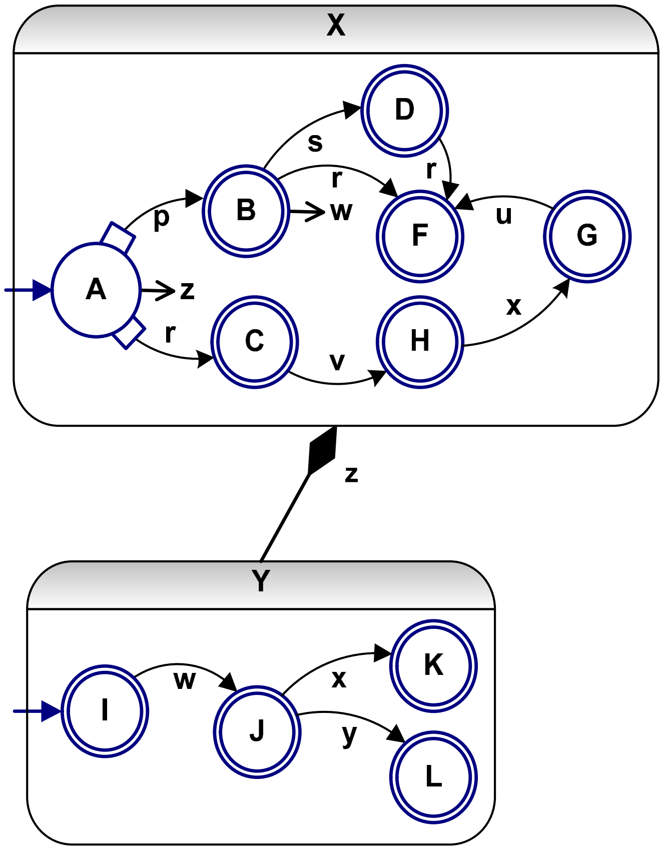


Figure 7.4 Process Determinism

Figure 7.4 shows a further modification of the object model to ensure determinism. Performing the achievability algorithm would indicate that this object model is completely achievable. The second step in proving that this process is deterministic resides in the fact that during the achievability algorithm backtracking in stack *V* only occurs as a result of completing the process.

Recall that during the achievability algorithm, if a process could not be completed given a certain choice, this processing was abandoned and the next choice in stack *V* was tried. In the process of determining "complete" achievability, stack *V* is popped when either a path is abandoned or a process is completed. If stack *V* is exhausted only because all paths resulted in completion of the process, then the process is guaranteed to complete regardless of the path chosen. Thus, the process is deterministic.

$$\begin{aligned} X &\overset{CAF}{\equiv} A.\overline{z}.z.p.B.\overline{w}.r.F \mid A.\overline{z}.z.r.C.v.H.x.G.u.F \mid A.\overline{z}.z.Y + \\ &\quad A.\overline{z}.z.p.B.\overline{w}.s.r.F \mid A.\overline{z}.z.r.C.v.H.x.G.u.F \mid A.\overline{z}.z.Y \\ Y &\overset{CAF}{\equiv} I.w.J.x.K + I.w.J.y.L \end{aligned}$$

$$\begin{aligned} M &\overset{CAF}{\equiv} O.r.P.v.N_1 \mid O.w.Q.v.N_1 \\ M &\xrightarrow{start} r.P.v.N_1 \mid w.Q.v.N_1 \end{aligned}$$

$$\begin{aligned} &U \triangleright \{w, r\} \\ &V \triangleright \{A.\overline{z}.z.p.B.\overline{w}.r.F \mid A.\overline{z}.z.r.C.v.H.x.G.u.F \mid A.\overline{z}.z.Y : 7, \\ &\qquad A.\overline{z}.z.p.B.\overline{w}.s.r.F \mid A.\overline{z}.z.r.C.v.H.x.G.u.F \mid A.\overline{z}.z.Y : 8\} \\ &\textit{pop V to}\, \theta \\ &\theta \triangleright \{A.\overline{z}.z.p.B.\overline{w}.r.F \mid A.\overline{z}.z.r.C.v.H.x.G.u.F \mid A.\overline{z}.z.Y\} \end{aligned}$$

$$\begin{aligned} X &\xrightarrow{create} p.B.\overline{w}.r.F \mid r.C.v.H.x.G.u.F \mid w.J.x.K + \\ &\quad p.B.\overline{w}.r.F \mid r.C.v.H.x.G.u.F \mid w.J.y.L + \\ &\quad p.B.\overline{w}.s.r.F \mid r.C.v.H.x.G.u.F \mid w.J.x.K + \\ &\quad p.B.\overline{w}.s.r.F \mid r.C.v.H.x.G.u.F \mid w.J.y.L \\ X &\xrightarrow{p,r} v.H.x.G.u.F \mid x.K + v.H.x.G.u.F \mid y.L \\ M &\xrightarrow{p,r} v.N_1 \mid v.N_1 \end{aligned}$$

$$U \triangleright \{v\}$$

$$\begin{aligned} &X \xrightarrow{v} x.G.u.F \mid x.K + x.G.u.F \mid y.L \\ &M \xrightarrow{v} x.R.u.S \\ &\textit{and so on until completion} \end{aligned}$$

$$\begin{aligned}
&pop\ V\ to\ \theta\\
&\theta \triangleright \{A.\overline{z}.z.p.B.\overline{w}.r.F \mid A.\overline{z}.z.r.C.v.H.x.G.u.F \mid A.\overline{z}.z.Y,\\
&\qquad A.\overline{z}.z.p.B.\overline{w}.s.r.F \mid A.\overline{z}.z.r.C.v.H.x.G.u.F \mid A.\overline{z}.z.Y\}
\end{aligned}$$

$$\begin{aligned}
&X \xrightarrow{p,s,r} v.H.x.G.u.F \mid x.K + v.H.x.G.u.F \mid y.L\\
&M \xrightarrow{p,s,r} v.N_1 \mid v.N_1\\
&continue\ as\ before
\end{aligned}$$

This example demonstrates that all terms in the object model can be followed to complete the process. Since the set of all choices *S* is equal to the final set of all achievable choices *θ*, the process model is completely achievable using this object model. In addition, during the achievability algorithm, every choice placed in stack *V* was achievable and thus this process is also deterministic.

7.3 Process Enforcement

The previous example of process determinism shows that creating a object model that guarantees the completion of a given process is possible and can be verified; however, it can be quite difficult with large models to create such models. An alternative to this approach is to use a process model as a constraint on an object model. This provides a simpler mechanism of guaranteeing completion of a process. The method for constraining the object model is by ensuring that any event received during execution of the object model that also exists in the process model must be eligible in both models.

In order for this to function properly, a process must be completely achievable on a given object model. The reason for this is quite simple. If there are paths which may be followed in the workflow of an object model that do not result in process completion it is likely because these paths do not contain events that exist in the process model. Figure 7.1 shows a process model that is achievable with respect to the object model. If the process model is used as a constraint against the object model, it does not guaranty completion.

$$\begin{aligned}
&X \xrightarrow{p} s.D.t.E \mid q.C.v.H + r.F.u.G \mid q.C.v.H + z.s.D.t.E \mid z.B_i \mid B.z.Y_i \mid q.C.v.H + \\
&\qquad z.r.F.u.G \mid B.z.B_i \mid B.z.Y \mid q.C.v.H \\
&M \xrightarrow{p} r.P.v.N_1 \mid w.Q.v.N_1 \quad p \text{ does not exist in process} \\
&X \xrightarrow{s} t.E \mid q.C.v.H \\
&M \xrightarrow{s} r.P.v.N_1 \mid w.Q.v.N_1 \quad s \text{ does not exist in process} \\
&X \xrightarrow{t,q,v} 0 \qquad\qquad v \text{ exists in } M \text{ but is not eligible in } M
\end{aligned}$$

Thus, two things can occur when attempting to constrain an object model with a incompletely achievable process: deadlock or object completion with no corresponding process completion. In this example, a deadlock resulted as event *v* is constrained by the process but enabling events *r* and *w* no longer exist in the object model. It is a trivial exercise to create a model where a path in the object model contains no constraining events in the process model. Thus, the object workflow would complete without the process even starting.

Using a completely achievable process model does not have this problem, as all paths can result in completion of the process. Recall that the only thing preventing a completely achievable process from being completed is the correct events occurring in the wrong order. However, if the process model is used to constrain the order of events, then the process model is guaranteed to complete.

As shown earlier, the process model of Figure 7.1 is completely achievable with respect to the object model in Figure 7.3 but is not deterministic. The main problem reason this model is not deterministic is that the process must receive event *v* before event *x* but this is required in the corresponding object model. The following demonstrates how the process model is used to ensure proper sequence of received events.

$$\begin{aligned}
X &\xrightarrow{p} r.F.u.G \mid w.J.x.K \mid q.C.v.H.x.G + r.F.u.G \mid w.J.y.L \mid q.C.v.H.x.G + \\
&\quad s.D.r.F.u.G \mid w.J.x.K \mid q.C.v.H.x.G + s.D.r.F.u.G \mid w.J.y.L \mid q.C.v.H.x.G \\
M &\xrightarrow{p} r.P.v.N.x.R.u.S \mid w.Q.v.N.x.R.u.S \qquad p\ does\ not\ exist\ in\ process \\
X &\xrightarrow{q} r.F.u.G \mid I.w.J.x.K \mid v.H.x.G + r.F.u.G \mid I.w.J.y.L \mid v.H.x.G + \\
&\quad s.D.r.F.u.G \mid I.w.J.x.K \mid v.H.x.G + s.D.r.F.u.G \mid I.w.J.y.L \mid v.H.x.G \\
M &\xrightarrow{q} r.P.v.N.x.R.u.S \mid w.Q.v.N.x.R.u.S \qquad q\ does\ not\ exist\ in\ process
\end{aligned}$$

At this point in execution, it is clear that event *v* is eligible in the object model but it is not in the process model. Thus, a receipt of an event *v* will be rejected since the process modeling is enforcing sequence in the object model. The only other eligible events in the object model are events *r*, *s*, and *w*. These are completely eligible since events *r* and *w* are eligible in the process model and event *s* is not constrained by it. Continuing with execution:

$$\begin{aligned}
X &\xrightarrow{r} u.G \mid w.J.x.K \mid v.H.x.G + u.G \mid w.J.y.L \mid v.H.x.G \\
M &\xrightarrow{r} v.N.x.R.u.S \mid w.Q.v.N.x.R.u.S
\end{aligned}$$

Now in addition to event *v*, the *u* event is eligible in the object model but is not in the process model. Thus, events *u* and *v* are ineligible.

$$\begin{aligned}
X &\xrightarrow{w} u.G \mid x.K \mid v.H.x.G + u.G \mid y.L \mid v.H.x.G \\
M &\xrightarrow{r} v.N.x.R.u.S \mid v.N.x.R.u.S
\end{aligned}$$

Now event *v* and *y* are the only eligible events.

$$\begin{aligned}
X &\xrightarrow{y} u.G \mid v.H.x.G \\
M &\xrightarrow{y} v.N.x.R.u.S \mid v.N.x.R.u.S \\
X &\xrightarrow{v} u.G \mid x.G \\
M &\xrightarrow{v} x.R.u.S \mid x.R.u.S \\
X &\xrightarrow{x} u.G \\
M &\xrightarrow{x} u.S \mid u.S \\
X &\xrightarrow{u} 0 \\
M &\xrightarrow{u} 0
\end{aligned}$$

This demonstrates the ability for a process to be used as a constraint on an object model. This is obviously unnecessary if the process is already deterministic, but it offers another alternative to creating and refining a deterministic model that, in practice, can be quite difficult.

7.4 Document Management Example

The Formal Object Interaction Language (FOIL) object diagrams can be used to model complex systems. In FOIL, the process and object models use the same notational elements and algebraic constructs. By the algorithm described in this chapter, the object model can be simulated to determine if it can perform the work of the process model (achievability). More importantly, if the object model is built correctly, the process model can be used as a constraint on execution during run-time.

7.4.1 Object Model

Consider a document management system in which there are multiple documents and multiple logons. A *System User* initiates a *Session* with the application and authenticates. There are two types of logons: a *User* logon and an *Editor* logon. There only difference between these two types of users is that the *Editor* can edit a *Document* while the user can merely open and close a *Document*. Multiple *Users* can open a *Document* at the same time, but no *User* may open a *Document* that is being edited. An *Editor* may not edit a *Document* that is open but must wait until all *Users* have closed the *Document*. To avoid resource starvation, if an *Editor* requests to edit a *Document* that is currently open then no other *User* may open that *Document* until the *Editor* completes the changes. Figure 7.5 shows the FOIL object model for such a system.

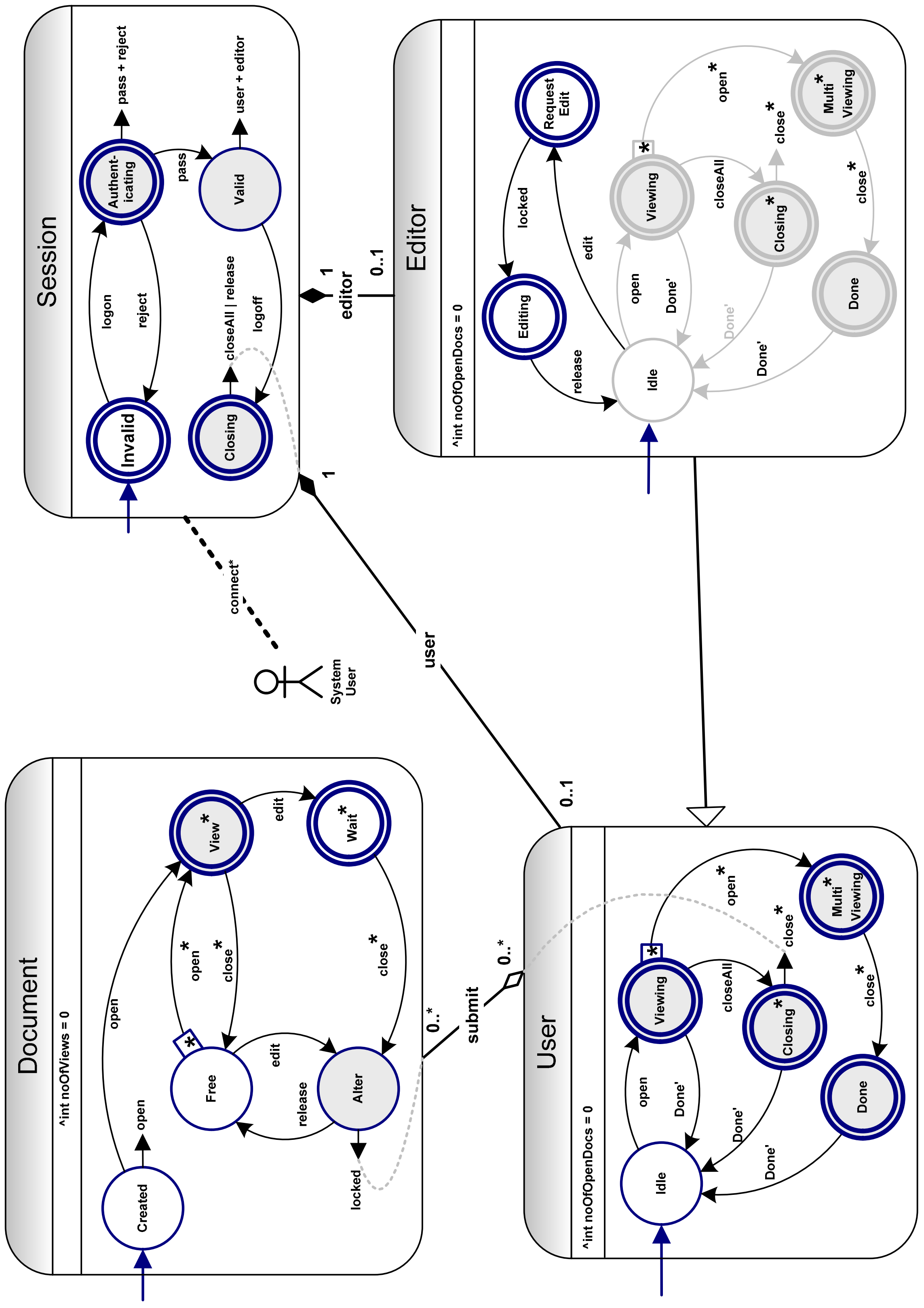


Figure 7.5 FOIL Document Management Object Model

This model is very concise, containing all of the necessary components to ensure all of the constraints listed previously. For instance, the locking property of an *Editor* request as discussed is an important aspect of this system. This behavior is completely specified in the *Document* object.

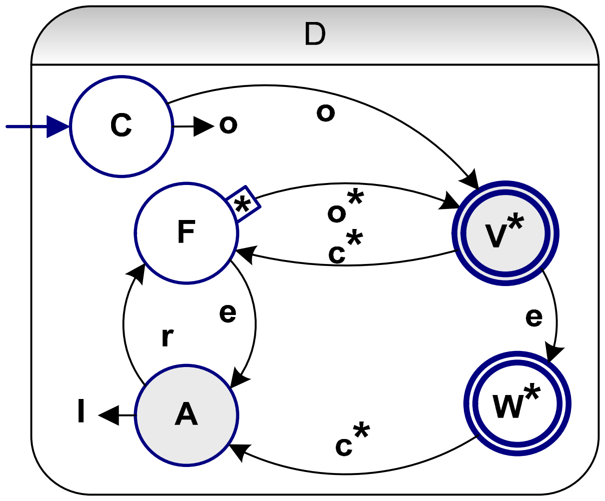


Figure 7.6 FOIL Document Object

Figure 7.6 shows the *Document* object in this example. When an *s* event is received the document is created and is immediately opened by the submitting *User*. After the submitting *User* closes the document then other *Users* may open it.

Each state and event is replaced with a letter in order to demonstrate the locking behavior with FOIL algebra. The following shows the construction of the *Document* object expression:

$$C \equiv \bar{o}.o.V_x \qquad F \equiv o_x.(V_x \mid F) + e.A \quad V_x \equiv c_x.F + e.W_x \qquad W_x \equiv c_x.A \qquad A \equiv \bar{a}.a.\bar{l}.r.F$$

$$\xrightarrow[C]{} \bar{o}.o.V_1$$

$$\xrightarrow[V]{} \bar{o}.o.c_1.F + \bar{o}.o.e.W_1$$

$$\xrightarrow[F]{} \bar{o}.o.c_1.o_2.V_1 \mid \bar{o}.o.c_1.o_2.F + \bar{o}.o.c_1.e.A + \bar{o}.o.e.W$$

$$\xrightarrow[W]{} \bar{o}.o.c_1.o_2.V_1 \mid \bar{o}.o.c_1.o_2.F + \bar{o}.o.c_1.e.A + \bar{o}.o.e.c_2.A$$

$$\xrightarrow[A]{} \bar{o}.o.c_1.o_2.V_1 \mid \bar{o}.o.c_1.o_2.F + \bar{o}.o.c_1.e.\bar{a}.a.\bar{l}.r.F + \bar{o}.o.e.c_2.\bar{a}.a.\bar{l}.r.F$$

In order to keep this expression simple for demonstration purposes, there is no specific object instance qualifier and state markers are not used. In addition, the behavior of active states is abbreviated for state *A* and is not shown in state *V*. These simplifications can be made because they do not impact the result in this case. The following demonstrates the algebra for two *Users* opening a *Document* and then an *Editor* requesting to edit the document.

1. $\xrightarrow{s} \bar{o}.o.c_1.o_2.V_1 \mid \bar{o}.o.c_1.o_2.F + \bar{o}.o.c_1.e.\bar{a}.a.\bar{l}.r.F + \bar{o}.o.e.c_2.\bar{a}.a.\bar{l}.r.F$
2. $\xrightarrow{o} c_1.o_2.V_1 \mid c_1.o_2.F + c_1.e.\bar{a}.a.\bar{l}.r.F + e.c_2.\bar{a}.a.\bar{l}.r.F$
3. $\xrightarrow{c} o_2.V_1 \mid o_2.F + e.\bar{a}.a.\bar{l}.r.F$
4. $\xrightarrow{o} V_1 \mid F$
5. $\xrightarrow[FVWA]{} o_2.c_2.F \mid o_2.F \mid c_1.F + o_2.e.c_2.\bar{a}.a.\bar{l}.r.F \mid o_2.F \mid c_1.F +$
$e.\bar{a}.a.\bar{l}.r.F \mid c_1.F + o_2.c_2.F \mid o_2.F \mid e.c_1.\bar{a}.a.\bar{l}.r.F +$
$o_2.e.c_2.\bar{a}.a.\bar{l}.r.F \mid o_2.F \mid e.c_1.\bar{a}.a.\bar{l}.r.F + e.\bar{a}.a.\bar{l}.r.F \mid e.c_1.\bar{a}.a.\bar{l}.r.F$
6. $\xrightarrow{o} c_2.F \mid o_2.F \mid c_1.F + e.c_2.\bar{a}.a.\bar{l}.r.F \mid F \mid c_1.F + c_2.F \mid F \mid e.c_1.\bar{a}.a.\bar{l}.r.F +$
$e.c_2.\bar{a}.a.\bar{l}.r.F \mid F \mid e.c_1.\bar{a}.a.\bar{l}.r.F$

$\xrightarrow[FVWA]{} c_2.F \mid o_3.c_3.F \mid o_3.F \mid c_1.F + c_2.F \mid o_3.e.c_x.\bar{a}.a.\bar{l}.r.F \mid o_3.F \mid c_1.F + c_2.F \mid e.\bar{a}.a.\bar{l}.r.F \mid c_1.F +$
$e.c_2.\bar{a}.a.\bar{l}.r.F \mid o_3.c_3.F \mid o_3.F \mid c_1.F + e.c_2.\bar{a}.a.\bar{l}.r.F \mid o_3.e.c_3.\bar{a}.a.\bar{l}.r.F \mid o_3.F \mid c_1.F +$
$e.c_2.\bar{a}.a.\bar{l}.r.F \mid e.\bar{a}.a.\bar{l}.r.F \mid c_1.F + c_2.F \mid o_3.c_3.F \mid o_3.F \mid e.c_1.\bar{a}.a.\bar{l}.r.F +$
$c_2.F \mid o_3.e.c_3.\bar{a}.a.\bar{l}.r.F \mid o_3.F \mid e.c_1.\bar{a}.a.\bar{l}.r.F + c_2.F \mid e.\bar{a}.a.\bar{l}.r.F \mid e.c_1.\bar{a}.a.\bar{l}.r.F +$
$e.c_2.\bar{a}.a.\bar{l}.r.F \mid o_3.c_3.F \mid o_3.F \mid e.c_1.\bar{a}.a.\bar{l}.r.F +$
$e.c_2.\bar{a}.a.\bar{l}.r.F \mid o_3.e.c_3.\bar{a}.a.\bar{l}.r.F \mid o_3.F \mid e.c_1.\bar{a}.a.\bar{l}.r.F +$
$e.c_2.\bar{a}.a.\bar{l}.r.F \mid e.\bar{a}.a.\bar{l}.r.F \mid e.c_1.\bar{a}.a.\bar{l}.r.F$

8. $\xrightarrow{e} c_2.F \mid \bar{a}.a.\bar{l}.r.F \mid c_1.F + c_2.\bar{a}.a.\bar{l}.r.F \mid o_3.c_3.F \mid o_3.F \mid c_1.F + c_2.\bar{a}.a.\bar{l}.r.F \mid \bar{a}.a.\bar{l}.r.F \mid c_1.F +$
$c_2.F \mid o_3.c_3.F \mid o_3.F \mid c_1.\bar{a}.a.\bar{l}.r.F + c_2.F \mid \bar{a}.a.\bar{l}.r.F \mid c_1.\bar{a}.a.\bar{l}.r.F +$
$c_2.\bar{a}.a.\bar{l}.r.F \mid o_3.c_3.F \mid o_3.F \mid c_1.\bar{a}.a.\bar{l}.r.F + c_2.\bar{a}.a.\bar{l}.r.F \mid \bar{a}.a.\bar{l}.r.F \mid c_1.\bar{a}.a.\bar{l}.r.F$
9. $\xrightarrow{a} c_2.F \mid r.F \mid c_1.F$
10. $\xrightarrow{c} c_2.F \mid r.F \mid F \equiv c_2.F \mid r.F$

Lines 1 and 2 show the initial submission of the document. The initial creation reduces the expression such that an *o* event is fired. Thus, the document is open immediately upon submission by a user. Line 3 shows that the document was closed by the submitter. Lines 4 and 5 show that a user opened the document. This reduces the expression and substitution is performed. Lines 6 and 7 show that an additional user has opened the document and that substitution of terms has again been performed. Line 8 shows the reduction that occurs when an editor requests to edit the document. This results in an immediate transmission of event *a*, to indicate the starting of the active state code, but only one term is in a state to receive this event (line 9). Thus, in line 10, it can be clearly seen that no one may open a document (event *o*) until the users close the document (events c_1 and c_2) and the editor releases the document (event *r*).

7.4.2 Process Model

Thus, the algebra constrains the system to prevent problems. In the example of Figure 7.5, users must be authenticated in order to perform any task. Behavioral inheritance is demonstrated as an editor is a type of user but can also edit document. This completely conforms to the concept of inheritance as discussed in section 5.2.2. Despite this concise object model complete with inheritance, concurrency and resource management, it does not guaranty that it will perform its desired function.

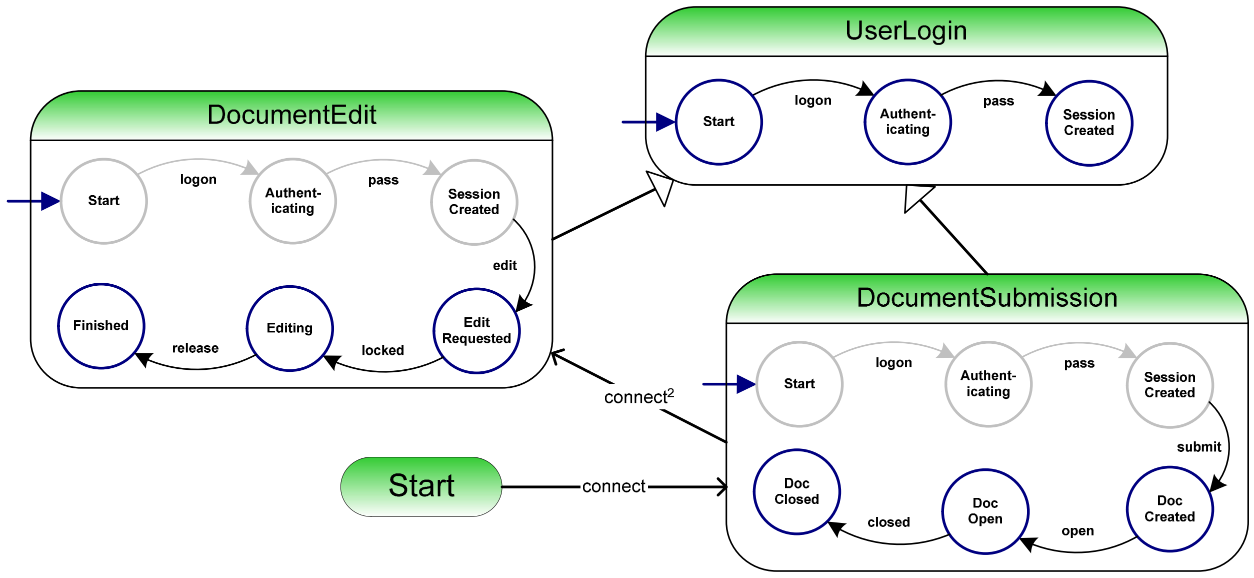


Figure 7.7 FOIL Document Management Process Model

Figure 7.7 shows a FOIL model for a process that is desired to be performed using the FOIL object model in Figure 7.5. This process is composed of two activities. First a document is submitted by a user; then it is desired that two editors make changes to the document. The editing steps to this process can be performed concurrently. Note the use of behavioral inheritance with respect to these to activities. This could be referred to as process inheritance but it does not significantly differ from that discussed for objects in section 5.2.2.

7.4.3 Achievability

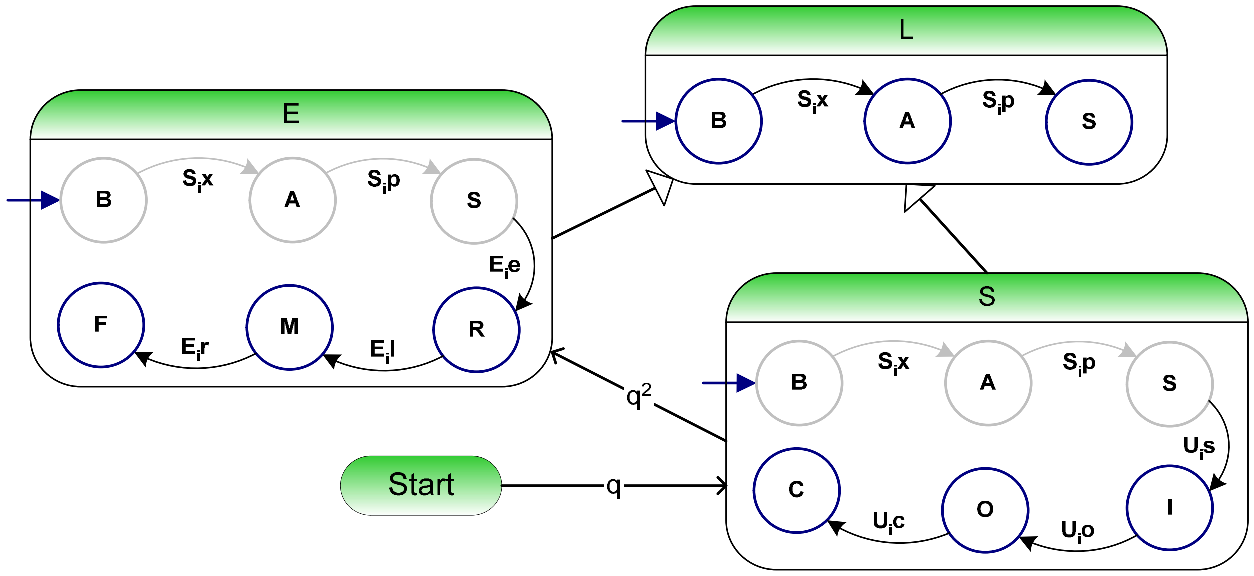


Figure 7.8 FOIL Document Management Process Algebra

The remainder of this section is devoted to demonstrating the concepts of this chapter as they apply to determining whether the process in Figure 7.7 is achievable with the object model of Figure 7.5. While this section contains a large amount of algebra, it is necessary to demonstrate the usefulness of the process validation feature of FOIL. For simplicity, state markers are shown in the initial construction but are removed during the validation process. Since this example involves multiple classes and multiple instances of the same class, object qualifiers are required. These are not shown during initial class construction but are added during object instantiation. Stack *V* during the achievability algorithm is not shown as its function in this example is trivial. The following two diagrams are identical to the previous models but have had their events and states substituted with letters for algebraic representation.

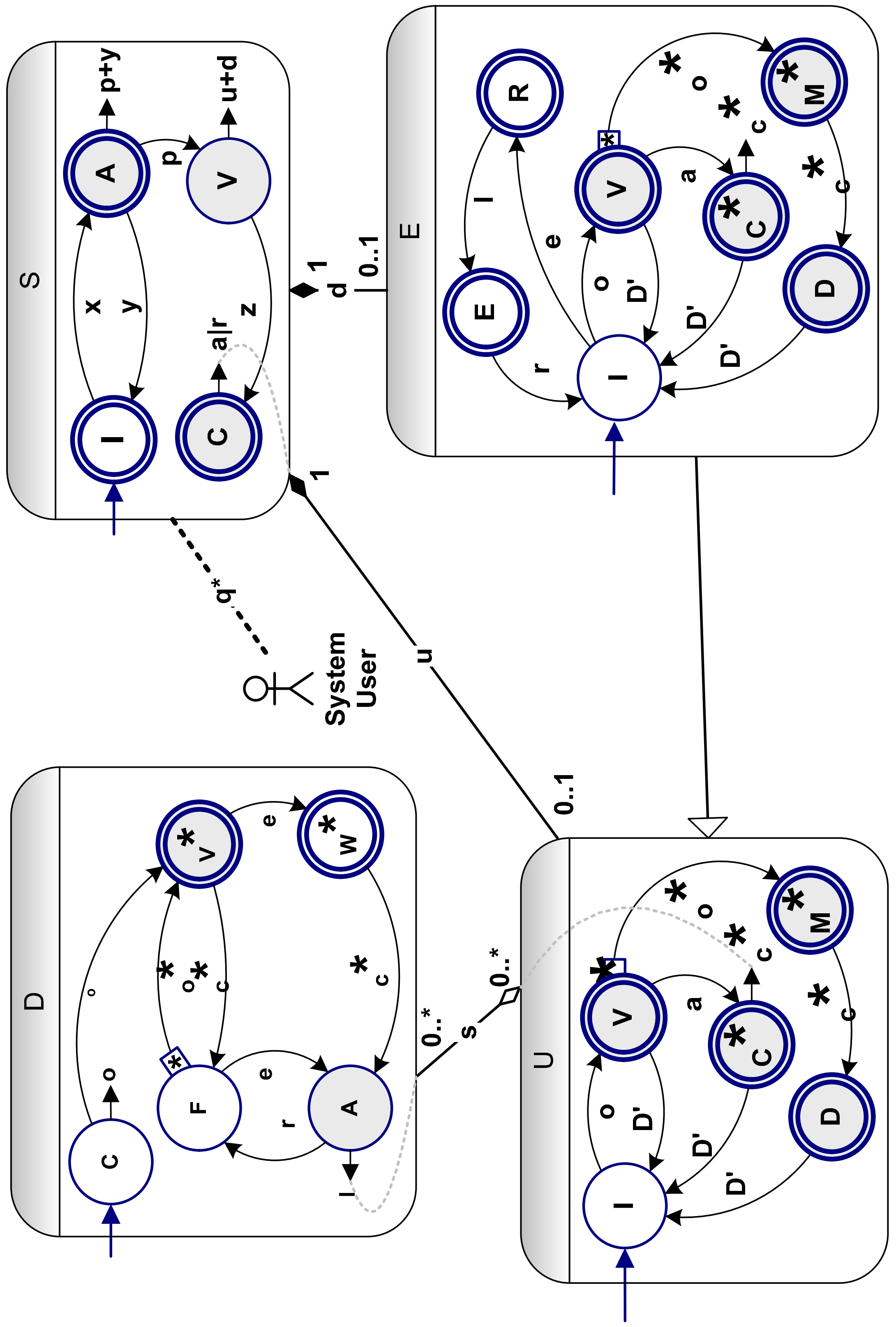


Figure 7.9 FOIL Document Management Object Model

7.4.3.1 Object Model Construction

The following show the construction of each object in the object model:

Document Object (D)

$$C \equiv \overline{o}.o.V_i \qquad V \equiv c_i.F + e.W_i \qquad W \equiv c_i.A$$
$$A \equiv \overline{l}.r.F \qquad F \equiv o_i.(V_i \mid F) + e.A \qquad D \equiv C.\overline{o}.o.V_i$$
$$D \overset{CAF}{\equiv} C.\overline{o}.o.V_1.c_1.F.o_1.V_2 \mid C.\overline{o}.o.V_1.c_1.F.o_1.F + C.\overline{o}.o.V_1.c_1.F.e.A + C.\overline{o}.o.V_1.e.W_1.c_1.A.\overline{l}.r.F.o_1.V_2 \mid C.\overline{o}.o.V_1.e.W_1.c_1.A.\overline{l}.r.F.o_1.F + C.\overline{o}.o.V_1.e.W_1.c_1.A.\overline{l}.r.F.e.A$$
$$D \overset{CAF}{\equiv} \overline{o}.o.c_1.o_1 \mid \overline{o}.o.c_1.o_1.F + \overline{o}.o.c_1.e.A + \overline{o}.o.e.c_1.\overline{l}.r.o_1.V_1 \mid \overline{o}.o.e.c_1.\overline{l}.r.o_1.F + \overline{o}.o.e.c_1.\overline{l}.r.e.A$$

User Object (U))

$$I \equiv o.V + s.(D \mid o.V) \qquad V \equiv o_i.(M_i \mid V) + a.C_i + D'.I$$
$$M \equiv c_i.D \qquad C \equiv \overline{c*}.D'.I \qquad D \equiv \overline{`D.}`D.D.D'.I$$
$$U \overset{CAF}{\equiv} o.V.o_i.M_1.c_1.\overline{`D.}`D.D.D'.I \mid o.V.o_1.V + o.V.a.C_i.\overline{c*}.D'.I + o.V.D'.I + s.D_i \mid s.o.V.o_1.M_1.c_1.\overline{`D.}`D.D.D'.I \mid s.o.V.o_1.V + s.o.V.a.C_1.\overline{c*}.D'.I + s.o.V.D'.I$$
$$U \overset{CAF}{\equiv} o.o_1.c_1.\overline{`D.}`D.D'.I \mid o.o_1.V + o.a.\overline{c*}.D'.I + o.D'.I + s.D_i \mid s.o.o_1.c_1.\overline{`D.}`D.D'.I \mid s.o.o_1.V + s.o.a.\overline{c*}.D'.I + s.o.D'.I$$

Editor Object (E)

$$I \equiv o.V + e.R + s.(D \mid (o.V + e.R)) \qquad R \equiv l.E \qquad E \equiv r.I$$
$$V \equiv o_i.(M_i \mid V) + a.C_i + D'.I \qquad M \equiv c_i.D \qquad C \equiv \overline{c*}.D'.I \qquad D \equiv \overline{`D.}`D.D.D'.I$$
$$E \overset{CAF}{\equiv} o.V.o_i.M_i.c_i.\overline{`D.}`D.D.D'.I \mid o.V.o_i.V + o.V.a.C_i.\overline{c*}.D'.I + o.V.D'.I + e.R.l.E.r.I + s.D \mid s.o.V.o_i.M_i.c_i.\overline{`D.}`D.D.D'.I \mid s.o.V.o_i.V + s.D \mid s.o.V.a.C_i.\overline{c*}.D'.I + s.D \mid s.o.V.D'.I + s.D \mid s.e.R.l.E.r.I$$
$$E \overset{CAF}{\equiv} o.o_i.c_i.\overline{`D.}`D.D'.I \mid o.o_i.V + o.a.\overline{c*}.D'.I + o.D'.I + e.l.r.I + s.D \mid s.o.o_i.c_i.\overline{`D.}`D.D'.I \mid s.o.o_i.V + s.D \mid s.o.a.\overline{c*}.D'.I + s.D \mid s.o.D'.I + s.D \mid s.e.l.r.I$$

Session Object (S)

$$I \equiv x.A \qquad A \equiv (\overline{p} + \overline{y}).(p.V + y.I)$$
$$V \equiv (\overline{u} + \overline{d}).(z.C + d.(E \mid z.C) + u.(U \mid z.C)) \qquad C \equiv (\overline{a} \mid \overline{r})$$
$$V \overset{CAF}{\equiv} z.C + \overline{d}.d.E \mid \overline{d}.d.z.C + \overline{u}.u.U \mid \overline{u}.u.z.C$$
$$\begin{aligned} S \overset{CAF}{\equiv}\ & x.A.\overline{p}.p.V.z.C.\overline{a} \mid x.A.\overline{p}.p.V.z.C.\overline{r} + \\ & x.A.\overline{p}.p.V.\overline{d}.d.E \mid x.A.\overline{p}.p.V.\overline{d}.d.z.C.\overline{a} \mid x.A.\overline{p}.p.V.\overline{d}.d.z.C.\overline{r} + \\ & x.A.\overline{p}.p.V.\overline{u}.u.U \mid x.A.\overline{p}.p.V.\overline{u}.u.z.C.\overline{a} \mid x.A.\overline{p}.p.V.\overline{u}.u.z.C.\overline{r} + \\ & x.A.\overline{y}.y.I \end{aligned}$$
$$\begin{aligned} S \overset{CAF}{\equiv}\ & x.\overline{p}.p.z.\overline{a} \mid x.\overline{p}.p.z.\overline{r} + x.\overline{p}.p.\overline{d}.d.E \mid x.\overline{p}.p.\overline{d}.d.z.\overline{a} \mid x.\overline{p}.p.\overline{d}.d.z.\overline{r} + \\ & x.\overline{p}.p.\overline{u}.u.U \mid x.\overline{p}.p.\overline{u}.u.z.\overline{a} \mid x.\overline{p}.p.\overline{u}.u.z.\overline{r} + x.\overline{y}.y.I \end{aligned}$$

System Expression (α)

$$\alpha \equiv q.(S_i \mid \alpha) \overset{CAF}{\equiv} q.S_i \mid q.\alpha$$

7.4.3.2 Process Model Construction

Document Submission (P_S)

$$B \equiv S_i x.A \qquad A \equiv S_i p.S.U_i s.I.U_i o.O.U_i c.C.(q_1.E_1 \mid q_2.E_2)$$
$$P_s \equiv S_i x.A.S_i p.S.U_i s.I.U_i o.O.U_i c.C.(q_1.E_1 \mid q_2.E_2)$$
$$P_s \overset{CAF}{\equiv} S_i x.A.S_i p.S.U_i s.I.U_i o.O.U_i c.C.q_1.E_1 \mid S_i x.A.S_i p.S.U_i s.I.U_i o.O.U_i c.C.q_2.E_2$$
$$P_s \overset{CAF}{\equiv} S_i x.S_i p.U_i s.U_i o.U_i c.q_1.E_1 \mid S_i x.S_i p.U_i s.U_i o.U_i c.q_2.E_2$$

Document Edit (P_E)

$$B \equiv S_i x.A \qquad A \equiv S_i p.S.E_i e.R.E_i l.M.E_i r.F$$
$$P_E \overset{CAF}{\equiv} S_i x.A.S_i p.S.E_i e.R.E_i l.M.E_i r.F$$
$$P_E \overset{CAF}{\equiv} S_i x.S_i p.E_i e.E_i l.E_i r$$

Document Process (P)

$$E_1 = L_2 \qquad E_2 = L_3$$
$$P \equiv q.P_s$$
$$P \overset{CAF}{\equiv} q.S_1 x.S_1 p.U_1 s.U_1 o.U_1 c.q_1.S_2 x.S_2 p.E_2 e.E_2 l.E_2 r \mid q.S_1 x.S_1 p.U_1 s.U_1 o.U_1 c.q_2.S_3 x.S_3 p.E_3 e.E_3 l.E_3 r$$

7.4.4 Achievability Algorithm

Iteration 1

$$U \triangleright \{q\}$$
$$\alpha \equiv q.S_i \mid q.\alpha$$
$$\alpha \xrightarrow{q} S_1 \mid q.S_i \mid q.\alpha$$
$$\equiv S_1x.\overline{S_1p}.S_1p.S_1z.\overline{S_1a} \mid S_1x.\overline{S_1p}.S_1p.S_1z.\overline{S_1r} \mid q.S_i \mid q.\alpha +$$
$$S_1x.\overline{S_1p}.S_1p.\overline{S_1d}.S_1d.E_i \mid S_1x.\overline{S_1p}.S_1p.\overline{S_1d}.S_1d.S_1z.\overline{S_1a} \mid S_1x.\overline{S_1p}.S_1p.\overline{S_1d}.S_1d.S_1z.\overline{S_1r} \mid q.S_i \mid q.\alpha +$$
$$S_1x.\overline{S_1p}.S_1p.\overline{S_1u}.S_1u.U_i \mid S_1x.\overline{S_1p}.S_1p.\overline{S_1u}.S_1u.S_1z.\overline{S_1a} \mid S_1x.\overline{S_1p}.S_1p.\overline{S_1u}.S_1u.S_1z.\overline{S_1r} \mid q.S_i \mid q.\alpha +$$
$$S_1x.\overline{S_1y}.S_1y.S_1I \mid q.S_i \mid q.\alpha$$
$$P \xrightarrow{q} S_1x.S_1p.U_1s.U_1o.U_1c.q_1.S_2x.S_2p.E_2e.E_2l.E_2r \mid S_1x.S_1p.U_1s.U_1o.U_1c.q_2.S_3x.S_3p.E_3e.E_3l.E$$

Iteration 2

$$U \triangleright \{S_1x\}$$
$$\alpha \xrightarrow{S_1x} \overline{S_1p}.S_1p.S_1z.\overline{S_1a} \mid \overline{S_1p}.S_1p.S_1z.\overline{S_1r} \mid q.S_i \mid q.\alpha +$$
$$\overline{S_1p}.S_1p.\overline{S_1d}.S_1d.E_i \mid \overline{S_1p}.S_1p.\overline{S_1d}.S_1d.S_1z.\overline{S_1a} \mid \overline{S_1p}.S_1p.\overline{S_1d}.S_1d.S_1z.\overline{S_1r} \mid q.S_i \mid q.\alpha +$$
$$\overline{S_1p}.S_1p.\overline{S_1u}.S_1u.U_i \mid \overline{S_1p}.S_1p.\overline{S_1u}.S_1u.S_1z.\overline{S_1a} \mid \overline{S_1p}.S_1p.\overline{S_1u}.S_1u.S_1z.\overline{S_1r} \mid q.S_i \mid q.\alpha +$$
$$\overline{S_1y}.S_1y.S_1I \mid q.S_i \mid q.\alpha$$
$$P \xrightarrow{S_1x} S_1p.U_1s.U_1o.U_1c.q_1.S_2x.S_2p.E_2e.E_2l.E_2r \mid S_1p.U_1s.U_1o.U_1c.q_2.S_3x.S_3p.E_3e.E_3l.E$$
$$\alpha \xrightarrow{S_1p} S_1z.\overline{S_1a} \mid S_1z.\overline{S_1r} \mid q.S_i \mid q.\alpha + \overline{S_1d}.S_1d.E_i \mid \overline{S_1d}.S_1d.S_1z.\overline{S_1a} \mid \overline{S_1d}.S_1d.S_1z.\overline{S_1r} \mid q.S_i \mid q.\alpha +$$
$$\overline{S_1u}.S_1u.U_i \mid \overline{S_1u}.S_1u.S_1z.\overline{S_1a} \mid \overline{S_1u}.S_1u.S_1z.\overline{S_1r} \mid q.S_i \mid q.\alpha$$

$$P \xrightarrow{S_1p} U_1s.U_1o.U_1c.q_1.S_2x.S_2p.E_2e.E_2l.E_2r \mid U_1s.U_1o.U_1c.q_2.S_3x.S_3p.E_3e.E_3l.E$$
$$\alpha \xrightarrow{S_1u} U_1 \mid S_1z.\overline{S_1a} \mid S_1z.\overline{S_1r} \mid q.S_i \mid q.\alpha$$
$$\equiv U_1o.U_1o_1.U_1c_1.\overline{U_1\text{`}D}.U_1\text{`}D.U_1D'.U_1I \mid U_1o.U_1o_1.U_1V \mid S_1z.\overline{S_1a} \mid S_1z.\overline{S_1r} \mid q.S_i \mid q.\alpha +$$
$$U_1o.U_1a.\overline{U_1c}*.U_1D'.U_1I \mid S_1z.\overline{S_1a} \mid S_1z.\overline{S_1r} \mid q.S_i \mid q.\alpha +$$
$$U_1o.U_1D'.U_1I \mid S_1z.\overline{S_1a} \mid S_1z.\overline{S_1r} \mid q.S_i \mid q.\alpha +$$
$$U_1s.U_1D_i \mid U_1s.U_1o.U_1o_1.U_1c_1.\overline{U_1\text{`}D}.U_1\text{`}D.U_1D'.U_1I \mid U_1s.U_1o.U_1o_1.U_1V \mid S_1z.\overline{S_1a} \mid S_1z.\overline{S_1r} \mid q.S_i \mid q.\alpha +$$
$$U_1s.U_1o.U_1a.\overline{U_1c}*.U_1D'.U_1I \mid S_1z.\overline{S_1a} \mid S_1z.\overline{S_1r} \mid q.S_i \mid q.\alpha +$$
$$U_1s.U_1o.U_1D'.U_1I \mid S_1z.\overline{S_1a} \mid S_1z.\overline{S_1r} \mid q.S_i \mid q.\alpha$$
$$P \xrightarrow{S_1u} \textit{does not exist in process } P$$

Iteration 3

$U \rhd \{U_1s\}$

$\alpha \xrightarrow{U_1s} U_1D_1 \mid U_1o.U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid U_1o.U_1o_1.U_1V \mid S_1z.\overline{S_1a} \mid S_1z.\overline{S_1r} \mid q.S_i \mid q.\alpha +$

$U_1o.U_1a.\overline{U_1c*}.U_1D'.U_1I \mid S_1z.\overline{S_1a} \mid S_1z.\overline{S_1r} \mid q.S_i \mid q.\alpha +$

$U_1o.U_1D'.U_1I \mid S_1z.\overline{S_1a} \mid S_1z.\overline{S_1r} \mid q.S_i \mid q.\alpha$

$\equiv \overline{U_1D_1o}.U_1D_1o.U_1D_1c_1.U_1D_1o_1 \mid \overline{U_1D_1o}.U_1D_1o.U_1D_1c_1.U_1D_1o_1.U_1D_1F \mid$

$U_1o.U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid \quad U_1o.U_1o_1.U_1V \mid S_1z.\overline{S_1a} \mid S_1z.\overline{S_1r} \mid q.S_i \mid q.\alpha +$

$\overline{U_1D_1o}.U_1D_1o.U_1D_1c_1.U_1D_1e.U_1D_1A \mid U_1o.U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid$

$U_1o.U_1o_1.U_1V \mid S_1z.\overline{S_1a} \mid S_1z.\overline{S_1r} \mid q.S_i \mid q.\alpha +$

$\overline{U_1D_1o}.U_1D_1o.U_1D_1e.U_1D_1c_1.\overline{U_1D_1l}.U_1D_1r.U_1D_1o_1.U_1D_1V_1 \mid$

$\overline{U_1D_1o}.U_1D_1o.U_1D_1e.U_1D_1c_1.\overline{U_1D_1l}.U_1D_1r.U_1D_1o_1.U_1D_1F \mid$

$U_1o.U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid U_1o.U_1o_1.U_1V \mid S_1z.\overline{S_1a} \mid S_1z.\overline{S_1r} \mid q.S_i \mid q.\alpha +$

$\overline{U_1D_1o}.U_1D_1o.U_1D_1e.U_1D_1c_1.\overline{U_1D_1l}.U_1D_1r.U_1D_1e.U_1D_1A \mid U_1o.U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid$

$U_1o.U_1o_1.U_1V \mid S_1z.\overline{S_1a} \mid S_1z.\overline{S_1r} \mid q.S_i \mid q.\alpha +$

$U_1o.U_1a.\overline{U_1c*}.U_1D'.U_1I \mid S_1z.\overline{S_1a} \mid S_1z.\overline{S_1r} \mid q.S_i \mid q.\alpha +$

$U_1o.U_1D'.U_1I \mid S_1z.\overline{S_1a} \mid S_1z.\overline{S_1r} \mid q.S_i \mid q.\alpha$

$P \xrightarrow{U_1s} U_1o.U_1c.q_1.S_2x.S_2p.E_2e.E_2l.E_2r \mid U_1o.U_1c.q_2.S_3x.S_3p.E_3e.E_3l.E$

Step 4

$\alpha \xrightarrow{U_1D_1o} U_1D_1c_1.U_1D_1o_1 \mid U_1D_1c_1.U_1D_1o_1.U_1D_1F \mid U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid$

$U_1o_1.U_1V \mid S_1z.\overline{S_1a} \mid S_1z.\overline{S_1r} \mid q.S_i \mid q.\alpha +$

$U_1D_1c_1.U_1D_1e.U_1D_1A \mid U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid U_1o_1.U_1V \mid S_1z.\overline{S_1a} \mid S_1z.\overline{S_1r} \mid q.S_i \mid q.\alpha +$

$U_1D_1o.U_1D_1e.U_1D_1c_1.\overline{U_1D_1l}.U_1D_1r.U_1D_1o_1.U_1D_1V_1 \mid U_1D_1e.U_1D_1c_1.\overline{U_1D_1l}.U_1D_1r.U_1D_1o_1.U_1D_1F \mid$

$U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid U_1o_1.U_1V \mid S_1z.\overline{S_1a} \mid S_1z.\overline{S_1r} \mid q.S_i \mid q.\alpha +$

$U_1D_1e.U_1D_1c_1.\overline{U_1D_1l}.U_1D_1r.U_1D_1e.U_1D_1A \mid U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid$

$U_1o_1.U_1V \mid S_1z.\overline{S_1a} \mid S_1z.\overline{S_1r} \mid q.S_i \mid q.\alpha +$

$U_1a.\overline{U_1c*}.U_1D'.U_1I \mid S_1z.\overline{S_1a} \mid S_1z.\overline{S_1r} \mid q.S_i \mid q.\alpha +$

$U_1D'.U_1I \mid S_1z.\overline{S_1a} \mid S_1z.\overline{S_1r} \mid q.S_i \mid q.\alpha$

$P \xrightarrow{U_1D_1o} U_1c.q_1.S_2x.S_2p.E_2e.E_2l.E_2r \mid U_1c.q_2.S_3x.S_3p.E_3e.E_3l.E$

Step 5

Note that the process model allows for an edit event, this is not a problem as the object model does not allow U_1D_1e.

$$
\begin{aligned}
&U \triangleright \{U_1c\}\\
&\alpha \xrightarrow{U_1c} U_1D_1o_1 \mid U_1D_1c_1.U_1D_1o_1.U_1D_1F \mid U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid\\
&\qquad U_1D_1e.U_1D_1A \mid U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid U_1o_1.U_1V \mid S_1z.\overline{S_1a} \mid S_1z.\overline{S_1r} \mid q.S_i \mid q.\alpha\\
&P \xrightarrow{U_1c} q_1.S_2x.S_2p.E_2e.E_2l.E_2r \mid q_2.S_3x.S_3p.E_3e.E_3l.E
\end{aligned}
$$

Iteration 6

$$
\begin{aligned}
&U \triangleright \{q, q\}\\
&\alpha \xrightarrow{q} U_1D_1o_1 \mid U_1D_1c_1.U_1D_1o_1.U_1D_1F \mid U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid\\
&\qquad U_1D_1e.U_1D_1A \mid U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid U_1o_1.U_1V \mid S_1z.\overline{S_1a} \mid S_1z.\overline{S_1r} \mid S_2 \mid \alpha\\
&\qquad \equiv U_1D_1o_1 \mid U_1D_1c_1.U_1D_1o_1.U_1D_1F \mid U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid U_1D_1e.U_1D_1A \mid\\
&\qquad\qquad U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid\\
&\qquad\qquad U_1o_1.U_1V \mid S_1z.\overline{S_1a} \mid S_1z.\overline{S_1r} \mid S_2x.\overline{S_2p}.S_2p.S_2z.\overline{S_2a} \mid S_2x.\overline{S_2p}.S_2p.S_2z.\overline{S_2r} \mid q.S_i \mid q.\alpha +\\
&\qquad U_1D_1o_1 \mid U_1D_1c_1.U_1D_1o_1.U_1D_1F \mid U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid U_1D_1e.U_1D_1A \mid\\
&\qquad\qquad U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid U_1o_1.U_1V \mid S_1z.\overline{S_1a} \mid\\
&\qquad\qquad S_1z.\overline{S_1r} \mid S_2x.\overline{S_2p}.S_2p.\overline{S_2d}.S_2d.E_2 \mid S_2x.\overline{S_2p}.S_2p.\overline{S_2d}.S_2d.S_2z.\overline{S_2a} \mid\\
&\qquad\qquad S_2x.\overline{S_2p}.S_2p.\overline{S_2d}.S_2d.S_2z.\overline{S_2r} \mid q.S_i \mid q.\alpha +\\
&\qquad U_1D_1o_1 \mid U_1D_1c_1.U_1D_1o_1.U_1D_1F \mid U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid U_1D_1e.U_1D_1A \mid\\
&\qquad\qquad U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid U_1o_1.U_1V \mid S_1z.\overline{S_1a} \mid\\
&\qquad\qquad S_1z.\overline{S_1r} \mid S_2x.\overline{S_2p}.S_2p.\overline{S_2u}.S_2u.U_2 \mid S_2x.\overline{S_2p}.S_2p.\overline{S_2u}.S_2u.S_2z.\overline{S_2a} \mid\\
&\qquad\qquad S_2x.\overline{S_2p}.S_2p.\overline{S_2u}.S_2u.S_2z.\overline{S_2r} \mid q.S_i \mid q.\alpha +\\
&\qquad U_1D_1o_1 \mid U_1D_1c_1.U_1D_1o_1.U_1D_1F \mid U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid U_1D_1e.U_1D_1A \mid\\
&\qquad\qquad U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid U_1o_1.U_1V \mid S_1z.\overline{S_1a} \mid\\
&\qquad\qquad S_1z.\overline{S_1r} \mid S_2x.\overline{S_2y}.S_2y.S_2I \mid q.S_i \mid q.\alpha\\
&P \xrightarrow{q} S_2x.S_2p.E_2e.E_2l.E_2r \mid q_2.S_3x.S_3p.E_3e.E_3l.E
\end{aligned}
$$

Iteration 7

$$U \triangleright \{S_2x, q, q\}$$

$$\alpha \xrightarrow{S_2x} U_1D_1o_1 \,|\, U_1D_1c_1.U_1D_1o_1.U_1D_1F \,|\, U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \,|\, U_1D_1e.U_1D_1A \,|$$
$$U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \,|$$
$$U_1o_1.U_1V \,|\, S_1z.\overline{S_1a} \,|\, S_1z.\overline{S_1r} \,|\, \overline{S_2p}.S_2p.S_2z.\overline{S_2a} \,|\, \overline{S_2p}.S_2p.S_2z.\overline{S_2r} \,|\, q.S_i \,|\, q.\alpha +$$
$$U_1D_1o_1 \,|\, U_1D_1c_1.U_1D_1o_1.U_1D_1F \,|\, U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \,|\, U_1D_1e.U_1D_1A \,|$$
$$U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \,|\, U_1o_1.U_1V \,|\, S_1z.\overline{S_1a} \,|$$
$$S_1z.\overline{S_1r} \,|\, \overline{S_2p}.S_2p.\overline{S_2d}.S_2d.E_2 \,|\, \overline{S_2p}.S_2p.\overline{S_2d}.S_2d.S_2z.\overline{S_2a} \,|$$
$$\overline{S_2p}.S_2p.\overline{S_2d}.S_2d.S_2z.\overline{S_2r} \,|\, q.S_i \,|\, q.\alpha +$$
$$U_1D_1o_1 \,|\, U_1D_1c_1.U_1D_1o_1.U_1D_1F \,|\, U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \,|\, U_1D_1e.U_1D_1A \,|$$
$$U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \,|\, U_1o_1.U_1V \,|\, S_1z.\overline{S_1a} \,|$$
$$S_1z.\overline{S_1r} \,|\, \overline{S_2p}.S_2p.\overline{S_2u}.S_2u.U_2 \,|\, \overline{S_2p}.S_2p.\overline{S_2u}.S_2u.S_2z.\overline{S_2a} \,|$$
$$\overline{S_2p}.S_2p.\overline{S_2u}.S_2u.S_2z.\overline{S_2r} \,|\, q.S_i \,|\, q.\alpha +$$
$$U_1D_1o_1 \,|\, U_1D_1c_1.U_1D_1o_1.U_1D_1F \,|\, U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \,|\, U_1D_1e.U_1D_1A \,|$$
$$U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \,|\, U_1o_1.U_1V \,|\, S_1z.\overline{S_1a} \,|$$
$$S_1z.\overline{S_1r} \,|\, \overline{S_2y}.S_2y.S_2I \,|\, q.S_i \,|\, q.\alpha$$

$$P \xrightarrow{q} S_2p.E_2e.E_2l.E_2r \,|\, q_2.S_3x.S_3p.E_3e.E_3l.E$$

$$\begin{aligned}
\alpha \xrightarrow{S_2p,S_2d} & U_1D_1o_1 \mid U_1D_1c_1.U_1D_1o_1.U_1D_1F \mid U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid \\
& U_1D_1e.U_1D_1A \mid U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid U_1o_1.U_1V \mid S_1z.\overline{S_1a} \mid \\
& S_1z.\overline{S_1r} \mid E_2 \mid S_2z.\overline{S_2a} \mid S_2z.\overline{S_2r} \mid q.S_i \mid q.\alpha \\
& \equiv U_1D_1o_1 \mid U_1D_1c_1.U_1D_1o_1.U_1D_1F \mid U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid U_1D_1e.U_1D_1A \mid \\
& U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid U_1o_1.U_1V \mid S_1z.\overline{S_1a} \mid \\
& S_1z.\overline{S_1r} \mid E_2o.E_2o_1.E_2c_1.\overline{E_2`D}.E_2`D.E_2D'.E_2I \mid E_2o.E_2o_1.E_2V \mid S_2z.\overline{S_2a} \mid S_2z.\overline{S_2r} \mid q.S_i \mid q.\alpha + \\
& U_1D_1o_1 \mid U_1D_1c_1.U_1D_1o_1.U_1D_1F \mid U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid U_1D_1e.U_1D_1A \mid \\
& U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid U_1o_1.U_1V \mid S_1z.\overline{S_1a} \mid \\
& S_1z.\overline{S_1r} \mid E_2o.E_2a.\overline{E_2c*}.E_2D'.E_2I \mid S_2z.\overline{S_2a} \mid S_2z.\overline{S_2r} \mid q.S_i \mid q.\alpha + \\
& U_1D_1o_1 \mid U_1D_1c_1.U_1D_1o_1.U_1D_1F \mid U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid U_1D_1e.U_1D_1A \mid \\
& U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid U_1o_1.U_1V \mid S_1z.\overline{S_1a} \mid \\
& S_1z.\overline{S_1r} \mid E_2o.E_2D'.E_2I \mid S_2z.\overline{S_2a} \mid S_2z.\overline{S_2r} \mid q.S_i \mid q.\alpha + \\
& U_1D_1o_1 \mid U_1D_1c_1.U_1D_1o_1.U_1D_1F \mid U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid U_1D_1e.U_1D_1A \mid \\
& U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid U_1o_1.U_1V \mid S_1z.\overline{S_1a} \mid \\
& S_1z.\overline{S_1r} \mid E_2e.E_2l.E_2r.E_2I \mid S_2z.\overline{S_2a} \mid S_2z.\overline{S_2r} \mid q.S_i \mid q.\alpha + \\
& U_1D_1o_1 \mid U_1D_1c_1.U_1D_1o_1.U_1D_1F \mid U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid U_1D_1e.U_1D_1A \mid \\
& U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid U_1o_1.U_1V \mid S_1z.\overline{S_1a} \mid \\
& S_1z.\overline{S_1r} \mid E_2s.E_2D_i \mid E_2s.E_2o.E_2o_1.E_2c_1.\overline{E_2`D}.E_2`D.E_2D'.E_2I \mid E_2s.E_2o.E_2o_1.E_2V \mid \\
& S_2z.\overline{S_2a} \mid S_2z.\overline{S_2r} \mid q.S_i \mid q.\alpha + \\
& U_1D_1o_1 \mid U_1D_1c_1.U_1D_1o_1.U_1D_1F \mid U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid U_1D_1e.U_1D_1A \mid \\
& U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid U_1o_1.U_1V \mid S_1z.\overline{S_1a} \mid \\
& S_1z.\overline{S_1r} \mid E_2s.E_2D_i \mid E_2s.E_2o.E_2a.\overline{E_2c*}.E_2D'.E_2I \mid S_2z.\overline{S_2a} \mid S_2z.\overline{S_2r} \mid q.S_i \mid q.\alpha + \\
& U_1D_1o_1 \mid U_1D_1c_1.U_1D_1o_1.U_1D_1F \mid U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid U_1D_1e.U_1D_1A \mid \\
& U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid U_1o_1.U_1V \mid S_1z.\overline{S_1a} \mid \\
& S_1z.\overline{S_1r} \mid E_2s.E_2D_i \mid E_2s.E_2o.E_2D'.E_2I \mid S_2z.\overline{S_2a} \mid S_2z.\overline{S_2r} \mid q.S_i \mid q.\alpha + \\
& U_1D_1o_1 \mid U_1D_1c_1.U_1D_1o_1.U_1D_1F \mid U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid U_1D_1e.U_1D_1A \mid \\
& U_1o_1.U_1c_1.\overline{U_1`D}.U_1`D.U_1D'.U_1I \mid U_1o_1.U_1V \mid S_1z.\overline{S_1a} \mid \\
& S_1z.\overline{S_1r} \mid E_2s.E_2D_i \mid E_2s.E_2e.E_2l.E_2r.E_2I \mid S_2z.\overline{S_2a} \mid S_2z.\overline{S_2r} \mid q.S_i \mid q.\alpha
\end{aligned}$$

$$P \xrightarrow{S_2p,S_2d} E_2e.E_2l.E_2r \mid q_2.S_3x.S_3p.E_3e.E_3l.E$$

Iteration 8

$$U \rhd \{E_2e, q, q\}$$

$$\begin{aligned}\alpha \xrightarrow{E_2e} {} & U_1D_1o_1 \mid U_1D_1c_1.U_1D_1o_1.U_1D_1F \mid U_1o_1.U_1c_1.\overline{U_1{`}D}.U_1{`}D.U_1D'.U_1I \mid U_1D_1e.U_1D_1A \mid \\ & U_1o_1.U_1c_1.\overline{U_1{`}D}.U_1{`}D.U_1D'.U_1I \mid U_1o_1.U_1V \mid S_1z.\overline{S_1a} \mid \\ & S_1z.\overline{S_1r} \mid E_2l.E_2r.E_2I \mid S_2z.\overline{S_2a} \mid S_2z.\overline{S_2r} \mid q.S_i \mid q.\alpha\end{aligned}$$

$$P \xrightarrow{E_2e} E_2l.E_2r \mid q_2.S_3x.S_3p.E_3e.E_3l.E$$

Iteration 9

$$U \rhd \{E_2l, q, q\}$$

$$\begin{aligned}\alpha \xrightarrow{E_2l} {} & U_1D_1o_1 \mid U_1D_1c_1.U_1D_1o_1.U_1D_1F \mid U_1o_1.U_1c_1.\overline{U_1{`}D}.U_1{`}D.U_1D'.U_1I \mid \\ & U_1D_1e.U_1D_1A \mid U_1o_1.U_1c_1.\overline{U_1{`}D}.U_1{`}D.U_1D'.U_1I \mid \\ & U_1o_1.U_1V \mid S_1z.\overline{S_1a} \mid S_1z.\overline{S_1r} \mid E_2r.E_2I \mid S_2z.\overline{S_2a} \mid S_2z.\overline{S_2r} \mid q.S_i \mid q.\alpha\end{aligned}$$

$$P \xrightarrow{E_2l} E_2r \mid q_2.S_3x.S_3p.E_3e.E_3l.E$$

Iteration 10

$$U \rhd \{E_2r, q, q\}$$

$$\begin{aligned}\alpha \xrightarrow{E_2r} {} & U_1D_1o_1 \mid U_1D_1c_1.U_1D_1o_1.U_1D_1F \mid U_1o_1.U_1c_1.\overline{U_1{`}D}.U_1{`}D.U_1D'.U_1I \mid \\ & U_1D_1e.U_1D_1A \mid U_1o_1.U_1c_1.\overline{U_1{`}D}.U_1{`}D.U_1D'.U_1I \mid \\ & U_1o_1.U_1V \mid S_1z.\overline{S_1a} \mid S_1z.\overline{S_1r} \mid E_2I \mid S_2z.\overline{S_2a} \mid S_2z.\overline{S_2r} \mid q.S_i \mid q.\alpha\end{aligned}$$

$$P \xrightarrow{E_2r} q_2.S_3x.S_3p.E_3e.E_3l.E$$

For brevity, the algorithm is terminated at this point because it is clear that the system state is similar to that of iteration 6 and the only remaining term in the process equation is also the same as that of iteration 6. Thus, in this case, it is not necessary to show the final steps. The anticipated completion of the process simulation demonstrates that, indeed, this process can be performed by the object model. A complete execution of the algorithm showing stack *V* and emptying state *V* would also demonstrate that this model is also completely achievable. Thus, the process model of Figure 7.7 can be used to enforce event eligibility on the object model of Figure 7.5 during run-time to guaranty that this process will complete.

8. CONCLUSION AND FUTURE WORK

The Formal Object Interaction Language (FOIL), as presented in this thesis, is a complete modeling language that can model software structure, behavior and process using a single unified notation. All aspects are reflected algebraically for analysis and verification. In this thesis, there have been three examples given of systems modeled using FOIL. These examples demonstrate all the major features and benefits of FOIL and provide a significant range of complexity.

While not addressed in this thesis, the complexity of modeling a system in FOIL is not substantially more difficult than standard Unified Modeling Language (UML) and likely to be simpler than Object Petri-nets (OPN). Experience in using FOIL for the examples in this thesis suggests that FOIL requires more abstract thinking than simpler languages, but with some practice is suitable for real-world applications. A cursory overview of FOIL suggests that it is ideally suited for an executable modeling language. At a minimum, FOIL is a springboard to spur renewed interests in formal graphical modeling languages.

8.1 Benefits and Limitations

The Formal Object Interaction Language (FOIL) is designed to be a complete and comprehensive graphical modeling language. FOIL is meant to have a user friendly graphical notation while providing more expressive power. It was intended that FOIL be able to model structure, behavior and process with a single notation, and with a common mathematical underpinning. Complete support for behavioral inheritance and concurrency were key design goals. Finally, the ability to verify that a process can be completed by an object model is a unique advantage. It is likely that there are modeling languages and frameworks that are superior to FOIL in one or more of these areas. This thesis was specifically written to show that FOIL is unique in its ability to perform well in ALL of these major design areas.

8.1.1 Graphical Notation

Graphically, the Formal Object Interaction Language (FOIL) is comprised of what has worked well in current modeling practices. The basic structure of the class diagram, as provided by UML, has remained effectively unchanged in FOIL. Many of the attributes and methods required in standard UML are not necessary in FOIL. The reason for this is that many of the attributes and methods in the UML notation are used to implement object behavior. Since FOIL represents behavior graphically (where the UML class diagram does not), many of these attributes and methods are specified in the behavioral portion of the class notation. Additionally, as used in the Business Object Notation (BON), attributes can be specified as read-only, while UML requires an attribute and method to accomplish this feature. The focus on FOIL structural modeling was to follow the example of UML but simplify the notation to avoid redundancies and allow room for behavioral specification without making the diagram overly complex.

The behavioral specification of classes in FOIL is a completely new notation but should look familiar, as a hybrid of simple state diagrams and Petri-nets. The choice to use ports to model variations of concurrent behavior stems from the desire to remove the ‘token’ concept from Petri-nets. In the Object Petri-net (OPN) notation, a “class” can basically function as a process or a token. This requires that the modeler know which function a class is performing in the model. The idea with FOIL was to keep the structure of a typical UML class diagram, where such distinctions are not necessary, while still providing complete support for concurrent behavior within and between objects.

Many of the notations for behavioral modeling were designed specifically to prevent the system diagram from becoming overly large or complex. This problem is well-known in other modeling languages, such as Petri-nets, but have had solutions offered by other languages such as YAWL and BON. Specifically, FOIL uses the concept of optional events (represented by a dotted line), that would require significantly more diagrammatic elements to represent with basic notations. Also, the notation for interleaving could be modeled as a serious of sequential steps encompassing all known possibilities but this quickly becomes incomprehensible as the number of sequential steps grows beyond three (see 8.2.1).

The focus of FOIL process modeling was to ensure a consistency with the FOIL modeling notation for structure and behavior. As such, the process model, from a high-level, flows much like many of the process modeling notations in current practice such as UML activity diagrams, YAWL, SEAM, and Business Process Diagram Notation. However, the internal behavior of processes can be represented by more complex specifications than most of these languages. This behavioral specification of processes in FOIL is done in the same way as that of objects. The goal, again, was to maintain similarity with current methods, where such features did not inhibit the ability of FOIL to model all know workflow patterns or ruin the ability of FOIL to be used for mathematical analysis and verification.

Finally, the behavioral notation of both classes and processes in FOIL was designed to ensure that the construction of the mathematical expressions could be done on a state-by-state basis. By maintaining this notational property, the mathematical construction assures that a system expression is the combination of all class or process expressions and that these expressions are a combination of their individual state expressions. This is critical in the scalability of the modeling language both graphically and mathematically.

8.1.2 Algebra

The FOIL algebra is heavily modeled after π-calculus. Since, some of the features in π-calculus, such as scoping, are not necessary, the process algebra expressions in FOIL are simplified. The algebraic construction of a system is done in a bottom-up fashion allowing for progressively more complex models to be built while assuring that, if graphical conventions are followed, there is always a corresponding algebraic representation.

The elements that truly make FOIL useful for mathematical verification are graphically implicit. This allows for fairly complex analysis of a FOIL model without adding significantly complex graphical constructs. The concept of *event scope* is added to ensure that mathematical reductions can have sufficient granularity and selectivity in their response to the system. The added concepts of *unique* and *non-unique* events are used to ensure that a *reduction eligibility rule* could be provided to ensure run-time and design-time verification of system state. Finally, the concept of *non-events* is given to allow for an externally responsive system where certain actions are optional without requiring excessively large graphical representation.

The construction of FOIL algebraic representations is done on a state-by-state basis. After construction, the various laws and identities offered for the FOIL algebra allow for the manipulation of the algebraic expressions for use by the system during run-time. The main purpose for providing these mathematical processes is to make construction, manipulation and verification simple to perform either manually or by a computer system.

Finally, the reductions performed during run-time have a predictable algorithm and have strong performance characteristics. The reduction eligibility rule is checked prior to reduction to ensure system stability during run-time. All of the algorithms given for algebraic construction, manipulation, execution and verification make FOIL suitable as a directly executable modeling language.

8.1.3 Behavioral Inheritance

Inheritance is a concept that allows a large system to grow without the need to recode elements that exhibit common structure. Inheritance is a well-known and studied concept in object-oriented design and development; however, most research and implementation centers on the concept of interface conformity. FOIL allows for an optionally more strict interpretation of inheritance to ensure both structural (interface) and behavioral conformity. Thus, with this new stricter interpretation of inheritance, the code savings involved in "inheriting" classes from more generalized classes are much larger. Extending a class both structurally and behaviorally means that code for interaction of the class with the encompassing system and internal control flow of actions within the object are already specified. Ensuring behavioral inheritance is a simple algorithm done on the FOIL algebraic expressions, once again, making this feature suitable for enforcement by any underlying executable system.

8.1.4 Concurrency

Because the behavior notation is derived from Petri-nets and the algebraic representation is derived from π-calculus, FOIL is built on previous advances that have, as one of their key features, support for concurrency. Thus, it is not surprising that FOIL has inherent support for concurrency. This concurrency support makes FOIL suitable for modeling complex distributed systems. The literature review performed for this thesis indicates that FOIL is likely to be the only modeling language which can be used to generate multi-threaded source code without explicit thread modeling.

Concurrency modeling is useful beyond simple distributed systems. By modeling the internal behavior of active states, certain choices about how software handles concurrent events and processing can be made. In this thesis, most examples involve sequentially processed events with responsive behavior from concurrent threads; however, active state modeling provides a clear mechanism for responding to concurrent events on single threads.

FOIL concurrency modeling does not implicitly enforce resource dependency or race conditions. These must be considered when modeling any system using FOIL. In addition, concurrency is graphically represented in-line with other system features whereas other languages have chosen to do this outside of basic structural diagramming. As such, FOIL does require more abstract thinking on the part of the modeler than those modeling languages without concurrency support.

Finally, FOIL's inherent support for concurrency gives it the ability to model all known and studied workflow patterns. While there are many process languages that have support for these patterns, many of them do not have a formal semantic or object-orientation. FOIL's ability to do all of these things makes it truly unique among modeling languages.

8.1.5 Model Analysis and Verification

The underlying algebraic representation of a FOIL model, combined with the various mathematical laws and identities, allows for broad analysis of systems prior to implementation. This thesis presents the basic ideas of object state reachability, inherent inconsistency, and deadlock potential, as design-time analyses which can be performed on a FOIL object system. Reachability and inconsistency can be determined during run-time as well. Thus, with FOIL, a system could be designed to avoid these undesirable conditions. Additionally, the occurrence of a deadlock can be detected using FOIL allowing run-time events to be rejected if they are found to result in a deadlock condition.

More impressive is the ability of FOIL to respond algebraically to events as part of a simulation. This simulation capability was shown to be useful in performing analysis on processes as they relate to an object model. Using the algorithm provided in this thesis, FOIL can determine process achievability, complete achievability, and determinism. If a process is determined to be "completely achievable" then this thesis showed that such a process can be used as a run-time constraint on an object model to ensure that a process will always complete.

8.1.6 Limitations

The intended purpose of the Formal Object Interaction Language (FOIL) is to simplify and enhance the design and implementation of software. Other areas of software engineering, such as requirements gathering and analysis, hardware infrastructure design, and software deployment are not addressed by the FOIL model.

FOIL is ideally suited for interactive or reactive systems that are object-oriented or service-oriented in nature. This covers a large segment of the software being developed today. FOIL is very expressive and if the details of active states are specified, it can be used to fully generate application or executable code. The initial basis for the development of FOIL was as a formal object-oriented language as the foundation for a workflow management system [77] and thus, it is well suited for this purpose.

FOIL is not a requirements gathering or system deployment notation and thus is not suitable for those purposes. Good design of software would dictate the use of UML Use Case diagrams for requirements modeling, while package, system and deployment diagrams would still be used for their independent purposes. The FOIL diagram can take advantage of requirements specifications as demonstrated by the passenger “actor” in Figure 3.12. FOIL is primarily suited for the design and implementation of the software once the requirements have been determined.

FOIL may not be the modeling notation of choice for some applications. FOIL’s abbreviated notation for attributes and their access make it less suitable for applications without a significant behavior component. Thus, if the main feature of an application is the storage and retrieval of objects, attributes or data, the FOIL notation offers little advantages over other options. However, a system which would require one or more UML sequence or state diagrams to specify behavior would benefit from the FOIL notation.

Mathematically, FOIL is not temporal as are other languages [13, 34, 63] and thus would not be suitable for real-time or discrete event systems that must have an inherent mathematical concept of time. However, it is possible that FOIL could be extended to support a temporal semantic.

8.2 Future Work

While this thesis has attempted to present a complete picture of the Formal Object Interact Language (FOIL) and provide sufficient depth so as to appreciate its benefits and uses, the subject of software modeling, in general, is very broad. The successful blending of structure, behavior, and process in a graphical and formal manner has raised potential issues that need to be addressed, uses that need to be attempted, and extensions that need to be explored.

8.2.1 State Explosion

One of the primary issues related to using state-based analysis of systems is the state explosion problem. It should be relatively easy to surmise that the algebraic manipulations performed during reductions as well as the application of the reduction eligibility rule during run-time are really just a form of state-based analysis. The main problem with state-based analysis is that the number of state options grows exponentially resulting in some tractability problems involved with analytical algorithms. State explosion is a known problem with process algebra [78] but is not unique to FOIL. Solutions have been offered for other modeling languages such as Petri-nets [79].

Most of the examples in this thesis use the choice-action form (CAF) as the basic mathematical form for run-time execution and analysis. However, this form grows exponentially for certain control flow patterns. Specifically, interleaved routing and multiple choices are two patterns that exhibit this problem early in the mathematical process. FOIL has some notations designed to eliminate this problem from a graphical standpoint; however, these notations do little to minimize the growth rate of the underlying algebraic expressions.

It was briefly mentioned in this thesis, that an alternate algebraic form can be used to prevent the state explosion problem. This form, called the choice-compressed form (CCF), delays the expansion of choices until the last possible moment. Preliminary work suggests that reductions can be done on expressions in CCF, but that such rules are far more complicated than their CAF counterparts. While it seems logical that such rules could be proven and codified, this has only been done on a very basic level. Additionally, the research on using CCF is incomplete. For instance, while basic reductions and analyses have been explored using CCF, the achievability algorithm has not been attempted.

8.2.2 Process Metrics

The ability for FOIL to determine whether a given process is achievable with a given object system is a distinctive feature of FOIL. A thorough survey suggests that there is no modeling language offered today with this capability. The FOIL achievability algorithm shows that there is a coupling between an object model and the processes it is designed to achieve. With other modeling languages this coupling is implicit or inferred; while in FOIL, it is explicit and verifiable.

The FOIL achievability algorithm performs its work by executing a simulation of events in the system based on expected process results. Many of the events simulated, however, do not actually show up in the process model. In other words, it may be that in order to determine that a given process is achievable; the *assumption* of an event sequence of *n* length is required. Yet a more detailed process model may be determined to be achievable with the same object model with only *n-3* event *assumptions*.

Another possible metric is to complete the achievability algorithm even after achievability is determined. If the process model is determined to be *achievable* but not *completely achievable*, then there is the possibility of placing a coined *achievability index* to the system. This would represent the number or magnitude of internal control flow paths inside the object model that do not lead to *achievability*. This could be represented as a number or a percentage of the total number or magnitude of control flow paths.

These two possible metrics are merely given as a suggestion or beginning on what may be possible with future research into this area. Likely, further contemplation would reveal many more possible measurements that could be performed on system models created with FOIL. The main focus of this thesis has been on model production, execution, and verification with little attention given to model optimization. The formulation and understanding of such metrics derived with FOIL models would open a whole avenue of research into FOIL model optimization.

8.2.3 Process Mining

Process mining is a technique used to generate a process model from the transaction logs of existing systems. These systems are usually transactional and procedural in nature. The problem of process mining is not an easy one, as all systems show variations in their logging capability, and methods for computer analysis of such logs are necessarily complex. Despite this, process mining holds much promise, as a tool for business analysis, to reduce the time required to model as-is business processes. Also, mining techniques can be used to determine if the operations of a system correspond with the designed intent.

There has been a fair amount of research into mining process logs. EMiT is a low-level process mining tool that can be used to read event logs and determine the workflow structure of the underlying system [80]. One of the notable advancements offered by this tool is the use of an intermediate XML log format to which logs from various applications are converted. The EMiT system was made part of a larger workflow mining tool called TeamLog [81]. The InWoLvE workflow mining processor uses a more inductive approach and essentially solves the problem of task-oriented workflow mining in two steps[82]. First it derives a stochastic activity graph (SAG) from a given log and then combines repeated activities at the end.

The Process Miner was a product whose theoretical foundation and program implementation ware done almost exclusively by Guido Schimm. The first iteration of the product [83] was based on his ideas presented in 2000 [84]. It differs from other approaches in that it extracts an exact model of the workflow based on the logs. It also presents its model in a block-oriented fashion. Using this model type, a process model will have an algebra that has distributive, associative and commutative properties [85] much like FOIL algebra.

Since a FOIL object model is an event driven system, it is easy to contemplate how a workflow or process analyzer could be implemented. As each event is received, the associated reductions in the algebraic expressions are recorded. Then such event reductions could be mined to determine the probabilities of various event sequences. Based on this idea, a FOIL process model could be created. A FOIL process which is determined to be achievable may still have other processes that are more prevalent. This generated process model would be useful as an informational tool to determine what work is actually being done by a given FOIL object model. Additionally a generated process model might serve as an aid to process and object model designers.

8.2.4 Distribution

There have been a large number of techniques introduced to provide scalable, distributed workflow services. These solutions range from purely event-driven models [86] to grid computing architectures [87]. ***One of the motivations for FOIL was in creating an object-oriented workflow management system.*** As such, system distribution has been a concern during development of FOIL but has not been fully addressed.

8.2.4.1 WfMC Reference Model

In 1993, the Workflow Management Coalition (www.wfmc.org) was formed to help standardize the industry with respect to workflow management systems. Their efforts have been only partially successful, but they have introduced modeling structures for building workflow systems to support scalability and interoperability. Figure 8.1 shows a diagram of the proposed reference model for workflow systems [88]. This model suggests that interfaces be standardized to allow for connections between different systems.

Distribution through this basic model is realized through the interaction (Interface 4) of a workflow engine with other existing workflow engines in addition to the ability to invoke outside applications (Interface 3) from within the engine. This model is very basic and does not take into account some of the more complex issues with distribution. For example, this model assumes that the Workflow Client Application will always be connected to a central workflow engine. In

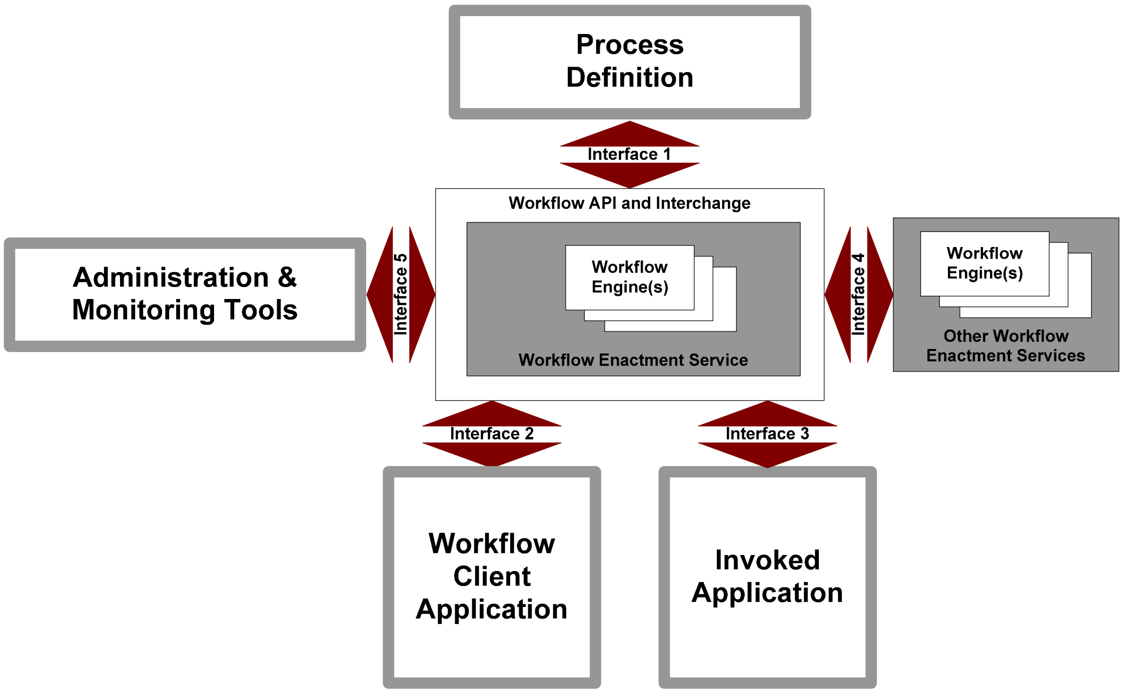

Process Definition
Interface 1
Workflow API and Interchange
Administration & Monitoring Tools
Interface 5
Workflow Engine(s)
Workflow Enactment Service
Interface 4
Workflow Engine(s)
Other Workflow Enactment Services
Interface 2
Interface 3
Workflow Client Application
Invoked Application

Figure 8.1 WfMC Reference Model [88]

large scale implementations, the client may not be aware of the location of the closest workflow engine. Additionally, whether invocation of remote applications is synchronous or asynchronous and how these decisions affect the engine is unspecified.

8.2.4.2 Physical-Logical Separation

One of the key issues in interoperability is the desire to abstract the interface to a system away from its underlying platform implementations. This was the intent of the first WfMC model; however, this approach is rather simplistic dealing with just mere interoperability without regard for redundancy, load-balancing and geographic scalability.

One approach involved the use of assignment servers [89, 90]. An assignment server is a separate machine or program which has knowledge of the location and physical requirements of multiple workflow servers. When a client requests needs to perform a task, the message is sent to an assignment server which will then pass on the request to the appropriate workflow server. Thus, the assignment server functions as a translator for the target machine making the platform issues with interfacing with the server transparent to the client. One similar approach was to create workflow repositories that serve the same function as the assignment server but also stores the interfaces for each workflow[91, 92].

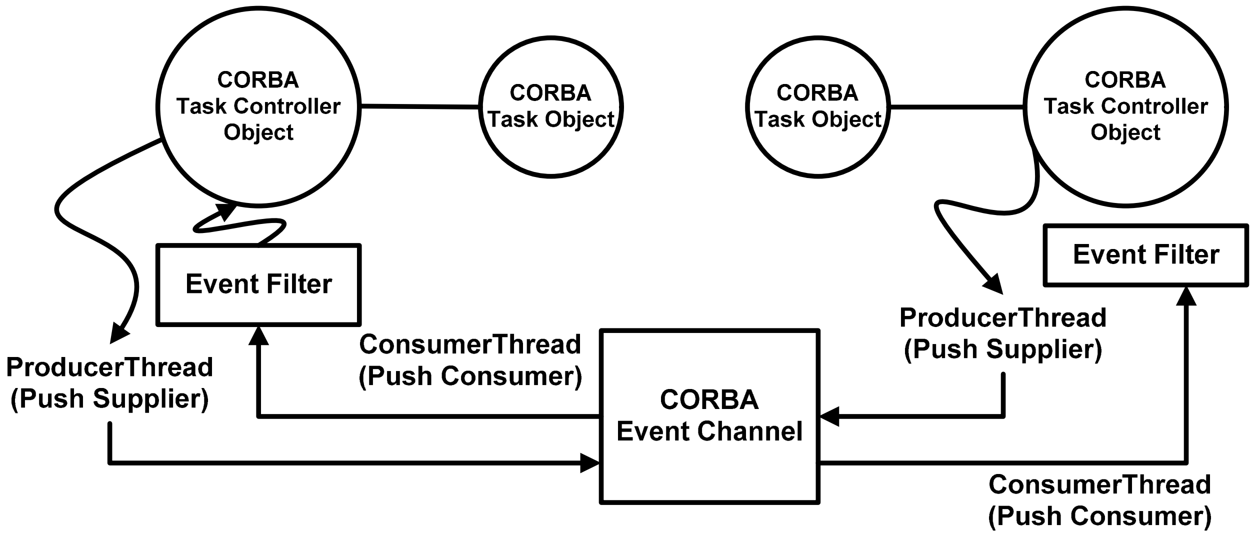

CORBA Task Controller Object
CORBA Task Object
CORBA Task Object
CORBA Task Controller Object
Event Filter
Event Filter
ProducerThread (Push Supplier)
ConsumerThread (Push Consumer)
CORBA Event Channel
ProducerThread (Push Supplier)
ConsumerThread (Push Consumer)

Figure 8.2 Event-Driven WFMS using CORBA [86]

Figure 8.2 [86] shows the basic setup of this idea within a CORBA framework. Each process registers with a central event channel as to which events it listens for. In addition, it registers which events it provides. Thus, each process is both a consumer and producer of events. The event channel has filters which ensure that events are only sent to those processes for which it is applicable. This extends beyond just mere registry but the event channel will also take into account the sequence and data involved in the event in determining applicability. To some degree, the event channel with its associated filters acts as a workflow engine, making decisions on behalf of the processes under its charge. However, the work is performed completely by the target objects and the event channel is completely unaware of the logic, data manipulation, implementation or platform of the processes.

Each object as modeled in FOIL can be decoupled with a central event controller as offered by CORBA or other workflow-based systems [88]. This decoupling allows for distributed or mobile objects to interact under a defined service-based interface. Additionally, the security services that enforce the interaction between objects can be more strictly specified than in typical object-oriented implementation. For example, in Figure 3.12, it may be necessary to ensure that a *reachedFloor* event can only be fired by the elevator and no other object. In typical object-oriented design, the *reachedFloor* method is public and thus accessible to all objects. FOIL with its decoupling capability and inherent support for concurrency is an ideal candidate to be considered for distributed system design in the future.

REFERENCES

[1] E. F. Codd, "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, no. 6, pp. 377-387, 1970.

[2] P. P.-S. Chen, "The entity-relationship model—toward a unified view of data," *ACM Transactions on Database Systems*, vol. 1, no. 1, pp. 9-36, 1976.

[3] T. Rentsch, "Object oriented programming," *SIGPLAN Not.*, vol. 17, no. 9, pp. 51-57, 1982.

[4] G. Booch, *Object-Oriented Analysis and Design with Applications*, 2nd ed. Redwood City: Benjamin Cummings, 1993.

[5] T. Halpin, "Data modeling in UML and ORM revisited," in *The Proceedings of Workshop on Evaluation of Modeling Methods in Systems Analysis and Design*. Heidelberg, Germany, 1999.

[6] C. A. Petri, *Kommunikation mit Automaten* PhD Thesis: Bonn: Institut fur lnstrumentelle Mathematik, 1962.

[7] R. Milner, *A Calculus of Communicating Systems*. Secaucus, NJ: Springer-Verlag New York, Inc, 1982.

[8] R. Milner, *Communicating and mobile systems: the π-calculus*: Cambridge University Press, 1999.

[9] H. Giese, J. Graf, and G. Wirtz, "Seamless Visual Object-Oriented Behavior Modeling for Distributed Software Systems," in *Proceedings of the IEEE Symposium on Visual Languages*: IEEE Computer Society, 1999.

[10] S.-K. Kim and D. Carrington, "Formalizing the UML Class Diagram Using Object-Z," *Lecture Notes in Computer Science*, pp. 83, 1999.

[11] P. Bichler and M. Schrefl, "Active object-oriented database design using active object/behavior diagrams," in *Fourth International Workshop on Research Issues in Data Engineering*: IEEE, 1994, pp. 163.

[12] D. Coleman, F. Hayes, and S. Bear, "Introduction Objectcharts or How to Use Statecharts in Object-Oriented Design," *IEEE Trans. Softw. Eng.*, vol. 18, no. 1, pp. 9-18, 1992.

[13] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas, "TROLL - A Language for Object-Oriented Specification of Information Systems," *ACM Transactions on Information Systems*, vol. 14, no. 2, pp. 175-211, 1996.

[14] D. Harel and E. Gery, "Executable Object Modeling with Statecharts," in *IEEE 18th International Conference on Software Engineering*, 1996.

[15] J. A. Saldhana and S. M. Shatz, "UML Diagrams to Object Petri Net Models: An Approach for Modeling and Analysis," in *International Conference on Software Engineering and Knowledge Engineering*. Chicago, Illinois, 2000.

[16] G. Kappel and M. Schrefl, "Object/behavior diagrams," in *Seventh International Conference on Data Engineering*. Kobe, 1991, pp. 530.

[17] X. Blanc, M. P. Gervais, and R. Le-Delliou, "Using the UML language to express the ODP enterprise concepts," in *Third International Enterprise Distributed Object Computing Conference*. Mannheim, Germany: IEEE, 1999, pp. 50-59.

[18] A. Kleppe and J. Warmer, "Making UML activity diagrams object-oriented," in *Proceedings of the Technology of Object-Oriented Languages and Systems*: IEEE, 2000, pp. 288.

[19] M. Peleg and D. Dori, "From Object-Process Diagrams to Natural Object-Process Language," in *4th International Workshop on Next Generation Information Technologies and Systems*, vol. 1649, R. Y. Pinter and S. Tsur, Eds.: Springer-Verlag, 1999, pp. 221-228.

[20] W. M. P. v. d. Aalst, A. H. M. t. Hofstede, B. Kiepuszewski, and A. P. Barros., "Workflow Patterns," *Distributed and Parallel Databases*, vol. 14, no. 3, pp. 5-51, 2003.

[21] S. A. White, "Process Modeling Notations and Workflow Patterns," in *BPTrends*, 2004.

[22] A. Bajaj and S. Ram, "SEAM: A state-entity-activity-model for a well-defined workflow development methodology," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 14, no. 2, pp. 415, 2002.

[23] G. Kappel, S. Rausch-Schott, and W. Retschitzegger, "A framework for workflow management systems based on objects, rules and roles," *ACM Computing Survey*, vol. 32, no. 1, pp. 27, 2000.

[24] W. M. P. van der Aalst and A. H. M. ter Hofstede, "YAWL: Yet Another Workflow Language," Queensland University of Technology, Brisbane, QUT Technical report FIT-TR-2003-04, 2003.

[25] P. Coad and E. Yourdon, *Object Oriented Design*: Prentice Hall, 1991.

[26] S. Grumbach and V. Vianu, "Tractable query languages for complex object databases," in *Proceedings of the tenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. Denver, Colorado, United States: ACM Press, 1991.

[27] A. M. Alashqur, S. Y. W. Su, and H. Lam, "OQL: a query language for manipulating object-oriented databases," in *Proceedings of the 15th international conference on Very large data bases*. Amsterdam, The Netherlands: Morgan Kaufmann Publishers Inc., 1989.

[28] W. M. P. van der Aalst, B. F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A. J. M. M. Weijters, "Workflow mining: a survey of issues and approaches," *Data Knowl. Eng.*, vol. 47, no. 2, pp. 237-267, 2003.

[29] N. Turbit, "Business Process Modeling Overview" available at http://www.projectperfect.com.au/info_business_process_modelling_overview.php, accessed in 2005.

[30] W. v. d. Aalst, "Workflow Patterns" available at http://www.workflowpatterns.com, accessed in 2005.

[31] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros, "Workflow Patterns," *Distrib. Parallel Databases*, vol. 14, no. 1, pp. 5-51, 2003.

[32] R. M. Bastos and D. D. A. Ruiz, "Extending UML activity diagram for workflow modeling in production systems," in *Proceedings of the 35th Hawaii International Conference on System Sciences*: IEEE, 2002.

[33] M. Bjerkander and C. Kobryn, "Architecting systems with UML 2.0," *Software, IEEE*, vol. 20, no. 4, pp. 57, 2003.

[34] M. Huadong, "A Workflow Model Based on Temporal Logic," in *Proceedings of the 8th International Conference on Computer Supported Cooperative Work in Design*, vol. 2. Beijing, China: IEEE, 2004, pp. 327.

[35] M. M. Kwan and P. R. Balasubramanian, "Dynamic workflow management: a framework for modeling workflows," in *Proceedings of the 30th International Conference on System Sciences: Information Systems Track—Internet and the Digital Economy*, vol. 4. Hawaii: IEEE, 1997, pp. 367.

[36] W. M. P. van der Aalst, K. M. van Heez, and G. J. Houben, "Modelling and analysing workflow using a Petri-net based approach," Eindhoven University of Technology, Dept. of Mathematics and Computing Science, Eindhoven, The Netherlands. 2003.

[37] Y. Wenqi and L. Feng, "Workflow modeling: a structured approach," in *The 8th International Conference on Computer Supported Cooperative Work in Design*, vol. 1. Beijing, China, 2004, pp. 433.

[38] J. Yueping, W. Zhaohui, D. Shuiguang, and Y. Zhen, "Service-Oriented Workflow Model," in *Proceedings of the 19th International Conference on Advanced Information Networking and Applications - Volume 2*, vol. 2: IEEE, 2005, pp. 484.

[39] W. M. P. v. d. Aalst, "Don't go with the flow: Web services composition standards exposed," *IEEE Intelligent Systems*, Jan/Feb 2003.

[40] H. Xudong, "Formalizing UML class diagrams-a hierarchical predicate transition net approach," in *24th International Computer Software and Applications Conference*: IEEE, 2000, pp. 217.

[41] M. Richters and M. Gogolla, "On Formalizing the UML Object Constraint Language OCL," *Lecture Notes in Computer Science*, pp. 449, 1998.

[42] A. Mar, a. Funes, and C. George, "Formalizing UML class diagrams," in *UML and the unified process*: Idea Group Publishing, 2003, pp. 129-198.

[43] A. Welc, S. Jagannathan, and A. Hosking, "Safe futures for Java," in *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*. San Diego, CA, USA: ACM Press, 2005.

[44] M. B. Dwyer, L. A. Clarke, J. M. Cobleigh, and G. Naumovich, "Flow analysis for verifying properties of concurrent software systems," *ACM Trans. Softw. Eng. Methodol.*, vol. 13, no. 4, pp. 359-430, 2004.

[45] E. Battiston, A. Chizzoni, and F. D. Cindio, "CLOWN as a Testbed for Concurrent Object-Oriented Concepts," in *Concurrent Object-Oriented Programming and Petri Nets: Advances in Petri Nets*: Springer-Verlag, 2001, pp. 131.

[46] A. Ahern and N. Yoshida, "Formalising Java RMI with explicit code mobility," in *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*. San Diego, CA, USA: ACM Press, 2005.

[47] A. Ferrara, "Web Services: A Process Algebra Approach," in *International Conference On Service Oriented Computing*: ACM Press, 2004.

[48] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541, 1989.

[49] C. Lakos, "The Challenge of Object Orientation for the Analysis of Concurrent Systems," in *Applications and Theory of Petri Nets 2002: 23rd International Conference, Proceedings*. Adelaide, Australia, 2002, pp. 59.

[50] X. Amatriain, "An Object-Oriented Metamodel for Digital Signal Processing" available at http://www.iua.upf.es/~xamat/Thesis/html/Thesis_forHTML.html, accessed in 2004.

[51] J. Esparza and M. Nielsen, "Decidability Issues for Petri Nets - a Survey," *Journal of Informatics, Processing and Cybernetics*, vol. 30, no. 3, pp. 143-160, 1994.

[52] P. Huber, K. Jensen, and R. M. Shapiro, "Hierarchies in Coloured Petri Nets," in *Proceedings on Advances in Petri nets 1990*: Springer-Verlag New York, Inc., 1991.

[53] K. Jensen, "Coloured Petri nets," in the Proceedings of IEEE Colloquium on Discrete Event Systems: A New Challenge for Intelligent Control Systems, London, 1993.

[54] K. Jensen, "High-Level Petri Nets," in *Selected Papers from the 3rd European Workshop on Applications and Theory of Petri Nets*: Springer-Verlag, 1983.

[55] O. M. G. (OMG). "UML Object Constraint Language Specification. Version 1.5," Rational Software Corporation, Santa Clara, CA-95051, USA March 2003.

[56] A. Church, "An Unsolvable Problem of Elementary Number Theory," *American Journal of Mathematics*, vol. 58, no. 2, pp. 345-363, 1936.

[57] H. Barendregt, *The lambda calculus, its syntax and semantics*: North-Holland, 1984.

[58] C. Barker, "Lambda Calculus Tutorial" available at http://ling.ucsd.edu/~barker/Lambda/, accessed in 2005.

[59] C. A. R. Hoare, *Communicating Sequential Processes*: Prentice Hall, 1985.

[60] J. Parrow, "An Introduction to the π-Calculus," in *Handbook of Process Algebra*, Bergstra, Ponse, and Smolka, Eds.: Elsevier, 2001, pp. 479-543.

[61] P. Sewell, "Applied Pi Calculus," University of Cambridge, July 28, 2000, pp. 65.

[62] H. Smith and P. Fingar, "Workflow is just a Pi Process," Computer Sciences Corporation, 2003.

[63] C. Combi and G. Pozzi, "Architectures for a temporal workflow management system," in *Proceedings of the 2004 ACM symposium on Applied computing*. Nicosia, Cyprus: ACM Press, 2004.

[64] W. M. P. van der Aalst, "Making Work Flow: On the Application of Petri Nets to Business Process Management," *Lecture Notes in Computer Science*, pp. 1, 2002.

[65] D. Dori, "Object-process methodology applied to modeling credit card transactions," in *Advanced topics in database research vol. 1*: Idea Group Publishing, 2002, pp. 87-105.

[66] D. Dori, "Object-Process Methodology Website" available at http://www.objectprocess.org, accessed in 2005.

[67] C. Lakos, "From Coloured Petri Nets to Object Petri Nets," in *Proceedings of the 16th International Conference on Application and Theory of Petri Nets*: Springer-Verlag, 1995, pp. 278 - 297.

[68] C. Lakos, "Object Oriented Modelling with Object Petri Nets," in *Concurrent Object-Oriented Programming and Petri Nets: Advances in Petri Nets*: Springer-Verlag, 2001, pp. 1 - 37.

[69] O. Biberstein, D. Buchs, and N. Guelfi, "Object-Oriented Nets with Algebraic Specifications: The CO-OPN/2 Formalism," in *Concurrent Object-Oriented Programming and Petri Nets: Advances in Petri Nets*: Springer-Verlag, 2001, pp. 73.

[70] J. o. P. Barros and L. Gomes, "On the Use of Coloured Petri Nets for Object-Oriented Design," in *Applications and Theory of Petri Nets 2004*: Springer-Verlag, 2004, pp. 117.

[71] L. Sea and L. Seng Wai, "Advanced Petri Nets for modelling mobile agent enabled interorganizational workflows," in *Proceeding of the 9th IEEE International Conference on Engineering of Computer-Based Systems*: IEEE, 2002, pp. 245.

[72] P. Sanchez, P. Letelier, and I. Ramos, "Constructs for Prototyping Information Systems with Object Petri Nets.," in *Proceedings of IEEE Systems, Man and Cybernetics*, vol. 5. Orlando (USA): IEEE, October 1997, pp. 4260-4265.

[73] N. Aoumeur and G. Saake, "Towards an Object Petri Nets Model for Specifying and Validating Distributed Information Systems," in *Advanced Information Systems Engineering: 11th International Conference, CAiSE'99*. Heidelberg, Germany: Springer, 1999, pp. 381.

[74] D. Harel and O. Kupferman, "On object systems and behavioral inheritance," *IEEE Transactions on Software Engineering*, vol. 28, no. 9, pp. 889-903, 2002.

[75] R. F. Paige and J. S. Ostroff, "Comparison of the Business Object Notation and the Unified Modeling Language," *Lecture Notes in Computer Science*, pp. 67, 1999.

[76] E. Horowitz, S. Sahni, and D. Mehta, *Fundamentals of Data Structures in C++*. New York, NY: W.H. Freeman and Company, 1995.

[77] W. M. P. van der Aalst, A. H. M. ter Hofstede, and M. Weske, "Business Process Management: A Survey," *Lecture Notes in Computer Science*, pp. 1, 2003.

[78] W. J. Yeh and M. Young, "Compositional reachability analysis using process algebra," in *Proceedings of the symposium on Testing, analysis, and verification*. Victoria, British Columbia, Canada: ACM Press, 1991.

[79] M. Heiner, "Verification and Optimization of Control Programs by Petri Nets without State Explosion," in the Proceedings of 2nd Int. Workshop on Manufacturing and Petri Nets, Toulouse, 1997.

[80] B. F. van Dongen and W. M. P. van der Aalst, "EMiT: A Process Mining Tool," *Lecture Notes in Computer Science*, pp. 454, 2004.

[81] S. Dustdar, T. Hoffman, and W. M. P. van der Aalst, "Mining of ad-hoc business processes with TeamLog," Technical University of Vienna TUV-1841-2004-07, 2004.

[82] J. Herbst and D. Karagiannis, "Workflow mining with InWoLvE," *Computers in Industry*, vol. 53, no. 3, pp. 245, 2004.

[83] G. Schimm, "Process Miner - A Tool for Mining Process Schemes from Event-Based Data," *Lecture Notes in Computer Science*, pp. 525, 2002.

[84] G. Schimm, "Generic linear business process modeling.," in *International Workshop on Conceptual Modeling Approaches for E-Business*. Salt Lake City: Springer, 2000.

[85] G. Schimm, "Mining exact models of concurrent workflows," *Computers in Industry*, vol. 53, no. 3, pp. 265, 2004.

[86] Z. Tari and V. Pande, "Dynamic workflow management in CORBA distributed object systems," in *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*: IEEE, 2000, pp. 51.

[87] C. Junwei, S. A. Jarvis, S. Saini, and G. R. Nudd, "GridFlow: workflow management for grid computing," in *Proceedings of the 3rd International Symposium on Cluster Computing and the Grid*: IEEE, 2003, pp. 198.

[88] D. Hollingsworth, "The Workflow Reference Model: 10 Years On," in *Workflow Handbook 2004*. United Kingdom: Fujitsu Services, 2004.

[89] H. Schuster, J. Neeb, and R. Schamburger, "A configuration management approach for large workflow management systems," in *Proceedings of the international joint conference on Work activities coordination and collaboration*. San Francisco, California, United States: ACM Press, 1999.

[90] H. Schuster and P. Heinl, "A workflow data distribution strategy for scalable workflow management systems," in *Proceedings of the 1997 ACM symposium on Applied computing*. San Jose, California, United States: ACM Press, 1997.

[91] J. Neeb, M. Schlundt, and H. Wedekind, "Repositories for workflow-management-systems in a middleware environment," in *Proceedings of the 33rd International Conference on System Sciences*. Hawaii: IEEE, 2000, pp. 10 pp. vol.2.

[92] OMG, "Common Object Request Broker Architecture: Core Specification Version 3.0.3" available at http://www.omg.org/docs/formal/04-03-01.pdf, accessed in March 2004.