A TOPOLOGICAL MODEL FOR PARALLEL ALGORITHM DESIGN

DISSERTATION

Jeffrey A Simmers
Captain, USAF

AFIT/DS/ENG/91-02

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/DS/ENG/91-02

A TOPOLOGICAL MODEL FOR PARALLEL ALGORITHM DESIGN

DISSERTATION

Jeffrey A Simmers
Captain, USAF

AFIT/DS/ENG/91-02

Approved for public release; distribution unlimited

AFIT/DS/ENG/91-02

A TOPOLOGICAL MODEL FOR PARALLEL ALGORITHM DESIGN

DISSERTATION

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

Jeffrey A Simmers, B.S.E.E., M.S.E.E.

Captain, USAF

September, 1991

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1991	PhD Dissertation

4. TITLE AND SUBTITLE

A Topological Model For Parallel Algorithm Design

5. FUNDING NUMBERS

6. AUTHOR(S)

Jeffrey A Simmers, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/DS/ENG/91-02

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This research demonstrates that formal, mathematical analysis in theoretical computer science can be recast in terms of the topology of complete metric spaces, and also presents a methodical technique for developing formal specifications. This effort shows that the topology of complete metric spaces provides a tool that can be used to both recreate major results about computational models and also to develop new results about these models. Using the two computational models CSP and UNITY, this effort shows that the required mathematics needed to support this alternative to the traditional analysis of computational models can be readily supported by a standard course sequence in real analysis. Since the approach of proving programs correct after being written has not been widely accepted, this effort presents an alternative approach based on the developed topological framework for the formal specification language UNITY. This approach, designed to be automated, uses a set of transformations applied to UNITY specifications that preserve desired program properties.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Computer Science, Computational Models, Formal Specifications, Parallel Computation, Specification Transformations.	394
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

AFIT/DS/ENG/91-02

A TOPOLOGICAL MODEL FOR PARALLEL ALGORITHM DESIGN

Jeffrey A Simmers, B.S.E.E., M.S.E.E.

Captain, USAF

Approved:

Gary B Lamont 14 June 91

Paul D Baila 14 JUN 91

Mark E Oxley 14 June 91

Will P Bail 14 Jun 91

J.S. Przemieniecki 14 Jun 91

J.S. Przemieniecki

Institute Senior Dean

Preface

The research results documented here represent my attempt to extend the usefulness of the formal specification as a tool in the software development process; and to expand the number of students who can obtain the necessary mathematical background to conduct practical and theoretical computer science and engineering. The extension of the formal specification is accomplished by presenting a methodology for developing such specifications through techniques that could lead to automated support. And by presenting an alternative mathematical development of computational models based on the topology of complete metric spaces, this effort supports efforts to increase the ability of colleges and universities to supply the formalism needed for computer science and engineering research.

I thank my advisor Dr. Gary Lamont for all of his support and inspiration during this research effort, and I thank my committee of Dr. Lamont, Major (Dr.) Paul Bailor, and Dr. Mark Oxley for their help in conducting this research and writing this dissertation.

My greatest thanks go to my wife Susan, my sons Steven and David, and my daughter April, for their support, encouragement, sacrifices, and love during this trying time.

Jeffrey A Simmers

Table of Contents

Figure Page

Abstract

This research demonstrates that formal, mathematical analysis in theoretical computer science can be recast in terms of the topology of complete metric spaces, and also presents a methodical technique for developing formal specifications. This effort shows that the topology of complete metric spaces provides a tool that can be used to both recreate major results about computational models and also to develop new results about these models. Using the two computational models CSP and UNITY, this effort shows that the required mathematics needed to support this alternative to the traditional analysis of computational models can be readily supported by a standard course sequence in real analysis.

Since the approach of proving programs correct after being written has not been widely accepted, this effort presents an alternative approach based on the developed topological framework for the formal specification language UNITY. This approach, designed to be automated, uses a set of transformations applied to UNITY specifications that preserve desired program properties.

A TOPOLOGICAL MODEL FOR PARALLEL ALGORITHM DESIGN

I. Introduction

So now I know everything anyone knows
From beginning to end. From the start to the close.
Because Z is as far as the alphabet goes.

- Dr. Seuss *On Beyond Zebra*

Although this quote was originally intended to be somewhat humorous, today it addresses a much more serious problem within the computer science and software engineering fields. I may know everything anyone knows, yet I still may not produce computer programs that do what anyone wants. Indeed, consider the following quote by C.A.R. Hoare (164):

> Long ago, the welfare of a society used to depend heavily on the skill and dedication of its craftsmen - the miller, the blacksmith, the cobbler and the tailor. These craftsmen acquired their skill by a long and poorly paid apprenticeship to some master of their craft. They learned by imitation and experience, and by trial and error. They did not read books or study science, they knew nothing of the theory of their subject, the geometry of their rudimentary drawings, nor the mathematics underlying their primitive calculations. They could not explain how or why they used their methods; yet they worked effectively by themselves or in small teams to complete their tasks at a predetermined cost, to a fairly well predicted timetable, and usually to the satisfaction of their clients.
>
> The programmer of today shares many of these attributes of a craftsman. He learns his craft by apprenticeship in an existing team of programmers - but his apprenticeship is highly paid and usually very short. He develops his skill by trial; but mostly by error. He does not study theory, or even read books on Computer Science. He knows nothing of the logical and mathematical foundations of his profession; and he hates to explain or justify, or even to document what he has done. Yet he can often manage to complete his undertaken tasks, sometimes at the predicted time and within the predicted cost, and occasionally even to the satisfaction of his client.

Thus the problem is stated, how to produce software that does satisfy the client (and not necessarily the programmer). There are many aspects of this problem, since the software development process typically includes (at a minimum) requirements and/or specification, code generation, testing, and maintenance. This research presents analysis and techniques for improving the first formal step in this development process, the formal specification. Specifically, this research addresses the theoretical and practical aspects of generating a formal specification, and then transforming these formal specifications into other, more efficent (typically with respect to architecture or execution time), formulations.

The 'formal' in formal specification implies that there is a fixed set of syntactic rules governing the writing of the specification. These rules dictate which symbol strings constitute 'legal' specifications, and which ones do not. Thus a formal specification is written in a *formal language* (217), such as the language of regular expressions (Section 4.2). When combined with a collection of given 'truths', called the axioms, and logical rules of inference, a formal language becomes a *formal system* (317) or *theory* (see Section 3.3), within which new truths, called *theorems*, can be proved (derived). These theorems are derived by applying the rules of inference to the axioms and any previously derived theorems. For a given formal system there is an algorithmic technique for producing all of the theorems resulting from the formal system. Note that for a formal system that has the reasoning power of the second order predicate logic (217), there exist true statements written in the formal language of the system that cannot be proven as theorems, a property known as *incompleteness* (91).

This investigation utilizes a formal system described in the book by Chandy and Misra (64), and comprised of the formal language defined by the syntax of UNITY, along with rules of inference based on the first order temporal logic and the execution model for UNITY. This formal system is designed to reason about a given specification, so that certain truths about the specification can be proven. This formal system has as axioms certain given true statements about a specification, so that the set of axioms is not fixed, but is a function of the specification being analyzed. The rules of inference are fixed, those resulting from the UNITY execution model and the first order temporal logic (Appendix A supplies the additional temporal rules to the first order predicate logic (217)).

Because of the formal system used to generate the formal specification, there is a mathematical and logical basis for performing the type of analysis done within this research. This, combined with the formal specification's role as the first formal product in the software development process, makes the formal specification a logical choice as a starting point in addressing the concerns raised in the previous quote from Hoare.

This research presents the mathematical and logical framework to support this analysis of the formal specification process. This framework is based on *topology*, specifically the topology of complete metric spaces. The choice of metric spaces results from the requirement to address conceptually computations that do not necessarily halt. By using metric spaces, such computations can be treated as *convergent* processes, that is processes that can be defined in terms of their behavior in *infinite* time. Within metric spaces such convergent processes can be analyzed, and statements can be made regarding what type of behavior they may exhibit in an unbounded (in time) future. Accordingly, this effort is presented in two major parts. The first is the topological analysis of the computational models used in the analysis of the formal specifications, while the second addresses the actual generation and transformations of formal specifications.

1.1 Background

The software development process starts with a natural (English) language problem statement, which then evolves through a series of transformations from one form into another, until a form is reached that can be executed on a computer. Traditionally, these transformations have been performed either manually only, or else manually with some automated help. Research into completely automating the process continues (123), with varied approaches presented by different authors (24, 362, 107). Many of these approaches are designed to support specific models of computation (64), while others are based on transformations of either the control structures or the data structures involved in a given solution to the problem (23, 51, 98, 353). Many of the control structure transformation strategies are based on the classic paper by Burstall and Darlington, in which they presented a transformational system for converting recursively defined functions into other recursively defined functions (to improve efficiency), or into iteratively defined functions

User Requirements	versus	Specifications
User		Software engineer
Natural language		Formal language
Imprecise		Precise
Nontechnical		Technical
Application terminology		Software terminology

- G. W. Jones, *Software Engineering*, Wiley & Sons, 1990.

Figure 1.1. The Dichotomy Between User Requirements and Specifications

(57). Another early paper by Polychronopoulos presented a class of transformations based on nonrecursively defined algorithms and associated data structures based on directed graphs (290). This concept of equating transformations on formal products with transformations on directed graphs has persisted, with recent research addressing the preservation of program properties through directed graph analysis (35). Another classification of transformational approaches is between those based on functional programming styles and languages (280), and those based on logic programming styles and languages (347).

This investigation addresses this transformation process as applied to the formal specification, by presenting analysis and techniques designed to generate the formal specification from an informal specification/requirement or problem statement, and to transform the specification into other forms. The concept of applying formal tranformations to specifications is not new (237), but the specific techniques presented here are. The analysis is based on mathematics and logic, since, as stated by Sommerville, "A formal software specification is ... expressed in a language whose vocabulary, syntax and semantics are formally defined. The need for a formal semantic definition means that the specification languages cannot be based on natural languages but must be based on mathematics." (322) The relationship between the informal statement of the problem given in the requirement, versus the formalization of the formal specification, is summarized in Figure 1.1 (186)).

Unfortunately, there exists an ironic disparity between the increased emphasis on the front-end planning and design phases of the software development model, as well as the lack of wide spread acceptance of formal specification techniques within the commercial software industry (322). An example of this emphasis on the front-end planning is a generalized rule

used in allocating development effort among the three major phases of the development cycle called the 40-20-40 rule. This rule states that 40% of the total effort should be directed to planning, requirements analysis, specification and design, with 20% invested into the actual coding, and then the final 40% into the testing (293). But the increased use of formal specifications has been slowed by at least three major problems. The first is a lack of training, in that software engineers typically do not have the background and/or experience in the mathematics and logic required to formulate formal specifications (322). The second problem is that the applicability of formal specification techniques to complex tasks has not been widely accepted, thus there is the classic image of the 'good for toy problems in textbooks' (322). Lastly, much of the research has been into the theoretical aspects of formal specification, with a corresponding lack of commercially available tools to support the software engineer (322, 293, 186).

Consequently, instead of designing a new formal specification language, the decision was made to choose an existing one, and then to increase the utility of that language. Creating a new language would probably hinder, and not help, the lack of wide spread acceptance of formal specification languages, since it would just be one more language to learn. And by investing the effort into improving the utility of an existing language instead of creating a new one, this research hopes to lay the groundwork for automated tools to aid in the formal specification development process.

There are many choices of formal specification languages to support this research, and these choices include several different types of formulation. One such formulation is those languages whose syntax resembles that of an imperative programming language (134, 3), which includes the language UNITY (64) and the commercially available 'wide spectrum language' REFINE (349). Additionally, there are the formal languages whose syntax is graph based (21, 284) in the manner of Petri nets (279). Other formalized specification languages in the literature include those that are based on algebraic equations (336, 36). and those based on assertions written in a formal logic (214).

Although all of these formulations may appear inherently different, they are not. For example, as demonstrated in Appendix C, a specification written in an imperative programming language based specification language (UNITY), can be decomposed into

pieces that can be expressed in the first order predicate calculus over finite domains. Also, researchers have demonstrated that the graphical specifications based on Petri nets correspond to those based on algebraic equations (142, 259, 338). This research demonstrates the correspondence between UNITY and another formulation known as *communicating sequential processes* (CSP) that is more algebraic (165), while Bailor has shown the equivalence between CSP and Petri nets (21).

The formal specification language chosen for this effort is Chandy and Misra's UNITY (Unbounded Non-deterministic Iterative Transform) (64). The choice of UNITY results from the capability to write specifications in UNITY that are independent of architectural and implementation language considerations. This separation of program design (until a certain point) from hardware considerations represents a goal of some of the latest research into parallel programming (321, 33).

UNITY consists of both a formal syntax and a formal semantics, with the semantics defined in terms of the UNITY execution model. As part of the topological analysis of computational models this effort presents a mapping from UNITY programs into CSP processes that preserves the *behavior* of the UNITY programs. With respect to a UNITY program, the behavior is defined by the collection of all possible execution sequences for the statements comprising the program. However, because UNITY permits simultaneous execution of atomic actions (assignments), while CSP does not, this mapping does not preserve atomic actions. Consequently, this research also presents two additional execution models for 'programs' written in the UNITY syntax. These additional execution models serve two purposes. One purpose is to provide a mapping of UNITY programs (under these execution models) into CSP that can preserve atomic actions (although not necessarily). The second purpose is to extend the *conceptual model* of UNITY. The conceptual model of UNITY is that model the UNITY user will use as an analogy for the UNITY execution model. This relates to the claim by Iverson that our internalization of a solution to a requirement is influenced by the formal specification language employed (178). That is, the UNITY solution is influenced by the fact that it's written in UNITY, and the solution may be quite different from a solution written in another formal specification language by the same person. This inclusion of two additional execution models for UNITY attempts

to broaden the range of these conceptual models for UNITY by supplying execution models that more closely match actual hardware architectures than those of the original execution model.

Since UNITY was primarily developed to support parallel programming (64), this effort also primarily supports the specification of parallel programs using UNITY. This does not preclude the use of these results to the design of sequential programs, since a sequential program can be thought of as a parallel program executing on a single processor. As an interesting aside, even though parallel computation is generally considered a newer development than sequential (single processor) computing, it was considered by the early pioneers of computing. For example, the very first digital computer designers experimented with both parallel and serial designs before deciding on what has been called (incorrectly) the von Neumann architecture (318). Going back even further in time, perhaps the first documented reference to parallel computing is the following quote from a lecture given by Charles Babbage in 1842 (250):

> When a long series of identical computations is to be performed, ... the machine can ... give several results at the same time, which will greatly abridge the whole amount of the processes.

1.2 Research Outline and Document Overview

This research effort can be naturally broken into two major parts. The first is the topological and temporal analysis of the comp[illegible]al models: finite automata, CSP, and UNITY. The topological analysis, based on [illegible] metric spaces of models, is contained in Chapter IV, while the temporal analy[illegible] reasoning about the models using the *temporal logic*, is in Chapter V. The secon[illegible] major part is the improvement of the generation and transformation of formal specifi[illegible] using UNITY, which is contained in Chapter VI. The required computational mod[illegible]round, which includes an overview of computational models and program correctness, is presented in Section 1.3 of this chapter and Chapter II. The mathematical background is in Chapter III. Figure 1.2 shows the overall structure of the research effort as it relates to the document structure.

Section 1.3 of this chapter defines and presents examples of the two major classes of computational models used in the study of parallel programming and architectures, the *message passing* and the *shared variable* models. This material represents the justification for the selection of two of the three computational models used in this research effort, the *communicating sequential processors* (CSP) (165), and the execution model for UNITY (64). CSP is a message passing model, while UNITY is inherently a shared variable model, although UNITY programs can be designed to model other paradigms such as message passing.

Chapter II presents the definitions and basic concepts of *programs*, *processes*, *verification*, and *correctness*. This chapter gives a more abstract definition of a program than just symbol strings written in a formal language which can be executed. This abstract definition of a program is based on another abstract idea, that of a process. Although the

Correctness of Specifications and Programs - Chapter II
Mathematical Background - Chapter III
Analysis of Computational Models
    Background - Section 1.3
    Topological - Chapter IV
    Temporal - Chapter V
Generation and Transformation of Formal Specifications - Chapter VI

Figure 1.2. Research and Document Structure

terms 'verification' and 'correctness' are usually applied to specific products of the software development process, this chapter also presents more abstract versions of these concepts. The ideas presented in this chapter are used extensively throughout this effort.

Chapter III presents the mathematical background required for the remaining chapters. This background is broken up into three major areas. The first is the concept of a *class*, which is used as the basic building block of collections of things, instead of the more traditional set, and is introduced in Section 3.1. The second is how these classes relate to each other, that is *relation, function*, and *predicate*, which are defined in Section 3.2. The final section of the chapter, Section 3.3, presents introductory definitions and concepts from *category theory*, a theory that is based on directed graphs. Category theory is included not only because it is used in later chapters, but also because the theory of categories has become a versatile tool for computer scientists (27), not only from a theoretical viewpoint, but also from a practical one. As an example of the practical side of category theory, consider that researchers at Paris Univeristy claim to be developing a *categorical abstract machine*, "which executes categorical code on a very simple abstract machine" (85). Section 3.3 also defines the category of complete metric spaces, which forms the basis for the topological analysis of Chapter IV.

Chapter IV gives some additional introductory material on the three specific computational models used in this research. The three computational models used here represent a diverse sampling from the wide spectrum of all of the different models. The first is the finite automaton, which represents a classic computational model, the lowest on the hierarchy of abstract machines whose highest position is held by the Turing machine (217). Section 4.2 defines the finite automaton. The second model is one based on an algebra of processes, complete with operations on the processes, and formal representation of the processes. This model is CSP, and Section 4.3 presents the required material on formal process representation and process operators. The last model is the execution model for UNITY, which represents a computational model based on both a formal language and a formal semantics for the language similar to a compiled high level imperative language (64). The philosophy behind UNITY is that the execution model is abstract enough that it should not reflect any specific architecture. This places UNITY in the class of parallel

programming languages which could be described as *expression of problem parallelism*, that is those languages designed to express the parallelism inherent in the problem. This terminology is inspired by the *detection of parallelism* and *expression of machine parallelism* classifications of Perrott (275), which refer to languages that are sequential (the compiler parallelizes), and which are architecture dependent, respectively. Section 4.4 and Appendix C present some introductory material regarding the UNITY execution model.

The primary purpose of Chapter IV is to present the mathematical theory of the topological analysis of computational models. This topological analysis demonstrates that finite automata, CSP, and UNITY are all objects of the category of complete metric spaces. Section 4.1 introduces the metric space of all words of finite length that can be formed from a given alphabet, a metric space that forms the basis for the remainder of the chapter. Section 4.2 then develops a complete metric space based on the finite automaton computational model. Section 4.2 also demonstrates that the topological analysis of computational power can be directly related to the standard machine based analysis. A complete metric space based on the CSP model is developed next in Section 4.3, while Section 4.4 concludes the chapter with a complete metric space of UNITY programs. The construction of the UNITY metric space also shows that any UNITY program can be mapped into a CSP process that exhibits the exact same behavior.

The goal of Chapter V is to unify the temporal analysis of UNITY programs from Chandy and Misra (64) with that of finite automata and CSP processes. The unifying tool is the temporal logic of Appendix A. Section 5.1 gives an overview of the three major classifications of formal semantics for programs, the *axiomatic*, *operational*, and *denotational*. Section 5.2 then presents a semantic analysis of finite automata based on the temporal logic (which could be considered operational and axiomatic). The chapter concludes with Section 5.3 presenting the same type of temporal semantic analysis for CSP processes. This chapter also presents an overview of the temporal reasoning of programs by other researchers.

Chapter VI presents the second major part of this research effort, an improvement to the process of generating and transforming formal specifications written in UNITY. Section 6.2 introduces the *state space semantics* that are crucial to the theory used in

the last section of the chapter. Whereas the semantic reasoning about UNITY programs in Chandy and Misra's book is primarily based on the syntax of the programs, the state space semantics reasons about the *trajectories* generated by a UNITY program through a state-space based on the named variables of the program. Section 6.3 presents a search style methodology for generating formal specifications written in UNITY from informal formulations, and also transforming UNITY programs into other UNITY programs so as to preserve certain program properties. This methodology is based on a collection of rules designed to be used in a heuristic type search. Based on information collected during the application of these rules, this section also supplies an algorithmic technique for the optimization of UNITY programs in terms of execution time on multiprocessor architectures.

Appendix A presents definitions and introductory material on the *modal, temporal,* and *predicate* logic. The primary purpose of this appendix is to supply the needed temporal logic background. Since the temporal logic is a special case of the modal logic, the appendix defines the modal logic first, and then defines the specific temporal logic used to reason about computation within this research (see the book by Rescher (300) for an overview of temporal logic in general). Using temporal logic to reason about programs is not new (Apt (10) credits Pnueli (285) with first introducing temporal logic to reason about proofs of program correctness), but many of the specific uses within this research effort are new. Not only is temporal logic used for reasoning about programs and computations, but the symbology of the temporal logic is also used as a shorthand notation in many of the proofs. The appendix concludes with a presentation of the predicate logic as a special case of the temporal logic.

Appendix B collects miscellaneous results regarding computability theory, the concept of chaotic processes, and the implications of the axiom of choice, that are used through out this effort. This appendix also contains a historical summary of the definition of the word 'computable'.

Appendix C presents a result which can be used in the statistical testing of implementations of UNITY specifications. Also included is the result which justifies the claim that UNITY is basically another representation of executable first order predicate calculus.

Appendix D supplies additional insight into the relationship between different specification techniques by showing how the execution model for UNITY can be modeled using Petri nets.

1.3 Computational Models

This section defines the concept of computational models, and presents background material on different computational models and their associated hardware (machines) and software (languages). This presentation emphasizes the two primary models for parallel computing, the *message passing* and the *shared variable*, since these include the two models that this resarch is based on, CSP (165) and UNITY (64). This section also presents the motivation for choosing UNITY as the formal specification language for the effort described in Chapter VI. The section concludes with the *weak fairness* definition that forms the basis for the UNITY execution model.

"A conceptual model or problem that embodies the major features of a whole class of problems is called a paradigm." (248) The paradigms of interest here are those for parallel computation, which are called *computational models*. A (parallel) computational model is some type of formalism for representing how algorithms can be implemented on multiprocessors. The major differences between the different computational models is in how information gets from one processor to another, and whether or not the different processors execute dependently or independently of each other. If the information is passed through messages between the processors, then the model is classified as a *message passing* model (228, 11, 62, 48, 323). Examples of such message passing models are CSP (165), EPL (154), CCS (242), and CHOCS (332). The high level languages that are based on the message passing model include Occam (213) and Ada (78), while examples of actual computers whose architecture is based on this model are the BBN Butterfly, FPS T Series, Intel iPSC Series, AMETEK S14, NCUBE Series, and the Loral Dataflow LDF computers (15, 127). If the information is passed between processors through the use of a common memory storage (say in the form of a global variable), then the model is a *shared memory* (or *shared variable*) model (184, 185, 267). Two such shared variable models are UNITY (64) and ALPS (342), and Modula-2 (358) and Pascal Plus (276), are examples of two high

level languages based on the shared variable model. The Alliant FX/8, Cray X-MP, IBM 3090, Sequent Balance, ETA, ELXSI, Encore, and FLEX computers all have architectures based on the shared variable model (15, 127).

The class of models representing processors that execute in a dependent manner are the *synchronous* models, while those whose processors are represented as executing independently form the *asynchronous* models (39). Within the message passing models, the two major subdivisions are the synchronous and the unsynchronous. A synchronous message passing model such as CSP, Ada, or Occam (213) requires that "both the sender and the receiver of a communication must be ready to communicate before a communication can be sent", while the asynchronous message passing model model allows that "the receiver does not have to be ready to accept a communication when the sender sends it" (4). This difference is usually not important, since an asynchronous model such as Action Systems (16) c ctor (159) can actually model synchronized communications.

The shared variable models can also be divided into the synchronous and asynchronous classifications. A synchronized shared variable model can be formed from any shared variable model by simply introducing a global clock (118). This division of the shared variable models is somewhat superflous though, since general purpose shared variable models such as UNITY can model synchronous and asynchronous shared variable systems (64). In fact the distinction between shared variable and message passing models is somewhat artificial, since shared variable models can model both synchronous and asynchronous message passing systems (206), using semaphores as a locking mechanism (103).

The shared variable models have split into two other subdivisions besides the synchronous and unsynchronous. The first division includes the classic shared variable models, that is those models whose global data structures are accessed by name such as in UNITY, while the second division includes the *shared dataspace* models whose global data structures are content accessible, such as Linda (135), Concurrent Prolog (316), and Swarm (84).

Another major division of the computational models includes the graph based models

(259, 181, 158), such as Petri nets (278, 279), abstract Petri nets (142), transition systems (communicating automata) (261), abstract process networks (236), and dataflow graphs (14, 13). Depending on exactly what the atomic actions in the nodes and transitions of the graphs represent, these models can also be grouped under either the message passing or the shared variable paradigm. This effort does not specifically address these graph based models.

The primary reason for differentiating between shared variable and message passing is that these are arguably the two major parallel computation architectures, since they are the two major subdivisions of the mutiple instruction mutiple data (MIMD) class of architectures. The shared variable and message passing models are called multiprocessor and multicomputer systems respectively (22), and are also called shared memory and distributed memory architectures respectively (326, 320). The single instruction mutiple data (SIMD) class contains two major subdivisions known as the vector and systolic architectures (see Duncan (109) for an overview of MIMD and SIMD, and Kuck (197) for a description of how MIMD and SIMD fit into Kuck's taxonomy). Because these are the two primary parallel architectures, many of the current programming methodologies for concurrent computation are classified into either the message passing or the shared variable paradigms (60). Although this categorization of computational models ignores the many functional programming models (301, 68, 175, 20, 341), the inclusion of concurrency into these models requires either the shared variable approach or some type of message passing.

The shared variable and message passing models are not only considered the two major paradigms for parallel computation (29), but can also be used for sequential computations, since they naturally match up with parameter passing by value (message passing), or by reference (shared variable). Also note that although the primary motivation behind these models is computational (computer programming), their applicability is not so restricted. Consider that the Actor model has been used to describe the momenta of multiple particle systems within the framework of quantum mechanics (227).

In choosing the primary computational model for this effort, UNITY, the following guidelines proposed by Pnueli and Harel were used: (288):

1. The model should adequately represent the computation's behaviour on the actual hardware architecture.
2. The model should supply needed formalism for reasoning about the properties (usually stated as propositions) of the computation.

UNITY, as a shared variable model, satisfies the first item when the target architecture is a multiprocessor shared variable machine. Chandy and Misra demonstrate in their UNITY text that UNITY also contains enough flexibility so as to satisfy the first item for the multiprocessor memory passing architecture (64). UNITY does satisfies the second item, as demonstrated by the proofs of program properties in the book by Chandy and Misra (64). UNITY also satisfies a major criterion for computational models which Wirth calls the "intellectual manageability of programs" based on the single entry, single exit property (357). This requires that the basic statement control structure should consist of program units that must execute all of their constituent pieces in the same manner for every possible execution. The assignment statements of UNITY satsify this property, since on every execution of a statement, every assignment component executes in exactly the same manner, simultaneously (although Wirth was actually addressing sequential execution of the units).

Another consideration in choosing a computational model is whether the model assumes simultaneous execution of atomic actions or not. UNITY forces the simultaneous execution of all assignment components within a single assignment statement. Other models, which are typically called *interleaving* models (288, 196), prevent this simultaneous atomic action execution. The *standard* UNITY execution model, another modification to the UNITY execution model and defined in Section 4.4, prohibits the simultaneous execution of assignment components within a single assignment statement, thus extending the modeling capabilities of the UNITY syntax.

Another major factor in the choice of UNITY is that the individual assignment components are equivalent to statements from the first order predicate logic over finite domains (see Appendix C). This satisfies a requirement for computational models attributed to Hoare that "programs should be equated with the predicates describing their observable

behaviour." (262) UNITY programs represent the execution of assignments that satisfy first order predicate calculus statements.

Usually the choice of which processor performs an atomic action at a given time is nondeterministic, so that some rules must be established such that a process is not 'neglected' for any substantial interval of time. Such neglect does not represent 'real world' concurrent computations, and can interfere with the formal analysis of the computations. These considerations have led to the concept of *fairness* requirements, so that all processors actually execute at least one atomic action within some given time interval. Two types of fairness are defined and used here, *weak* and *strong*. These concepts of fairness embody the requirement that if a parallel program is considered as a collection of concurrently executing processes, then no one process that could be running should sit idle indefinitely while the others are running. (See Appendix A for a reference on the symbology of Definition I.1 and the meaning of 'temporal predicates')

Definition I.1 *Given the two predicates from the temporal logic*

$$A \Longrightarrow a$$

$$B \Longrightarrow b$$

then the requirement for weak fairness *is given by the temporal predicate*

$$(\Box A \Longrightarrow \Box\Diamond a) \wedge (\Box B \Longrightarrow \Box\Diamond b)$$

that is, over any unbounded time interval during which both A and B are true, then a will become true an unbounded number of times, and b will become true an unbounded number of times. The requirement for strong fairness *is given by the temporal predicate*

$$(\Box\Diamond A \Longrightarrow \Box\Diamond a) \wedge (\Box\Diamond B \Longrightarrow \Box\Diamond b)$$

that is, over any unbounded time interval during which A becomes true an unbounded number of times and B becomes true an unbounded number of times, then a will become true an unbounded number of times and b will become true an unbounded number of times.

Definition I.1 can be extended to any finite number of implications of the form

$$A \Longrightarrow a$$

by simply forming all such possible combinations of two implications at a time.

Definition I.1 is an abstraction of the standard definition (131), and can be interpreted in the following manner. If A represents a test, or *guard*, and a being *true* represents the execution of some action or event, then

$$A \Longrightarrow a$$

denotes that if the guard A is satisfied at some instant, then the action corresponding to a occurs at that instant. For example, A could represent the statement that a certain process is enabled, and the truth of a that the process is actually executing, likewise for B and b. Then the assertion for weak fairness states that if two processes are continually enabled (with respect to the implied present time), then they both execute an unbounded number of times, since

$$\Box\Diamond a$$

states that the corresponding process will execute again regardless of how often it has executed already. Although some authors state that the process will execute an infinite number of times, instead of an unbounded number, there are no physically realisable processes which have executed or ever will execute an infinite number of times (as long as they require finite time intervals).

In a similar manner, the requirement for strong fairness states that if two processes are both enabled an unbounded number of times, then they will both execute an unbounded number of times. Or, more intuitivley, if both processes are eventually enabled with respect to any time which is considered the present, then they will both eventually execute. Note that UNITY uses weak fairness, since there are no guarded commands and thus all statements are continuously enabled (64). Chandy and Misra also make the interesting observation that 'unfair selection', which allows a random choice without having to satisfy

either weak or strong fairness, can be modeled with weak fairness.

1.4 Summary

This research effort addresses two major objectives. The first is a unified mathematical framework within which to analyze the many disparate models of sequential and parallel computation. This unifying effort is inspired by the multiple approaches within the literature to analyzing these models, based on the mathematical tools of lattices, domains (a special type of lattice), algebras, categories, and others. This unification is accomplished in this research utilizing the mathematical tools of the topology of complete metric spaces. Three different computational models are presented as complete metric spaces: finite automata, representing the basic state transition machine model for computation; communicating sequential processes (CSP), a message passing model designed primarily for modeling parallel computations; and UNITY, a shared variable model designed to model many different types of computations, both sequential and parallel.

The second major objective of this research effort is the design of a methodical technique for developing formal specifications written in UNITY. The purpose of this development technique is to aid the software designer in both the writing of the original formal specification, and also the rewriting of the formal specification so that each revision retains the desired properties of the original version. UNITY is chosen for this effort for two primary reasons: first, it is a specification language designed around an execution model that supports both sequential and parallel computations; and second, there exists a well documented approach to proving certain properties of specifications written in UNITY. Since these proofs of properties comprise what is known as *program correctness*, the next chapter introduces the concept of correctness, and the relationship between correctness and formal specifications.

II. Correctness and Specification

This chapter presents background material on *programs* and *processes*, and also on the *specification, verification*, and *correctness* of these programs and processes. The background material is used throughout this research effort, since the analysis of computational models and formal specificatons is based on reasoning about programs and processes. The concept of correctness and verification is crucial to this research, since the transformations developed in Chapter VI are designed to preserve correctness, thus implying that these transformations are self verifying.

Section 2.1 gives the definitions for *states*, *events*, processes, and programs, along with some basic results that follow from these definitions. The section also defines one program (process) *simulating* or *emulating* another, plus two programs (processes) being *equivalent.* Section 2.2 presents definitions for the specification, correctness, and verification of programs and processes. The section also presents the interrelationship between specification and correctness (verification). Section 2.3 addresses the concerns that motivate this research, the connection between the software development process, proofs of program correctness, and the design of formal specifications. The section also presents the basic motivations for the generation and transformation of formal specifications as analyzed in Chapter VI.

2.1 Programs and Processes

This section introduces and defines the abstract concepts of a *state* of a computation, an *event* that causes transitions between states, and a *process* and a *program* as a collection of sequences of events, such that each sequence of events can generate a sequence of states.

Definition II.1 *A* **state** *is an n-tuple whose components are called* **variables.**

This is the basic traditional definition of a state as an instantiation of all of the program variables at any given instant of time during the execution of the program (96).

Definition II.2 *An* **event** *is a function that maps states into states.*

Since an event is a function, then given a unique state, the function evaluated at that state yields just one state. So the event as a function correponds to a computational event in a program, say an assignment. The input state to the function is the state immediately prior to the assignment, and the output state resulting from the evaluation of the event function is the state immediately after the assignment (263). If the states are put into a one-to-one correspondence with intervals of time, then Definition II.2 implies that events can be put into a one-to-one correspondence with instants of time. This results from considering that a state corresponds to the interval of time for which it exists, and at the instant of time that an event occurs, it causes a new (although the new state could be equal to the old) state to exist for its corresponding interval of time. This coincides with the concept from the *distributed real time logic* (DRTL) that an event is a *temporal marker* (214).

Definition II.3 *A* process *is a countable set of sequences of events.*

This definition along with the previous one for an event also agree with the ones given by Hoare, who states that a process consists of units of behavior called events, where "events are regarded as instantaneous" (48). This definition also parallels the one from the *real time logic* (which forms the basis for the DRTL) that a process is an ordered set of actions (180). In the analogy with an executable program, the process corresponds to any sequence of executable statements that change the values of the variable (states). Thus a process could be a subroutine, or even multiple subroutines, since a process is a *set* of sequences of events. The basic idea is that given any one sequence, and an initial state, then the execution of the sequence corresponds to the sequential functional composition of the events in the sequence, yielding one final state.

A process does not have to be deterministic. For example, the process of rolling one fair die generates a set with a sequence of events, each event being one roll of the die. For each event (roll) there is the input state, the value of the previous roll, and an output state, the value of the next roll. If the die is truly random, then there is no algorithmic process that can generate the same sequence of states. Such a random process is based on the concept of a random choice, which is similar to another nonalgorithmic process that is expressed in the Axiom of Choice (see Section 3.1). Thus the following theorem:

Theorem II.4 *There exist processes for which the set of sequences of events cannot be generated by any Turing machine.*

Proof: See Theorem B.1 in Appendix B. ■

Although Definition II.3 does not so constrain a process, often the concept of a process is that of a sequence of statements executing on a single processor, while a *program* is a collection of such processes executing concurrently on multiple processors. This next definition of a program embodies this concept, while still allowing for a program and a process to be interchangeable.

Definition II.5 *A* program *is a set formed from a countable union of processes.*

This next definition supplies the link between the events that constitute a process or a program, and the sequence(s) of states *generated* by that process or program.

Definition II.6 *The program (or process) P* generates the sequence of states *S,*

$$S = \{s_0, s_1, ...\}$$

if and only if P contains a sequence of events E,

$$E = \{e_0, e_1, ...\}$$

such that

$$i \in N \iff e_i(s_i) = s_{i+1}$$

where

$$N = \{0, 1, 2, ...\}.$$

If E and S are finite, then

$$N = \{0, 1, 2, ..., n\}$$

and both E and S contain n terms.

A program or process can generate the sequence of states S, if it contains the sequence of events E as given in this definition such that

$$\begin{aligned} e_0(s_0) &= s_1 \\ e_1(s_1) &= s_2 \\ &\vdots \end{aligned}$$

The state s_0 is usually called the *initial state*, and if the sequence of states has only n terms, then s_n is called the *final state.*

Even processes that have a finite algorithmic definition can be intractable in analysis, as demonstrated by this next theorem.

Theorem II.7 *There exists a process P that generates only one sequence of states S, where S is Turing enumerable, such that there does not exist any Turing machine whose output is S given any finite subsequence of S as input.*

Proof: Consider the process P defined such that the sequence S is not finite

$$S = \{s_1, s_2, ...\}$$

where s_1 is the first output from the Turing machine that generates S from P, s_2 is the second output, and so on, and there are k (k is a natural number) possible distinct symbols for each s_i (a consequent of the Turing enumerability of S). These machine outputs will be considered as the 'outputs' of the process. Now, for any finite subset T of S, such that T contains the first n elements of S

$$T = \{s_1, ..., s_n\}$$

there does not exist any Turing machine that can produce S from T, since there exists at least $k - 1$ other processes distinct from P (and distinct from each other) that can also generate T (as their first n 'outputs') but whose next (i.e. $n + 1$) 'output' is different from s_n. And since a Turing machine can only read a finite number of

tape cells in finite time, the set S cannot be input in its entirety to the machine, only finite subsets such as T. ■

What this theorem states is that there exist finitely describable algorithms that define processes, such that these algorithms cannot be deduced given any finite behavior of the process. Although this may seem obvious enough, there was a time when the SAT test writers where unaware of this and wrote questions that asked for the next number in a given finite sequence of numbers.

This next definition is based on the definition of *emulation* and *simulation* given by Miller and Kasai (238).

Definition II.8 *Given the programs (processes) S and T, then T* simulates *S, and S* emulates *T, with respect to the function h, if and only if, for any sequence of states*

$$\hat{S} = (s_0, s_1, ...)$$

such that S generates $\hat{S}$ as given in Definition II.6, there exists a sequence of states

$$\hat{T} = (t_0, t_1, ...)$$

where T generates $\hat{T}$ as given in Definition II.6, such that

$$i \in N \iff h(s_i) = t_i$$

where

$$N = \{0, 1, 2, ...\}.$$

If $\hat{S}$ and $\hat{T}$ are finite, then

$$N = \{0, 1, 2, ..., n\}$$

and both $\hat{S}$ and $\hat{T}$ contain n terms.

If the function h is to be Turing computable, then h must be fully defined with only a finite representation. That h must be finitely defined is an important consideration

whenever the two sequences are not bounded in length. This definition is presented here because of the relationship between simulation, emulation, and *constructive verification*, which is addressed in Chapter VI.

If the statement is made that program T simulates program S without any reference to a function h, then h is implicitly the identity function. In this case, any sequence of states generated by S is also a sequence of states generated by T. But it is possible that there are other sequences generated by T which are not generated by S. If however, any sequence generated by one program is also generated by the other, then the two progams are *equivalent.*

Definition II.9 *Two given programs (processes) S and T are said to be* equivalent *with respect to the function h, if and only if, T simulates S and S simulates T, with respect to the function h.*

As with the concept of simulation, if the function h is not explicitly given in a statement of equivalence, it is assumed to be the identity function. Thus the claim that S and T are equivalent means that they cannot be differentiated with respect to the state descriptions.

2.2 Specification, Correctness, and Verification

This section primarily addresses the problems associated with the *specification, verification*, and *correctness* of parallel programs, since these are the concerns which motivated this research. Since this research effort is directed towards the development of specifications, and since verification is not tied to any one step of the software development process, the verification of most interest here is the verification of one form of the specification with respect to another. These different forms result when the original specification is transformed into other specifications, usually to address the questions of implementation. Each successive transformation generates a new specification that preserves the properties (expressed as predicates whose individual variables are the named variables of the specification) of the previous one, but represents more closely the design of the implemented software. Typically these transformations result from the need for mappings of processes onto processors, refinement of the algorithms, or efficiency improvements.

The basic idea behind the specification is that whether formal or not, it represents what *behavior* the program should exhibit. The behavior is defined as the different possible sequences of values for the named variables. The specification is usually not considered implementable, whereas a *prototype* is. As used in this research, specification applies to the development of programs, not to the development of executable languages (such as through an algebraic specification (36)). The following definition of specification from Hoare (165) summarizes the intended meaning as used here, although specifications that are not expressed as predicates are also considered:

> A specification of a [program] is a description of the way it is intended to behave. This description is a predicate containing free variables, each of which stands for some observable aspect of the behaviour of the [program].

Hennessy adds the constraint that the specification does not address how the behaviour is obtained, only what behaviour is demonstrated (154). Since UNITY can be considered as a formal representation of the first order predicate calculus over finite domains (64), UNITY meets the general requirement for a specification language. The UNITY syntax combined with an execution model thus forms a formal specification language.

Another form of a specification is a relation consisting of a set of 2-tuples, such that each tuple consists of an input state and an output state (240, 363). The actual definition of what a 'state' is doesn't matter to this concept of a specification, but this research defines a state (see Definition II.1) as a tuple (vector) representation of the current values for all program variables. If the specification is a relation, such that there are multiple tuples with identical input states, then the specification permits a nondeterministic choice of possible outputs for a given input. If the specification is a relation such that each tuple has unique input states, which makes the relation a function, then the specification requires a specified unique output for any given input.

The concept of *verification* can be defined in terms of the *waterfall* model of the software life cycle (308, 117, 160), a model which is still in widespread use within the Department of Defense (281), and which is considered to include the conventional software development process (362, 24). Figure 2.1 (from the book by Sommerville (322)) sum-

1. *Requirements analysis and definition* The system's services, constraints and goals are established by consultation with system users. Once these have been agreed (to), they must be defined in a manner which is understandable by both users and development staff.
2. *System and software design* Using the requirements definition as a base, the requirements are partitioned to either hardware or software systems. This process is termed systems design. Software design is the process of representing the functions of each software system in a manner which may readily be transformed to one or more computer programs.
3. *Implementation and unit testing* During this stage, the software design is realized as a set or programs of program units which are written in some executable programming language. Unit testing involves verifying that each unit meets its specification.
4. *System testing* The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the software system is delivered to the customer.
5. *Operation and maintenance* Normally (although not necessarily) this is the longest life-cycle phase. The system is installed and put into practical use. Maintenance involves correcting errors which were not discovered in earlier stages of the life-cycle, improving the implementation of the system units and enhancing the system's services as new requirements are perceived.

- I. Sommerville, *Software Engineering*, Addison-Wesley, 1989.

Figure 2.1. Waterfall Model of Software Development

marizes the waterfall method's five stages of software development. The basic idea with respect to this model is that verification involves a proof that one step in the software development model produces a product that is somehow 'equivalent' to the product produced in the immediately preceeding step (18, 209, 144, 29). Depending upon the actual details of the model, these products can be widely varying, as can the proof techniques actually used (40, 147, 360, 222, 268, 220, 75, 118) and (74, 130, 97, 281, 169). But the common concept for all of these approaches is that with respect to some definition of 'equivalent', the software development process should produce a series of 'equivalent products' (the definition of equivalent can change from step to step). Note that the term 'proof' is to imply that the technique of 'debugging' by repeated testing should not be considered verification (234).

The concept of *correctness* is similar to verification (often the two terms are used interchangeably), in that both require a proof that one product of the software development

phase is equivalent (in some manner) to a preceding product (218, 93, 183, 9, 151). But whereas verification proofs are based on the *syntax* of the products, correctness proofs reason about the *semantics* of the products (201, 267). This means that a verification proof presents arguments about how certain symbol strings from one product are equivalent to certain symbol strings from another product. But a correctness proof gives arguments to show how the symbol string from the one product can be mapped into some formal system, and how the symbol string from the other product can be mapped into some formal system, such that the two formal system representations are equivalent. Often the two formal systems are the same (in which case the correctness proof can be used to substantiate the verification proof). This definition of correctness follows from the term 'semantics' meaning that symbol strings have been mapped into another formulation which is considered to represent the 'meaning' (to humans) of the strings (141). With respect to the formal systems used in the semantics of the proof of correctness, showing the equivalency of two symbol strings within these formal systems is just another syntatic analysis, so that a more precise definition of correctness would be that its verification with respect to the formal systems that represent the semantics of the original symbol strings.

Consider the following example based on figure 2.2 that demonstrates the difference between verification and correctness. The two products from the development cycle are a formal specification written in UNITY, and the C program that supposedly implements the specification (some authors call a formal specification a 'prototype' (309), whereas a 'specification' is one or more assertions from the temporal predicate calculus of Appendix A). If the development proceeds directly from UNITY to C, as shown by the transformation T in Figure 2.2, then verification involves a proof that the mapping T generates a C program that satisfies the same predicates (that represent the specification requirements) as the UNITY specification (some authors consider the verification of a formal specification or prototype with respect to the informal specification as validation (24)). Correctness of the C program with respect to the UNITY specification requires two additional mappings, denoted by M and N in Figure 2.2. Denoting the codomains of M and N as 'mathematical models' follows the definition of correctness given by Lamport 201). Given that the UNITY specification is correct, then the C program is correct if the result of applying M to

the UNITY specification is equivalent to the result of applying N to the C program, where the equivalence definition depends upon exactly what M and N are. Observe that the mappings M and N could be the identity mappings, which would mean that correctness and verification would be the same. Also, some authors consider verification and correctness to be identical when the software development process only contains certain formal products (formal in the sense of being written and defined with respect to some formal system). For example, Howden states that "When the only development products are specifications and programs, then ... this is consistent with the use of the word verification to refer to proofs of correctness" (171).

Often the two mathematical models are identical. For example, both models could be the formal system based on *recursive functions*. This particular choice of recursive functions is appealing since McCarthy has shown that any formal product (that expresses some algorithmic relationship between 'inputs' and 'outputs') from the software development process can be represented as a recursive function, such that the function and the product have identical input/output behavior (230). Boyer and Moore use the set of all recursive functions that can be expressed in Lisp as the common mathematical model for correctness proofs using their mechanical theorem prover (15). Another mathematical model is the first order predicate logic, which can be used for proving correctness using the resolution method (66). The two products to be compared are mapped into sets of assertions from the first order predicate logic, so that the proof of correctness must show that the two sets of assertions are equivalent (partially decidable).

With respect to the earliest formal techniques for program verification, the term 'verification' was defined in terms of 'partial' or 'total' correctness (217, 18). Partial correctness meant that the program satisfied the predicates that constituted the specification, while total correctness added the constraint that the program terminated, concepts whose formalism have been credited to Floyd (126) and Hoare (161) (although Burstall (53) has credited McCarthy (232) with the first formalism of program proofs). However, recent trends in correctness proofs and definitions have indicated that there are two major types of proof techniques regarding what used to be called verification. The definition of verification given here is one of these techniques, which agrees with other recent definitions of

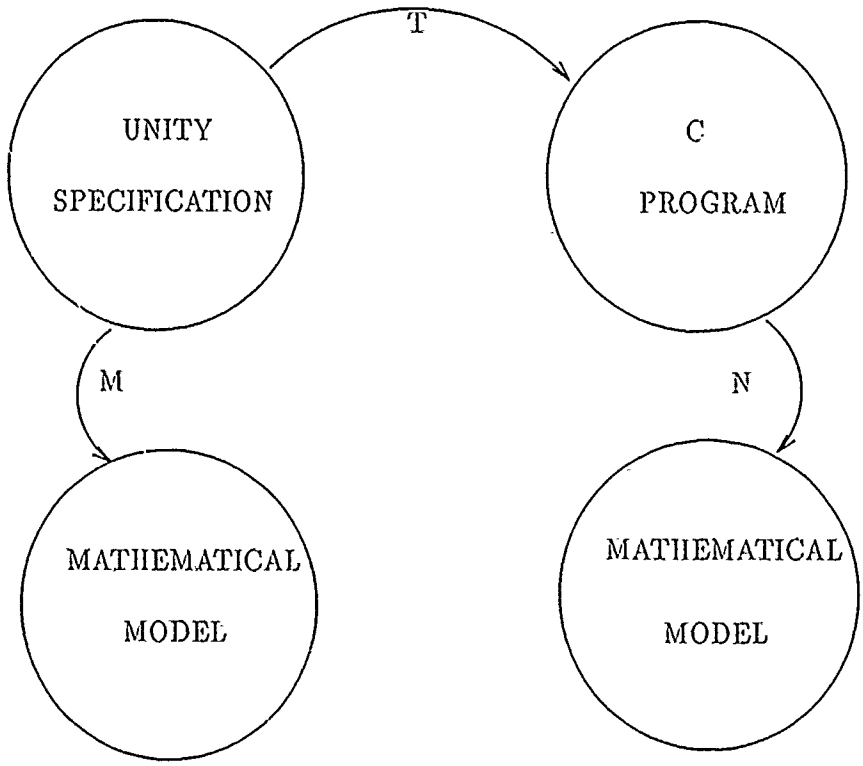

Figure 2.2. Contrasting Verification and Correctness

verification (although these authors may not have differentiated between verification and correctness) (40, 122, 362). The other technique closely follows the definition of correctness given here, a definition which also agrees with recent definitions and usages of the term 'correct' (209, 45, 201).

The philosophy that motivates Chapter VI is not to verify or prove a program correct once it's finished, but instead to develop the program methodically so that the final product is verified by construction, which has been termed *constructive verification* (361). This philosophy results in proofs about the resulting product that can fall under both the verification and the correctness concepts as presented in the following definition:

Definition II.10 *Given the function T, whose domain is a class A of formal strings of symbols representing programs, and whose codomain is another (possibly the same) class B of formal strings of symbols representing programs, functions M and N whose domains are A and B respectively, and whose codomains are formal systems (either logical or mathematical), and $a \in A$, then $T(a)$ is* verified *if and only if*

$$a = T(a)$$

with respect to an equality predicate defined between A and B. Further, $T(a)$ is correct *if and only if*

$$M(a) = N(T(a))$$

with respect to an equality predicate defined between the codomains of M and N. Given the elements $a \in A$ and $b \in B$, then $M(a)$ and $N(b)$ are called the semantics *of a and b respectively.*

Note that Definition II.10 does not specify what the codomains of M and N are, but they must be specified when defining the functions M and N. The equality relation between the codomains of M and N need only be well defined (i.e. Turing computable), and need not necessarily satisfy any other requirements. Chapter VI addresses in more detail a specific example of what types of functions M, N, and T might be, and Section 4.4 gives another example that relates more specifically to Figure 2.2. The codomains of M

and N correspond to what some authors call a 'semantic domain' (354), a concept which may have originated in a paper by Marvin Minsky (245), in which he stated that "once the ... problem is given a (semantic) interpretation, we can bring to bear heuristic methods acquired in a more familiar domain".

Often the differences between correctness and verification, and between specification and prototype, become blurred. For example, in a book about logic programming and declarative languages (which includes the logic programming languages such as Prolog), Hogger states (and credits the idea to Darlington and Kowalski (88)) that declarative languages have the "dual functions of both specification and computation" (167), This implies that code written in such languages serves as specification statement, implementable prototype, and final solution. Additionally, the following quote from Schnupp and Bernhard emphasizes the fuzziness between these concepts: (309)

> the Prolog versions serve ... as formal specifications. Whereas modern software engineering discourages the program developer from writing his implementation 'directly into the computer' until he has carefully prepared a detailed specification thereof, using Prolog one can hardly avoid such procedure. The implementation is the specification! The fact that his specification is immediately executable and testable certainly cannot be held against him!

This research does not attempt to address the issue of validation, which is related to verification but is typically defined in terms of the actual software development model used (24, 340). Validation is an attempt to demonstrate that the statement of the specification or the executable program properly reflects the original user requirements (79), or to "compare a software development product with the user's perceived requirements for that product" (171).

To demonstrate the relationships between specification, verification, and correctness, consider two separate predicate specifications, denoted by $R1$ and $R2$, which are graphically depicted in Figure 2.3. With respect to $R1$ and $R2$, A and C represent the inputs, while B and D represent the outputs, respectively. Although this figure is just a symbolic representation of the specifications and their input and output data and/or events, the analogy with circuitry implied by Figure 2.3 is not unwarranted (214). For each specifica-

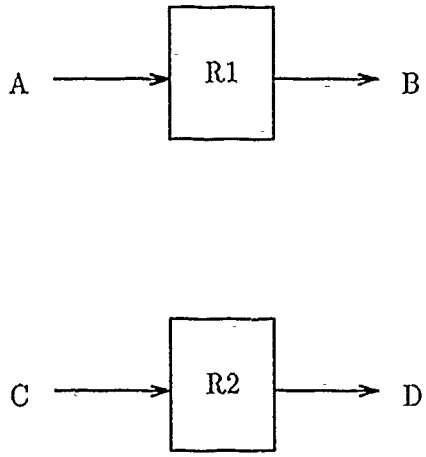

Figure 2.3. Graphical Depiction of Two Specifications

tion the inputs and outputs constitute the 'observable aspect(s) of the behaviour' refered to in the previous quote from Hoare.

Since Hoare's definition states that a specification is a predicate, then let

$$R1(A, B)$$

denote a predicate that evaluates to *true* for certain 'values' of A and B in such a manner as to satisfy the intent of the specification. Likewise for $R2(C, D)$. As an example of such an $R1$, consider a specification that states that for any input that satisfies an input predicate denoted by P, there only exists a *unique* instantiation of the outputs that satisfies the output behaviour predicate denoted by Q. With respect to Figure 2.3, this representation is given symbolically by

$$\exists A[P(A)] \Longrightarrow \exists! B[Q(B)] \tag{2.1}$$

This assertion actually comprises the specification given by $R1$ in the figure, so that Equation 2.1 is $R1(A, B)$ (179). Thus Equation 2.1 represents a specific $R1$, whose input and output predicates P and Q must be derived somehow from the specification statement. Since these specification predicates are based solely on the desired behaviour of the program, they evaluate to *true* or *false* following the rules of the predicate calculus, that is

their evaluation is not dependent upon any outside factors (such as the actual implementation, see the next paragraph).

Now consider another predicate

$$D1(A, B)$$

that syntactically is equivalent to $R1$ but denotes a predicate that represents the actual design that was based upon the specification $R1$. So that whereas the predicate $R1$ can be evaluated conceptually based on the specification, $D1$ evaluates to *true* or *false* depending upon the actual implementation of the specification. This means that one requirement for verification could be that

$$R1(A, B) \Longrightarrow D1(A, B)$$

In a similar manner if the design denoted by $D2(C, D)$ is based on the requirement given by $R2(C, D)$, then a verification requirement can be given by

$$R2(C, D) \Longrightarrow D2(C, D)$$

This verification requirement states that whenever the specification conceptually satisfies a given predicate, then the actual implementation derived from the specification must also satisfy the given predicate. In general, these predicates only contain *individual* variables, so they are expressed in terms of the first order predicate calculus (217). However, this requirement allows for the possibility that the design could satisfy a predicate that the specification does not, since

$$false \Longrightarrow true$$

evaluates to *true*. Thus this verification requirement permits the possibility that the design could exhibit behavior that was not explicitly addressed in the specification.

Another statement for verification could be that

$$R1(A, B) \Longleftrightarrow D1(A, B)$$

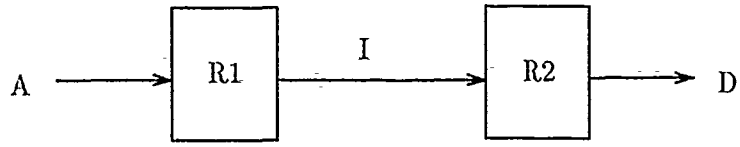

Figure 2.4. Graphical Depiction of Composite Specification

or

$$R2(C,D) \Longleftrightarrow D2(C,D)$$

which states that with respect to certain assertions, whatever the specification satisfies, then so does the design, and vice versa. Thus with respect to these assertions, the design and the specification are *behaviorally equivalent.* If the parameters A and B (C and D respectively) represent the observable actions of the specification and the implementation, then with respect to the assertions contained within $R1$ ($R2$), this definition of behavioral equivalence also satisfies the intent behind Hennessy's and Milner's observational equivalence (153). That is, with respect to these observable actions, the specification and design that are behaviorally equivalent cannot be distinguished from each other based only on the 'observational behaviour' defined by $R1$ ($R2$).

If the two specifications were combined into one specification, such that the 'things' represented by B were somehow (say in type) compatible with those of C, then the specification depicted by Figure 2.4 would only have A and D as observable behaviour. Additionally, the predicate $R(A,D)$ that would represent this 'composite' specification could be stated in terms of the individual predicates for the separate specifications by

$$R(A,D) = \exists I[R1(A,I) \wedge R2(I,D)]$$

which shows how the 'hidden' I is handled.

Zelkowitz has classified verification into three types, *algebraic*, *axiomatic*, and *functional* (363). Algebraic verification is based on the theory of signature algebras and type abstraction (353, 19, 298, 330, 199, 269, 146), and can be cast in terms of Definition II.10 with functions M and N whose codomains are some form of a *universal algebra* (143), often

based on algebraic specifications (36, 337, 336, 44) and/or abstract data types (331). The basic idea behind algebraic specification and/or verification is that the desired task and the program that implements that task can be represented as algebraic equations. This type of verification is not addressed any further in this effort.

Axiomatic verification includes those techniques that merge the symbol strings that form the program (from Definition II.10) text with statements from the predicate calculus, characterized by specifications that are predicates. This group includes the technique (268, 18, 145) that has emerged from the addition of *predicate transformers* (105) to the Floyd-Hoare logic (126, 161). The Floyd-Hoare logic was the basis for some of the earliest rigorous verification techniques (292, 217, 209). Other verification techniques, such as those based on lattices (312) or complete partial orders (241) may also fall into this group, depending upon the interpretation of Zelkowitz's definition. With respect to the techniques based upon the predicate transformers (and considering the domains of the functions to include symbol strings that are called 'programs'), the codomains of the functions M and N can include statements of the form

$$\{P\}S\{Q\}$$

where P and Q are predicates, and S is a program statement (or an entire program). Thi assertion states that if P is true when statement S starts execution, then after (if it does) S terminates the predicate Q will become true. Thus reasoning about the 'correctness' of the program is based on the relationships between sets of predicates and the program statements. Starting with predicates that are true at the beginning of the program's execution, and working through the statements of the program using assertions of the form above, the goal is to finally derive the desired predicate(s) as being true after the final statement executes. The key idea is th t the reasoning about the program uses rules of inference, or a system of logic, that is based on the syntax of the statements as they appear in the program. In a more abstract sense then, this verification technique is a logic system (or theory, see Section 3.3) whose underlying mathematical model uses the syntax of the program statements. This contrasts with the algebraic verification techniques whose

underlying mathematical models use equations that do not directly use the syntax of the program statements.

The third group of verification techniques is the functional. Their primary characteristic is that the specifications are given in a relational or functional form, as sets of 2-tuples of input/output states. This yields functions M and N from Definition II.10 whose codomains consist of assertions about the effects upon the 'state' (see Definition II.1) that result from the statements that comprise the programs in the domains of M and N. One such technique has been developed by Mills and others (240). Since a logic programming language such as Prolog can be considered as a relational language (as opposed to a functional language such as Lisp), then certain verification techniques for logic programming also fall into this category (67).

A completely different approach to informal verification is based on probabilistic methods. This approach uses testing to derive statistical estimates that the implementation contains no errors (340). Since testing does not prove there are no errors, this must be considered informal verification. And since the testing is usually designed to test for errors with respect to the original user's intent, this technique can also be considered validation.

An interesting problem that faces probabilistic statements about software verification results from the different perspectives of the producer and the consumer. For a given software package, the producer will increase the probability that the software is error free with each new error that is found, whereas the consumer does just the opposite, losing confidence in the software with each new fault (340). Ironically, the historical data doesn't necessarily support either belief. Consider that the January 1990 breakdown in AT&T's telephone network, which resulted in more than $60 million in lost income, resulted from a "single error in one line ... of a ... scrupulously tested" program (277).

2.3 Software Design and Formal Specifications

This research addresses the problem of writing a formal specification which will eventually be used to generate the executable code. Since there are not currently any UNITY compilers, the use of UNITY in writing specifications requires a manual effort

to transform this specification into the executable program. A technique called *stepwise refinement* is based on transformations that successively refine the original UNITY program into a form that more closely matches how the executable program is to be written. There are other possibilities (21), which are classified into the following three schemes, based on the work by Bouma and Walters (44):

Direct Execution In this approach the specification is executed (compiled) directly (as written). A major advantage is that verification of the compiler implies that the behaviour of the executed code matches that of the specification. Disadvantages include the possible inefficiency of the execution and the lack of such compilers.

Translation to Logic Programming Language This tact is a two step process, the first requires the conversion (either automated or manual) into a logic programming language such as Prolog, and the second is the execution of the logic program. Advantages include the documentation of techniques to convert from a specification into Prolog (44), the possibility of modifying the logic program directly to test changes to the specification, plus the availability of Prolog interpreters/compiliers. Disadvantages include the existence of specifications which cannot be converted into Prolog, and the lack of modularity for Prolog when compared to standard programming languages (see next item).

Translation to Standard Programming Language Instead of a logic programming language such as Prolog, the target language for the specification is a general purpose language such as Ada or C. Compared to the conversion to a logic program, this conversion can offer the advantage of increased efficiency for the executable program plus greater similarity between the modularity of the specification and the modularity of the target code. The comparative disadvantages include the increased complexity and corresponding problems with proof of correctness for the translation process; a consequent of the lack of documented formal semantics for the target languages (44).

Direct execution is also called the *single language* approach to the specification and execution of computational tasks. One conceptual example is Milner's CCS (242). In this approach a single (syntactic) language is used for both the specification and the imple-

mentation (execution) of the task. Consequently, one tact taken for such languages is to have a large number of constructs and statements. One subset of the language is designed for writing specifications, and another subset of the language is designed for efficient (and effective) execution of the tasks (263). Such languages are called *wide spectrum* languages, an example of which is the commercially available language REFINE (349). Another tact is to have all of the language constructs designed to be equally applicable to either specifications or executable forms, which is the philosophy behind UNITY (64). An interesting question is whether English is the single language for the human machine (at least those who think in English).

Contrasted with this single language approach is the *dual language* approach, which forms the basis for the *fair transition systems/temporal logic* approach of Manna and Pnueli (221, 220). One language is used for specifications, which with Manna and Pnueli is the language of temporal logic described in Appendix A, and the other language is used for the actual execution of the task. Because of the separation of the specification language from the language used for execution, the notion of task execution can be taken very literally. Thus the execution language could represent electrical, mechanical, chemical, etc. processes, and not just 'computational' tasks. Whenever humans write a program description in English, and then write the actual program in some executable language, the dual language approach is being used. Thus the traditional program development scheme is a dual language approach, with the efforts directed to verification and validation having to contend with the inherent problems of translating between different languages. This dual language approach includes both of the 'translation to' schemes described above.

Within four years of Floyd's seminal paper (126) on proving the correctness of programs (and three years before Manna's book (217) utilizing Floyd's concepts was published), papers started appearing suggesting that the technique of first writing the program code, and then proving its correctness, was not amenable to serious programming projects. Hoare (162) proposed that the proof be developed in concert with the program code. At any stage of the coding phase the code should be documented with the current assertions (including those Hoare defined as *invariants*) required to substantiate the proof of correctness of the completed code. These assertions are called the *preconditions* and

postconditions, since they state what is true before (pre) a statement (or block of statements) executes, and what is true after (post) the statements execute. Hoare claimed that the construction of the correctness proof during the coding would "prevent the intrusion of logical errors" (162). Unfortunately, Hoare's suggestion can easily degenerate into the same approach Floyd suggested, whenever a change in the program specification results in a change to the code that is significant enough to warrant a new correctness proof (which could easily result if the original proof used a relatively small number of assertions whose scope encompassed large portions of the program code). Another problem that was evident within ten years of Floyd's initial paper on using assertions in the verification process is that the "necessary assertions are often at least as lengthy and difficult to comprehend as the program they describe" (348). This trend towards increasingly complex proofs associated with increasingly complex programs led De Millo, Lipton, and Perlis to state in a somewhat infamous article (239):

> We believe that, in the end, it is a social process that determines whether mathematicians feel confident about a theorem - and we believe that, because no comparable social process can take place among program verifiers, program verification is bound to fail.

Recent research indicates that whatever techniques are being used in the development of software, they are still not preventing defective programs from being delivered to the customer. For example, during the period 1985-1988 the British Ministry of Defense found (thru the use of static code analysis) that almost ten percent of the individual software modules checked did not completely satisfy the specification, which led to the following quote from an official representative of the Ministry (83):

> Certainly the use of plain English with a little associated mathematics has not proved to be adequate. What is needed is a formal method of writing top level specifications and then proceeding in a mathematically rigorous manner to an ultimate implementation which can be proved to be correct, at least with respect to some narrow group of safety properties.

This research utilizes another approach which Wirth (355) dubbed *stepwise refinement*, which he and Dijkstra came to embrace as *the* method for software engineers to use

in developing programs (356, 105). The basic idea is to start with a formal specification, and then use the concept of successive mappings to transform this specification into the final executable code. Each step of the transformation process takes one or more (formal) constructs from the previous step and converts them into new (presumably more detailed) equivalent constructs. At the final step these constructs are just the statements of the target executable language. This process requires multiple (Wirth required at least two, one for the original specification and another for the executable code) formal languages within which the constructs at each step are formulated. As originally proposed by Dijkstra (103), stepwise refinement was a program verification technique. Each refinement step should be "very carefully carried out, so that it can be seen to preserve the correctness of the previous version of the program, then the final program must be correct by construction" (17). Note that this is the motivating philosophy for the language REFINE (349), although REFINE is only one formal language. Along with stepwise refinement, Wirth proposed the decomposing of tasks into subtasks, and data into data structures (the basic tenets of top down programming).

REFINE contains a large number of constructs to support the writing of formal specifications and efficient implementations. Because of the large number of statements REFINE actually violates Dijkstra's requirements for a 'good' programming language. Dijkstra stated that such a language should only contain the following statements (106):

A state preserving statement that always terminates (such as skip)

Multiple assignment

Scoped local variables within blocks

Statement composition

Alternative construct (such as if-then-else)

Loop construct (such as while-do)

Declarations for a few simple (such as boolean and integer) data types

An optional procedure call

UNITY however, comes much closer to matching this list of statements. UNITY has the multiple assignment, scoped local variables (within the ⟨ and ⟩), statement composition (the || operator), an alternative construct in the conditional assignment, and a small set of data types. UNITY does not have a skip statement, but a conditional assignment with a boolean expression that cannot be satisfied serves the same purpose. UNITY also does not have either the loop construct nor the optional procedure call, though UNITY does permit function calls.

The shift away from the 'write the code and then prove it correct' paradigm has continued, with proposed methodologies based on systematic 'correct' development over several phases, such that each phase is documented with respect to a complete formal (or at least rigorous) language (183, 184, 119, 253). Thus instead of just proving that programs are correct, the emphasis is on proving that the development is correct, even to the extent of not attempting any formal or rigorous proof of correctness, a concept that has been called 'proofless transformations' (25). This mind set shows in the following quote from Jones (184), which should be compared to the previous quote from Hoare:

> Program proofs can show the absence of bugs, not avoid their insertion.

Not only are these methodologies claimed to be more effective in producing executable code, but are also claimed to be more efficient, with Fagan quoting reductions in person hours of 10 to 40 percent (253) when compared to more traditional techniques.

The idea of correctness preserving transformations has been used for years in compilers (212, 289, 5, 350), in addition to being a tool for program development. Although there was strong criticism of this transformational approach in the mid 1970's, it has come to be widely accepted as an "established discipline in the field of programming methodology." (271) Additionally, the increased interest in functional programming languages has generated research into these transformations for compilers of such languages (121, 128). However, this research addresses the use of such transformations for program development.

Some of the earliest work in correctness preserving transformations was a series of papers by Burstall (54), and Burstall and Darlington (56, 87, 57), in which they classified

transformations (with such terms as folding, unfolding, and abstraction), and developed an automated system that would transform programs written in a high level language (of their own design) similar to LISP into equivalent programs in the same language. Their definition of equivalent was input/output equivalence for functions, and sequence of state equivalence for certain types of procedures. Other early work with program transformations includes Gerhart's work based on axiomatic semantics (136, 137), and the research of Manna, Broy, Bauer and others into the development of recursive constructs and their transformation into iterative constructs (224, 31, 188, 343, 49). More recent research also treats the transformation of recursive constructs into iterative ones (80), along with the transformation of iterative loops into other forms (346), and transformations based on the analysis of programs as directed acyclic graphs (DAG) (28, 35).

In addition to the system created by Burstall and Darlington, there are other automated tools designed to implement transformations on high level languages. For example, the Leeds Transformation System transforms programs with the goal of introducing greater 'structure' (215). A different approach is the Computer aided Intuition-guided Programming (CIP) project, which is based on transformations within a wide spectrum language (32). This wide spectrum language, denoted CIP-L, is designed for both specifications and implementable (in conjunction with an interpreter over a subset of the language) programs, so that the transformations can start with the specification and end with the implementation. A system similar in philosophy to the research here is the SPES specification and transformation system, which is designed so that "Transformations applied to the specification make it possible to modify it, with a view to building a program." (124). For a survey of these and other transformation systems (including the Programmer's Apprentice (302)) that is current through 1983, see the article by Partsch and Steinbruggen (272).

The program design process embodied in these transformation systems is one of starting with a formal specification that has been proven correct, and successively transforming this specification into the final program while preserving the correctness. This research does not attempt to address the entire design process, only the transformations performed on different versions of the specification, using the specification language UNITY. The concepts presented here should be applicable to other specification languages, which range

from complete formal languages such as an order sorted algebraic language (254), to an informal or nonrigorous language such as English. Although this research relies on manual transformations, Chapter VI presents techniques that could be partially au' ted.

2.4 Summary

Currently the most common verification technique is manual, due to the lack of automated verification tools (83). Recent research substantiates the claim that there is no evidence that automated verification is any easier or less time consuming than the manual approach. Consider that one report on the use of the automated tool SPADE estimated that verification of each (approximately 30 lines of Pascal code) procedure consumed 8.5 person-hours (281). Assuming that verification and proofs of correctness will remain a mostly manual process, a promising approach to software design is the use of formal specifications as presented in Chapter VI. This approach gives techniques for generating a formal specification that preserves the desired properties of the informal specification, and also for transforming formal specifications so that the desired properties of the specification prior to the transformation are preserved by the transformation, and are thus properties of the specification after the transformation.

III. Requisite Mathematical Background

This research is divided into two major efforts. The first is the metric space based development of computational models as an alternative to the more traditional (mathematical) development of these same models. The underlying philosophy behind this metric space development is that the mathematics can be supported by a standard advanced undergraduate or graduate level course sequence in real analysis. In keeping with this concept, this chapter presents certain fundamental mathematical concepts required for the following chapters besides the metric space material. The concepts presented in this chapter are based on the *class* approach to collections of things, as opposed to the more standard set based approach. Definitions and motivations for relations, functions, predicates, and category theory are given, all based on this class concept. The level of presentation is that of a course in mathematical analysis from the book by Apostol (8), and the reader familiar with this material may proceed directly to Chapter IV. Including this material in the real analysis courses (as presented here, particularly the category theory) can tailor these courses towards practical and theoretical research in computer science/engineering.

Section 3.1 presents the concept of a class, which is a hierarchical technique for grouping collections of things. By using classes, instead of the standard ZFS sets (i.e. Zermelo-Fraenkel-Skolem (364, 129, 182)), one can avoid certain paradoxes relating to sets. One of these paradoxes results from the claim that there exists a set U which is the set of all sets. Is then U an element of itself, and if so, what is the complement of U? The formalization of the concept behind this paradox is given in Russell's paradox (132, 73). Russell's paradox starts with a set A defined as:

$$A = \{x \in U | x \notin x\}$$

Thus A is the set of those sets (elements of U) that are not members of themselves. Although this definition seems strange, it is reasonable, since given any 'normal' set x, it should be easy to decide if x is an element of itself. Unfortunately, given a set that is not 'normal', this definition quickly leads to a paradox. And such a set is A itself! This is

because the definition of A implies that

$$A \in A \iff A \notin A.$$

The idea of a class though, permits the formation of the class of all sets, which, since it is not a set, does not lead to Russell's paradox (as long as there is no class of all classes).

Section 3.2 presents an overview of relations, functions, and predicates, within the framework established by the class concept of Section 3.1. The definitions given for these are in terms of classes, but do not contradict the standard definitions based on sets. Relations are basically ordered collections of things, that are considered as somehow *related* to each other. Very similar to relations, functions are often called *mappings*, to signify that given one thing, the function maps that thing into another (whereas a relation in a sense maps into more than one other thing). The section concludes with predicates, which are functions that map things into a special set whose only members are the symbols representing 'true' and 'false'. This research does not try to define exactly what true and false are, but instead accepts them as given atomic (cannot be defined in terms of other objects) objects.

This chapter concludes with Section 3.3, which introduces category theory (156, 247), and defines those terms used in the later chapters. Category theory is the study of categories, where each category is a combination of a collection of certain types of objects, and the collection of *morphisms* between these objects. A morphism is an abstraction of the concept of a relation or function. This section also gives a category based definition of a *theory*. The notion of a theory, as defined in this section, reflects the more abstract concept of a theory as containing *theorems* which can be proven to be true. This is a different presentation than the formalism of a theory as a category whose morphisms (mappings) represent projections and n-ary operations (112), or as a signature algebra plus equations (58). But the difference is very minor, and on a more abstract level the algebra based theory can also be shown to be a category (77), with different notions of what are the 'theorems'.

3.1 Classes, Genus, and Species

This section introduces the *class*, which is used as the basic concept of a collection of things instead of the traditional set (182). Since the categories used in this research are based on collections which cannot be treated as classic sets, this class concept yields a formalism that is used to reason about these collections. The class used here is based on Lewis Carroll's concept of a class: (61)

> "Classification", or the formation of Classes, is a Mental Process, in which we imagine that we have put together, in a group, certain things. Such a group is called a Class.

The following definition gives a formal definition of class, based on the ideas from Carroll of *putting together in a group*, and *things*. There is no attempt to try to define either of these concepts, but instead they are accepted as stated. Indeed, the following two statements could be called *hidden axioms*, where 'hidden' refers to the lack of a formal definition of the terms 'putting together in a group' and 'things':

1. There are things.
2. Things can be put together into groups.

That no attempt is made to define these two concepts here, reflects the 'semantic' component of the English language. That is, attempts to formally define these two concepts results in self reference, which should imply that these concepts cannot have 'meaning'. Yet they do have meaning to us, implying that this meaning results from some characteristic of the English language (to us) that is not formalized (i.e. not a formal language).

Definition III.1 *A* class *is a collection of things. Clases are formed in the following manner:*

1. There exists a nonempty class U of all things grouped together.

2. *Given a class called the* genus, *another class called the* species *can be formed by grouping together things from the genus. If X represents a class, then*

$$A \subset X$$

denotes that the class A is a species of X, and

$$x \in X$$

denotes that the thing x is one of the group of things that form the class X.

3. *Every genus has at least one species, which is the empty class that contains no things.*

As in set theory, it is necessary to differentiate between the thing a, and the class that contains only the thing a, say A. Thus $a \in A$ states that the class A contains the thing a, which implies that the class A is not an element of the class A, denoted by $A \notin A$, and also that the thing a is not a species of the class A, denoted by $a \not\subset A$. The reason $a \not\subset A$ is that the symbol a represents exactly the thing a, whereas $b \subset A$ is true only if the symbol b represents another class, that is a grouping together of things, that contains either no things or just the thing a.

Definition III.1 states the existence of a class called U, and also defines how other classes are formed from U. If the definition did not specify the class U, the only class that could be formed from the definition would be the empty class, since the second item in the definition requires that for a nonempty class to be formed there must exist another class. The empty class alone is not enough to form other classes, since it has no elements. So the definition contains two key concepts, that of a *base case*, that is a specific example of the item defined, and that of a *recursive* process for forming other items from the base case. This type of definition is called a *recursive definition.* In Definition III.1 the base case is the class U, while the recursive process is the formation of species from genus, where the 'first' genus is U.

The class U in Definition III.1 comes from the following quote from Lewis Carroll's *Symbolic Logic* (61):

> We may imagine that we have put together all Things. The Class so formed (i.e. the Class 'Things') contains the whole Universe.

The choice of U to denote the class of all things comes from the reference to the 'Universe'. The universe has not yet been specified, but in general this universe must contain all the things that are needed for some specific reason. This implies that another effort would need a different Universe based on different requirements. Thus this class of all things is not absolute, but contains just those things needed for a particular application.

Given the formation of such a class as U, then Definition III.1 can be used to form other classes, called genus and species. The terms genus and species also come from Lewis Carroll, and this usage of them as classes parallels his. But the concept of a class representing the universe, the class U, is not universally accepted (156, 247). If there can be no universal class of all things, then how else can classes be formed? Consider the following methodology for forming a hierarchy of collections of things that differs from this class based approach.

Start with the standard definition of a set (364, 182, 8), then consider the collection of all sets. Obviously such a collection is not a set, because of Russell's paradox. One term for such a collection is a class (defined differently from the class of Definition III.1), with a *conglomerate* defined as the collection of all classes (156). In this hierarchy, then, a class is what Hofstadter (166) would have called a meta-set, and a conglomerate as a meta-meta-set. Regardless of the names used, this process of starting with sets and then forming collections of all sets, and collections of collections of all sets, could continue indefinitely.

Thus Definition III.1 could have been reworded so that the set is the base case for the class definition, with additional classes formed based on the same conept of genus and species, except that a genus can only be formed whose elements are defined in terms of a given species. This approach can be summarized as starting with a given class and building other classes from it in both the 'upward' and 'downward' sense in abstraction. Contrasted with Definition III.1, this approach would not require the existence of the class U. A variant of this approach is to start with those things that are atoms, and then to form collections of atoms, and collections of collections of atoms, and so on. This generates a hierarchy of classes but only in the upward direction of abstraction (295).

The previous question regarded the impact of not having the class U. An interpretation of Definition III.1 would be to permit U to be a 'global' class for a particular application. This U would include all possible *atoms* needed for a specific application (such as this research effort), and classes would represent different collections of the atoms. An atom is a thing that can be represented with one symbol, such that any required 'meaning' associated with that atom can be derived from only that symbol. An example would be the forming of classes for the analysis of sets of real numbers. The atoms would be the individual numbers, plus other required things such as the unary minus operator, the binary addition operator, and other symbols that are standard within the realm of mathematical analysis (based on sets). Another example of a collection of atoms would be those based on the lambda calculus (301, 274); such a collection would not necessarily require the arithmetical operators, but would require other symbols such as λ. Using such an approach skirts the problem of the existence of a class such as U that represents (in Carroll's words) the whole Universe. Instead of the whole universe, U only represents that portion needed for a particular analysis.

The approach here is to define a universe U that contains all the atoms necessary for this research effort. This definition of U is somewhat informal, for a more rigorous mathematical treatment see Mac Lane (202) (which is based on an earlier paper by Bernays (37)). Starting with the standard definition of sets, the class U contains the following:

1. The set of all computable numbers (see Appendix B) is an element of U.
2. The arbitrary union or intersection of elements of U yields an element of U. (See Definition III.4 for the definition of class union and intersection)
3. If $D \in$ U and $f : D \rightarrow$ U is a function, then the class formed by evaluating f for every element of D is an element of U. (See Section 3.2 for a definition of function)
4. If $B \in$ U and $b \in B$, then $b \in$ U
5. If $C \in$ U and $S \subset C$, then S is an element of U.

A consequence of items 2 and 5 is the following result.

Theorem III.2 *Given a set C, $C \in U$, then the set of all subsets of C, denoted by 2^C and called the power set of C, is an element of U.*

Proof: Item 5 implies that

$$\forall S[S \subset C \Longrightarrow S \in \mathrm{U}].$$

Item 2 then implies that the union of all such subsets of C forms another element of U, even if the collection of all subsets is uncountable. (See Appendix A for a description of the predicate logic symbology. ■

Mac Lane required the inclusion of the set of all natural numbers in U, instead of the set of all computable numbers (from the first item), so that *sequences* could be formed (See Section 3.2 for a definition of a sequence). Since the set of natural numbers is a subset of the set of computable numbers, this U also allows the formation of sequences.

An example demonstrating the second item from the list of properties for the class U can be formulated using *bags* (225). Bags are identical to sets except that bags allow multiple elements to be represented with the same symbol. If the delimiters { and } are also used for bags, then the two bags A and B given as

$$A = \{a, b\}$$
$$B = \{a, a, b\}$$

are not equal. If both of these bags A and B are elements of U, then the second item states that the following bags are also elements of U

$$\{a, b\} \cup \{a, b\} = \{a, a, b, b\}$$
$$\{a, a, b, b\} \cup \{a, b\} = \{a, b, b, a, b, a\}$$

a process that can be continued indefinitely.

Now that classes can be formed, an important question is when are two classes *equal*. The answer to this question, Definition III.3, states that two classes are equal if and only if the following two conditions hold:

1. Both classes are species of the same genus.

2. Both classes contain exactly the same elements.

Definition III.3 (Class Equality) *Given two classes A and B,*

$$A = B \iff \exists X \forall x [A \subset X \wedge B \subset X \wedge (x \in A \iff x \in B)].$$

This definition uses a *sentence* from the predicate logic (225). Appendix A describes the concepts, symbology, and syntax from the predicate logic used within this research. Note that class equality, which requires the existence of a genus for the two equal classes, differs from set equality, which only requires that the two sets contain exactly the same elements (182).

Although a definition is given for equality between classes, there is no general purpose definition given for equality within a class. The definition of equality within a class is dependent upon the class, and is defined for each specific class as required. An equality definition for a class is not unique, so that the same class could have different definitions, each one tailored to a specific requirement. For this effort though, each class has either one equality defined, or where there are multiple equalities, each one is given a different name (see the discussion of extensional versus intensional functional equality in Section 3.2 for an example). For example, the class that contains only the natural numbers has the 'standard' equality based upon numerical evaluation, so $3 = 2 + 1$ in addition to $3 = 3$. But the class that contains all sets (see Section 3.3) uses a different equality, which requires the symbols representing the elements to be identical.

By defining classes in this manner Russell's paradox can be avoided. Genus and species are abstract names that represent different classes, so that instead of the set of all sets, which leads to Russell's paradox, there is the genus (or class) consisting of all those things that are sets. Given such a class as the genus of all sets, then any collection of sets would be an example of a species. A class can be both a genus with respect to another species, and a species with respect to another genus. One genus can have multiple species, and a class can be a species with respect to multiple genus.

Within a species A, $x \in A$ denotes that the thing represented by x is an *element* of the species, and the notation that A is a species of the genus X is given by $A \subset X$. With these conventions, the symbology for classes is identical to that for sets, so that

$$A \subset X$$

such that

$$A = \{x \in X | P(x)\}.$$

represents the definition of the species A as the class of those things (or elements) from the genus X such that the predicate $P(\cdot)$ evaluates to true. (See Section 3.2 for a definition of predicate)

An empty class is denoted by $\{\}$, so that given any class A

$$\{\} \in A.$$

The empty class is not the empty set, since the empty set is defined to contain no elements, whereas the empty class not only contains no elements, it is also a species of some genus. So if two classes A and B have different genus, then the empty class that is a species of A is not equal to the empty class that is a species of B, whereas in set theory all empty sets are considered equal. Note that within the class of all sets, the *unique* empty set is denoted by $\emptyset$.

Within these first two definitions of classes and class equality there is at least one major difference between sets and classes. Whereas a set is completely defined by its elements (182), a class is defined by both its elements and its genus. An interesting consequence of this dependence of a class upon its genus is the question of whether there can be another class (say defined in a different manner) that is equal to the universal class U. Since U has no genus (and is the only class that has no genus), then there cannot exist another class that is equal to U!

Another major difference between classes and sets is that sets contain unique elements, so that the union of the set $\{a, b\}$ and the set $\{b\}$ yields the set $\{a, b\}$, whereas

this is not necessarily true for classes (and isn't true for bags). The reason for not stating that elements of classes must be unique is that such a claim leads to a paradox (such as Russell's).

One consequence of the difference between sets and classes deals with the very concept that led to Russell's paradox, that is a set of sets. An element of a set of sets is itself another set, and this element could itself be a set of sets. This leads to the problem of using the same *name* for different objects. Consider the following sets:

$$A = \{1, 2, 3\}$$
$$B = \{\{1, 2, 3\}, \emptyset\}$$
$$C = \{B, \emptyset\}$$

Set theory does not have a convenient way to differentiate between A, B, and C. They are different types of sets, since A has a nesting level of one for its delimiters { and }, B has a depth of nesting of two, whereas C has a depth of nesting of three. So the set A is an element of B, but is not an element of C, even though both B, C are called sets of sets. Additionally, with only the symbolic representation of C as given above, there is no information to indicate what the depth of nesting is for C, unless B is specified in more detail. With classes, since a class is defined by both its genus and its elements, there is a more convenient way to differentiate between the three species A, B, and C, since they could each have a different genus.

Operations on classes correspond to those on sets, so that there is the *union, intersection, complement,* and *cartesian product.* All four are defined in an analogous manner to the set operations (182), but with important differences. Union and intersection are only defined for two species of the same genus, and the complement of a species is always another species within the same genus.

Definition III.4 *Given that A, B, and X are classes, define class* union $\cup$, intersection $\cap$, complement $^-$, *and* cartesian product $\times$, *in the following manner:*

$$A, B \subset X \iff A \cup B = \{x \in X | x \in A \vee x \in B\}$$

$$A, B \subset X \iff A \cap B = \{x \in X | x \in A \wedge x \in B\}$$

$$A \subset X \iff \bar{A} = \{x \in X | x \notin A\}$$

$$A \times B = \{(a,b) | a \in A \wedge b \in B\}.$$

Note that the $\vee$ symbol represents the *inclusive or*, which evaluates to *true* if either or both of its arguments are *true*, and the $\wedge$ symbol represents the logical *and*. Appendix A describes these and other symbols from the predicate logic. A result of this definition is that genus are closed under the operations of union, intersection, and complement. Additionally, the cartesian product of two classes is itself another class.

In Definition III.4 the symbols (a,b) denote the ordered pair of a and b. Some authors (8, 156) define ordered pairs in terms of sets as

$$\{(a,b) | a \in A \wedge b \in B\} = \{A, \{A, B\}\}. \tag{3.1}$$

Instead of this approach, this research defines a class of ordered pairs based on the following definition of ordered n-tuples:

Definition III.5 *A class of* ordered n-tuples, $n \in \mathrm{N}$, *denoted using the delimiters (and) as, for example*

$$(X_1, ..., X_n)$$

is defined to contain elements that are also denoted using the delimiters (and) as, for example

$$(x_1, ..., x_n)$$

such that

$$(x_1, ..., x_n) \in (X_1, ..., X_n) \iff [i \in \{1, ..., n\} \Longrightarrow x_i \in X_i].$$

Equality between elements of one class of ordered n-tuples, or equality between two classes of ordered n-tuples is defined by

$$(W_1, ..., W_n) = (Z_1 ..., Z_n) \iff (W_1 = Z_1 \wedge ... \wedge W_n = Z_n).$$

The zero-tuple, denoted (), is an empty class.

In testing for equality between two elements of a class of ordered n-tuples, each W_t, Z_t above are from the same class, although for $i \neq j$, W_i does not have to be from the same class as Z_j. If instead the test is for for equality between two classes of ordered n-tuples, then the W_t, Z_t themselves are classes, not necessarily all identical. Also note that the definition includes the ordering concept in that

$$X \neq Y \Longrightarrow (X,Y) \neq (Y,X).$$

Since the empty class is a species of any genus, then the zero tuple is also a species of any ordered n-tuple.

An example of n-tuples is the class whose elements are denoted by

$$(x_1, ..., x_n)$$

such that each x_t is an element from the set of computable numbers. This class of n-tuples is denoted by C^n, and its elements are called *vectors.*

Based on Definition III.5 is the following definition of a class called a *family.*

Definition III.6 *A* family *is a class that is denoted by a special form of a one-tuple, so that the family* $(X_a)_{a\in A}$ *is defined by*

$$(X_a)_{a\in A} = \{X_a \in X | a \in A\}$$

where X *is a genus, and* A *is a set called the* index *set.*

Another notation for a family, whenever the index set A need not be specifically given, is

$$(X_a)$$

If there is no confusion, then another equivalent formulation is given by

$$(Y)$$

such that

$$(Y) = (y_a)_{a \in A}$$

where

$$a \in A \iff y_a \in Y.$$

This nomenclature leads to the class of all 2 tuples of computable numbers being denoted by

$$(\mathrm{C}, \mathrm{C})$$

A class used in this research is the class of all sets. This class is not itself a set, but all of its elements are sets. Equality within this class is defined as the standard set equality, so that two elements are equal if and only if they are represented with the same symbol (182). This class is closed under the following operations:

1. Set Union.
2. Set Intersection.
3. Set Complement.
4. Set Cartesian Product.

The following are elements of this class:

1. $\{X, Y\}$ where X, Y are sets.
2. All functions from X to Y, where X, Y are sets. (See section 3.2 for a definition of function)
3. The empty set, denoted by $\emptyset$.

The following is a list of frequently used examples of elements from this class, all of which are also species of the genus R, the real numbers.

1. N = all natural numbers.
2. Z = all integer numbers.
3. Q = all rational numbers.
4. C = all Turing computable numbers. (See Appendix B)

To complete the background on classes one more item is needed, an extension of the axiom of choice for sets to an axiom of choice for classes. See section 3.2 for a formal definition of function.

Axiom III.7 (Axiom of Choice for Classes) *For any genus X, there exists a function C such that*

$$\forall S[(S \subset X \wedge S \neq \{\}) \Longrightarrow C(S) \in S].$$

A characteristic of this *choice function* C is that it is not a Turing computable function (see Appendix B).

3.2 Relations, Functions and Predicates

This section gives definitions for *relations, functions,* and *predicates* based on classes instead of sets. These definitions parallel those for the set-oriented constructs, such that the standard hierarchy, all predicates are functions, and all functions are relations, is also preserved by these definitions. These relations, functions, and predicates are used extensively in other definitions throughout this effort, and the predicate forms the basis for the modal and temporal logic (see Appendix A).

The concept of a relation is that it represents a subset of the cartesian product of two sets. For example, the equality relation over the natural numbers is a subset of $N \times N$ such that (2,2) is an element of the subset, but (2,3) is not. Two traditional definitions of a relation that are based on sets are:

1. Any set of ordered pairs is called a relation. (8)
2. Given a set X, a relation on X is a subset of $X \times X$. (305)

In the second definition, $X \times X$ is the set Cartesian product. The first definition is just a generalization of the second, and the two are equivalent if the relation is over some particular set. In both cases the relation is characterized as a set, and is a binary relation, that is a relation of two components. Since this research analyzes *classes* as they fit into the framework of *category theory*, the following definition for relation is not restricted to sets, nor is it restricted to just two arguments.

Definition III.8 *Given a class represented as an n-tuple, a* relation *over this class is a species of this class.*

The representation as an ordered n-tuple is a carry-over from set theory, so that the relation of Definition III.8 is a generalization of the concept of a relation as an ordered pair. If the given class in Definition is a two-tuple of sets, then the resulting relation is also a relation under the two traditional definitions given above. As an example, if the given class in the definition is

$$(U, U)$$

then any relation over this class is a further generalization of the two traditional definitions.

The nomenclature for statements about relations is the same as that used for set based relations. For example, if the given class is the 2-tuple

$$(X, Y)$$

where both X and Y are classes, then

$$(x, y) \in (X, Y).$$

denotes that the tuple (x, y) is an element of this class. If R denotes a relation over this class, that is

$$R \subset (X, Y)$$

then

$$(x, y) \in R.$$

denotes that the tuple (x, y) is an element of the relation R. Another representation that (x, y) is an element of R (which is commonly used for set-based relations) is

$$xRy.$$

Note that with respect to the standard definition of set based relations, other mathematical constructs that would fit the concept of a single or multi parameter relation, such as the subset relation over sets, are either simply called relations on an individual basis (225), or are implicitly considered predicates by definition (217, 225). Definition III.8 includes the single and multi parameter relations, while single parameter relations also fall under the definition of a 'family' (see Section 3.1).

Another representation for set based relations is based on the concept of a relation as a function that maps elements of a set of ordered pairs into the set $\{true, false\}$. For example, the equality relation over the set $\mathrm{N} \times \mathrm{N}$ can be considered a function which is

denoted by eq, such that

$$eq(2,2) = true$$
$$eq(2,3) = false$$

where $eq(x,y)$ means the function eq evaluated at (x,y). If $eq(a,b)$ evaluates to $true$, then (a,b) is an element of the equality relation, whereas if $eq(c,d)$ evaluates to $false$, then (c,d) is not an element of the equality relation. This example illustrates the concept of a relation as a function that maps elements of a class into the set $\{true, false\}$. Defined in this manner, eq can be used just as a symbol representing a set-based predicate (see Definition III.20) would be used.

The following definition concludes the required terminology regarding relations, plus supplies the concepts needed to support the analogy between relations and categories used in Section 3.3. This research adopts the convention that the terms 'relation' and 'partial relation' are synonymous.

Definition III.9 *A partial relation $R \subset (X,X)$ is called:*

Reflexive *iff*

$$\forall x[x \in X \Longrightarrow (x,x) \in R]$$

Symmetric *iff*

$$\forall x,y[(x \in X \wedge y \in X) \Longrightarrow ((x,y) \in R \Longleftrightarrow (y,x) \in R)]$$

Antisymmetric *iff*

$$\forall x,y[(x \in X \wedge y \in X) \Longrightarrow [((x,y) \in R \wedge (y,x) \in R) \Longrightarrow x = y]]$$

Transitive *iff*

$$\forall x,y,z[(x \in X \wedge y \in X \wedge z \in X) \Longrightarrow [((x,y) \in R \wedge (y,z) \in R) \Longrightarrow (x,z) \in R]]$$

Total *iff*

$$\forall x, y[(x \in X \wedge y \in X) \Longrightarrow ((x, y) \in R \vee (y, x) \in R)]$$

An Equivalence *iff R is reflexive, symmetric, and transitive.*

A Partial Order *iff R is reflexive, transitive, and antisymmetric.*

A Linear Order *iff R is a total partial order.*

A Strict Partial Order *iff R is transitive, antisymmetric, and*

$$\forall x[x \in X \Longrightarrow (x, x) \notin R]$$

A Strict Linear Order *iff R is a total strict partial order.*

Note that there exist equivalence relations that are also antisymmetric, such as the standard '=' relation, and there exist equivalence relations that are not antisymmetric, such as the 'mod n' induced equality relation over the natural numbers, where 0 equals 3 mod 3, but of course $0 \neq 3$.

Since a function within set theory is a special type of relation, then to preserve consistency the class based definition of a function should also be as a special form of a relation.

Definition III.10 *A* function *is a class of 2-tuples, that is a function is a species of the cartesian product of two classes, denoted by, say*

$$(F_1, F_2)$$

such that

$$\forall (f_1, f_2), (g_1, g_2)[((f_1, f_2) \in (F_1, F_2) \wedge (g_1, g_2) \in (F_1, F_2)) \Longrightarrow ((f_1 = g_1) \Longrightarrow f_2 = g_2)] \tag{3.2}$$

The class D such that

$$F_1 \subset D$$

is called the **domain** *of the function, while the class C such that*

$$F_2 \subset C$$

is called the **codomain** *of the function. If the function (F_1, F_2) is denoted by F, then*

$$F(f_1)$$

denotes F **evaluated** *at f_1, where*

$$\forall(f_1, f_2)[(f_1, f_2) \in (F_1, F_2) \Longrightarrow F(f_1) = f_2];$$

and

$$F : D \rightarrow C$$
$$D \xrightarrow{F} C$$

both denote the function F having domain D and codomain C.

Equation 3.2 implies that the function evaluation is defined for every element of the class F_1, which is equivalent to saying that F_1 is the *domain of definition* (8). Since the domain of definition can be a proper sublcass of the domain, that is there can exist elements of the domain that are not elements of the domain of definition, then the functions defined by Definition III.10 are the *partial* (i.e. not necessarily everywhere defined) functions. Just as with set based functions, if S denotes a class, then

$$F(S)$$

denotes the *image* class of S under F, and is given by

$$F(S) = \{y | \exists x[x \in S \wedge y = F(x)]\}$$

Within set based function theory, the image of any set is itself another set (82), which is consistent with this idea that the image of a class is another class.

This definition of function generalizes the standard definition of a function as a *mapping* from one set into another, such that for any function f,

$$f(x) \neq f(y) \Longrightarrow x \neq y.$$

Definition III.10 also does not conflict with the standard definition of a function based on Turing machines (155), since an undefined evaluation will not be equal to any other element of the codomain. The nomenclature used to designate that a function F is undefined at the element x is

$$F(x) = \perp$$

where $\perp$ is a symbol not used for any other purpose. Just as the standard definition requires that each individual element of the domain set must be mapped into just one single element of the codomain set, Definition III.10 also requires that all elements of the domain of definition class that are equal must map to equal elements of the codomain class. The function evaluation performs the mapping, and both the equality within the domain and the equality within the codomain are class equalities. The major difference between this definition and the standard set based defintion is that since classes do not require uniqueness of elements, there is the possibility that the domain and the codomain can have multiple elements that share the same symbol or are otherwise equal under the appropriate class equality. Within a class whose elements are sets, equal elements are exactly unique elements.

Definition III.10 states that the elements of a function are ordered 2-tuples subject to the constraint given by 3.2. This implies that any function is also a relation, since the 2-tuples that are elements of the function can also be the elements of a relation. The converse is not always true however, since there are relations that do not satisfy the constraint given by Equation 3.2, and thus are not functions. One example of a relation that is not a function is the relation $\leq$ over the natural numbers. Both of the two-tuples $(2,3)$ and $(2,4)$ are elements of this relation, whereas both of these tuples cannot elements

of a function, since that would require that $3 = 4$. Thus $\leq$ violates 3.2. As another example consider that all of the elements of the equality relation over the natural numbers can also be the elements of a function. Indeed this function is the identity function denoted by *id*, and defined by

$$n \in \mathrm{N} \Longrightarrow id(n) = n$$

Since any function can also be considered as a relation, Definition III.10 preserves the traditional concept of the set of all functions from a domain into a codomain, being a subset of the set of all relations between the domain and the codomain. This definition also preserves the traditional concept that there exists relations that do not satisfy the definition of a function. In terms of classes, this means that given a class of 2-tuples, the class of all functions whose elements include these 2-tuples is a species of the class of all relations also having these 2-tuples as elements. (Note that these definitions are consistent with those based on the concept of a relation as a function from one powerset into another, with the function being completely additive (111))

A relation $R \subset (X, Y)$ where X and Y are sets is also a function f

$$f : X \rightarrow 2^Y$$

where 2^Y represents the powerset of Y (the set of all possible subsets formed from the elements of Y). This statement stems from the concept of a function as a relation such that for every element of the domain the function evaluation yields a single element of the codomain, whereas for a relation the evaluation yields possibly more than one element of the codomain.

The following definition implies that within this discussion the terms 'function' and *partial function* are interchangeable.

Definition III.11 *A partial function* f,

$$f : D \rightarrow C$$

is called:

Total *iff*

$$\forall x[x \in D \Longrightarrow \exists y[y \in C \wedge y = f(x)]]$$

Surjective *iff*

$$\forall y[y \in C \Longrightarrow \exists x[x \in D \wedge y = f(x)]]$$

Injective *iff*

$$\forall x, y[f(x) = f(y) \Longrightarrow x = y]$$

Bijective *iff f is both surjective and injective.*

A Sequence *iff f is surjective and*

$$D = \mathrm{N}$$

The codomain of a sequence

$$\{f(0), f(1), ...\}$$

is also denoted by

$$\{f_0, f_1, ...\}$$

where each f_i is called the i^{th} term *of the sequence.*

Other common terms for surjective and injective are *onto* and *one-to-one* (or 1-to-1), respectively. These follow from the standard set based definitions, where a surjective function is one that maps the domain 'onto' every element of the codomain, while an injective function maps one element of the domain to one (and only one) element of the codomain. In class based definitions such as the ones here, these statements are relaxed to include equal elements, and not just unique elements.

As an example of a sequence, consider the function F defined as

$$F(n) = \begin{cases} 1 & \text{if } n = 0 \\ 1/n & \text{if } n \geq 1 \end{cases}$$

The (codomain) sequence generated by this function is

$$\{1, 1, 1/2, 1/3, ...\}$$

If F is a sequence, then the function G whose domain is the codomain of F is called a *subsequence* of the sequence F. Continuing with this example, one such subsequence is given by the function G, where

$$G(F(n)) = \begin{cases} F(n) & \text{if } n \in \{0, 2, 4, ...\} \\ \perp & \text{if } n \in \{1, 3, 5, ...\} \end{cases}$$

The (codomain) subsequence generated by G is

$$\{1, 1/2, 1/4, ...\}$$

The concept of a countable set is closely related to a sequence.

Definition III.12 *A set S is* **countable** *if it is the codomain of a bijective function whose domain is a subset of* N.

A countable set can have a finite number of elements, which means that the function from Definition III.12 either has a finite subset of the natural numbers for a domain, or else is not total. The *cardinality* of a finite set is just the number of elements in the set. A countable set can have an unbounded (or infinite) number of elements, in which case it is called *countably infinite*. This means the function from Definition III.12 has N as its domain, and can only be undefined for a finite subset of N. The cardinality of a countably infinite set is $\aleph_0$, called *aleph nought*. This is the cardinality of the natural numbers, the integers, and the rationals. Thus the term 'cardinality' refers in a sense to the number of elements in the set. Any set that contains only the terms of a sequence is countable.

Theorem III.13 *The following sets are countable:*

1. *$X \times Y$ where both X and Y are countable.*

2. *The union of a countable collection of countable sets.*

The proof of this theorem can be found in a standard analysis text.

Not all sets are countable, and the inclusion of the irrationals into the reals creates such an uncountable set.

Definition III.14 *A set S is* uncountable *if it is not countable.*

Theorem III.15 *The set of all real numbers is uncountable.*

Proof: (Based on the proof in Apostol (8)) This proof shows that the rational and irrational numbers between 0 and 1 form an uncountable set, so that all of the reals also form an uncountable set.

Assume that the set of real numbers greater than 0 and less than 1 is countable. Since this set is not finite, then there exists a total function R whose domain is N, such that

$$R(i) = 0.u_{i,1}u_{i,2}u_{i,3}...$$

is the decimal expansion of the i^{th} real number in the set. For real numbers these decimal expansions can be continued indefinitely (the number 0.1 can be written as 0.1000...). Now form the decimal expansion for the real number y, where

$$y = 0.v_1v_2v_3...$$

given by

$$v_n = \begin{cases} 1 & \text{if } u_{n,n} \neq 1 \\ 2 & \text{if } u_{n,n} = 1 \end{cases}$$

The number y is in the interval between 0 and 1, and yet it is not in the codomain of R. Thus the original assumption, that such an R existed making the set countable, is incorrect and the reals are uncountable. ■

This proof technique is called the Cantor diagonalization technqiue (110), and is used in general to prove that sets are uncountable. For example, this technique can be used to show

that the set of all total functions whose domain is the natural numbers and whose codomain is the natural numbers is uncountable, which means that there are an uncountable number of possible sequences.

An abstraction of a sequence is the *net*. Whereas a sequence is a surjective function whose domain is the natural numbers, a net (in the most general sense) is a surjective function whose domain is a *directed set*. A directed set is the two-tuple (X, R), such that X is a set and $R \subset (X, X)$ is a relation that is reflexive, transitive and satisfies (189)

$$\forall x, y[(x \in X \wedge y \in X) \Longrightarrow \exists z[z \in X \wedge (x, z) \in R \wedge (y, z) \in R].$$

A consequence of the Archimedian principle (305) is that the set of natural numbers N along with the relation $\leq$ is a directed set (see Section 3.3 for the definition of N as the category whose arrows are equivalent to the $\leq$ relation), so that any sequence is also a net. There are nets that are not sequences. For example, any surjective function whose domain is the real numbers R, which along with the relation $\leq$ forms a directed set, is a net but not a sequence. Nets permit the generalization of the concept of Cauchy sequences (see Section 4.1). For example, the book by Taylor and Lay (328) gives the definition of a *Cauchy net* whose codomain is a topological linear space. This definition of net suggests that a more general purpose definition of a sequence is a function whose domain is a countable directed set, which of course N is. This research elects to use the definition given in Definition III.11, since it parallels the standard definition given in analysis texts (305, 8). Since there exists a bijection whose domain is any countable set and whose codomain is the natural numbers N, this bijection can be composed with a function whose domain is N (a sequence), so that any net with a countable domain can be defined in terms of the definition of a sequence.

The reason nets require a directed set, instead of just a set with a partial or linear order, is that nets, like sequences, arise in situations where there is an unbounded number of elements from the codomain that in some sense are bounded in 'value', say with respect to some measure or norm, but have no maximum or minimum element. As an example, consider the sequence denoted by

$$\{1/n\}_{n \in \mathbf{N}}$$

which has elements that approach arbitrarily close to 0, but has no minimum element, only the infimum 0.

Given two classes that form the domain and codomain for a collection of functions, then this collection also forms a class, with the following definition of equality within this class.

Definition III.16 *Given two classes D and C, then for two functions f and g that are elements of the class of all functions whose domains are D and whose codomains are C,*

$$f = g \iff ((x,y) \in f \iff (x,y) \in g)$$

This definition of equality is based on the *extensional* view of set based functions, which says that a function is defined by its ordered tuples of elements. There is no requirement for any other representation of the function. Since there exist functions with an unbounded number of ordered tuple elements, then within the extensional view there exist functions that have no finite representation. Note that with respect to this definition of equality between two functions, if either function is undefined for a given element, then the other must also be undefined for the same element (229).

Another approach stems from the *intensional* or procedural view of functions, which defines a function based upon some finite representation. The intensional equality of functions would state that two functions are equal if and only if their representations were equal. Although these two definitions of equality may appear equivalent, they have important differences. A major difference lies with the comparison of functions that do not have finite representations. Two such functions could be equal under the extensional definition, but would not even be comparable under the intensional definition. This will not present a problem though, because this research (with just one exception) only requires functions that are Turing computable.

Theorem III.17 *Given two Turing computable functions, they satisfy the extensional equality definition if and only if they satisfy the intensional equality definition.*

Proof: The proof is the basis behind Church's Thesis, that the class of all finitely definable (lambda definable) functions is exactly the same as the class of all Turing computable functions (89, 301). ■

A consequence of this theorem is that the differences between the intensional and the extensional view of fuctions only matters for those functions that are not Turing computable.

The *inverse* of a function, if it exists, is a function that reverses the mapping of the original function.

Definition III.18 *Given the function F, if the relation defined by the class*

$$\{(y,x)|(x,y) \in F\}$$

is a function, then it is called the inverse *of F and is denoted by F^{-1}.*

If f is an injection, then it's easily shown that f^{-1} exists and is an injection, and if f is a total bijection, then f^{-1} is also a total bijection (305). Note that f being a bijection is not equivalent to saying that f is an isomorphism, even though the bijection property is a necessary condition for the isomorphism property (198).

Since a zero-tuple is an empty class, then

$$(\{\}, A)$$

represents the unique function F

$$F : \emptyset \rightarrow A.$$

This function appears again in Section 3.3 with respect to the category SET, and in some sense represents a 'generator of A'. The converse of the function $(\{\}, A)$ is the unique function whose evaluations are undefined for all elements of its domain, and is specified by (156)

$$f : A \rightarrow \emptyset.$$

The representation $(A, \{\})$ denotes this *totally undefined* function. This f is unique, since any other function that is also totally undefined over the same domain is equal to f with respect to the intensional view (331, 298).

An example of a function that is undefined for all of its domain is a function that satisfies

$$h : \mathrm{N} \rightarrow \mathrm{N}$$
$$h(n) = h(n + 1).$$

To evaluate $h(5)$ for example, first requires the evaluation of $h(6)$, which requires the evaluation of $h(7)$, and so on indefinitely. The everywhere undefined function satisfies this recursive definition, since $h(5) = h(6) = \ldots$ are all undefined. Note that this is not a unique solution for h, since $h(n) = n_0$, where n_0 is any natural number, is also a solution.

This next definition of the terms *retract* and *retraction* supports definitions used in category theory (see Section 3.3).

Definition III.19 *Given two sets X and Y, and the functions*

$$f : X \rightarrow Y$$
$$g : Y \rightarrow X$$

such that

$$x \in X \Longrightarrow g(f(x)) = x$$

then X is called a retract *of Y, and g is called a* retraction.

Note that given a function f whose domain is the set X, then any other function g, such that the composition of g with f forms the identity function on X, is called a retraction.

The final concept needed for this section is that of a *predicate*, which is a function that (if defined) evaluates to either true or false (232). This use of predicate includes the traditional concepts of predicate, temporal predicate, spatial predicate, and spatial temporal predicate (300).

Definition III.20 *A* **predicate** *is a function whose codomain is the set* $\{true, false\}$, *such that* $true$ *and* $false$ *are atomic symbols with respect to the class U.*

The stipulation that $true$ and $false$ be atomic symbols with respect to this universal class U is equivalent to saying that for any class formed from U these two symbols are interpreted in the same manner. Thus $true$ and $false$ have the standard meanings of true and false from the set based predicates of predicate logic.

Since a predicate is a function, predicates can be represented in a manner similar to the standard set-based representations. As an example, consider the predicate Q whose domain is the class of 2-tuples of natural numbers, such that

$$Q(n,m) = \begin{cases} true & \text{if } n+1 = m \\ false & \text{otherwise} \end{cases}$$

This predicate can be used to define a successor function, denoted by $succ$, whose domain and codomain are the natural numbers.

$$succ(n) = m \iff Q(n,m).$$

This definition of $succ$ uses $Q(n,m)$ just as a standard set-based predicate would be used, so that $succ(n) = m$ if and only if (iff) $Q(n,m)$ evaluates to $true$. Some representative elements of the predicate Q would include $((0,1), true)$, $((1,0), false)$, $((1,2), true)$, and $((2,2), false)$. This definition is a *finite representation* of the infinite collection of elements.

Since a predicate is also a function, and a function is a relation, there is a hierarchy formed by these concepts, which is shown in Figure 3.1. In Figure 3.1 $PREDICATE$ is the class of all predicates with a fixed domain, say D. Then $FUNCTION$ is the class of all functions with the same domain D, and whose codomain is fixed, say C, where C includes the atomic elements $true$ and $false$. This means that $RELATIONS$ is the class of all relations of the form (D, C). The $\bigcup$ symbol is just the class/species symbol $\subset$ turned up, so this figure shows that the class $PREDICATE$ is a species of the class $FUNCTION$, which is itself a species of the class $RELATION$. This hierarchy is also true for the standard set-based definitions of relations, functions, and predicates.

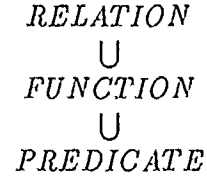

Figure 3.1. A Hierarchy of Classes

Since the Church-Turing thesis claims that the most general algorithmic method for generating unbounded numbers of things using a finite representation is the Turing machine (125), then for any relation, function, or predicate whose elements must be generated by some Turing machine (or another equivalent computing device (217)), there can only be a countable number of elements generated. This doesn't preclude the hypothesized existence of an uncountable number of elements, only that any computable or algorithmic technique for generating them can only produce a countable number. Appendix B presents a more detailed analysis of this idea of choosing a countable number of things out of an uncountable collection.

3.3 Categories and Theories

Not only has category theory emerged as a useful tool for theoretical (345, 330, 27, 216, 6, 205) and practical computer science (55, 59, 283), but it serves as a useful framework within which to embed mathematical analysis based on the concept of classes (247, 203, 156). For example, the primary category within which we perform our topological analyses is COMP, the category of complete metric spaces and the continuous mappings between them. (Note that we denote categories with boldface names) To answer the question of whether two metric spaces are 'equal', we must first show if the underlying sets for the metric spaces are equal, where set equality is defined with respect to the category SET, the category of all sets and total functions whose domains and codomains are these sets. Since the collection of all sets is not itself a set, we have that the category SET is actually based upon the class of all sets, and that it is the class definition that supplies the definition of equality within that class. We see that the basic concepts (such as the concept of set equality) regarding the different types of 'things' that we need are handled within the confines of the appropriate category, where each category is defined relative to some class.

Accordingly, this section presents the basic definitions and ideas from category theory that we need in the following chapters. Note that the only difference between our definition of category and the 'standard' one is the inclusion of the class requirement (see Definition III.23), which is implicit in Herrlich's definition (156), and implied in Mac Lanes's definition (Mac Lane defines a category based on a universal set, which is equivalent to our class of all sets, so that his small set is just our set) (203). We start with the definition of a *directed graph*, since we use directed graphs to represent categories and other concepts (such as the binary automaton tree of Section 4.2).

Definition III.21 *A* **directed graph** *is a four-tuple* (O, A, h, t)*, where* O *is a class whose elements are called the* **objects** *or the* **nodes**, A *is a class whose elements are called the* **arrows** *or the* **arcs**, h *is a total function*

$$h : A \to O$$

and t is a total function

$$t : A \rightarrow O$$

such that for any $a \in A$, $h(a)$ is called the head *of a, and $t(a)$ is called the* tail *of a.*

The directed graph (O, A, h, t) is called:

1. *A* small directed graph *if both O and A are sets.*
2. *A* countable directed graph *if both O and A are countable sets.*
3. *A* finite directed graph *if both O and A are finite sets.*
4. *A* locally finite directed graph *if both O and A are sets, and the set $\hat{A}$ given by*

$$o \in O \Longrightarrow \hat{A} = \{a | a \in A \wedge h(a) = o\}$$

is finite for all $o \in O$.

Although we have defined a directed graph in terms of 'classes', with one exception we only need directed graphs whose classes of objects and arrows are actually sets (the one exception is the definition of category). Thus we present the more useful (to our purposes) definitions of countable, finite, and locally finite directed graphs, which are based on directed graphs with 'sets' of objects and arrows. Since in the remaining chapters we only require directed graphs, versus nondirected graphs (72), the terminology *graph* can be used interchangably with directed graph without ambiguity. Note that the requirement for a locally finite directed graph states that for any node of the graph there exists only a finite number of arcs whose heads are this node, that is the outdegree (72) of every node must be finite. Also note that contrary to some definitions of a graph within the computer science literature (72, 330), we do not require that either the graph be finite or that a 'labelling function' be defined. The labeling of the vertices (arcs). which produces the so called 'labeled graph', is defined by the fact that the sets A and O and the functions h and t in our definition of a directed graph must satisfy the axioms of set theory (in particular the replacement axiom (203)). Thus (O, A, h, t) uniquely identifies the graph up to an 'equality' replacement of the symbols used in the sets O and A.

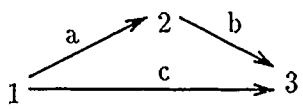

Figure 3.2. Example Directed Graph

Consider the directed graph defined by the four-tuple

$$(\{1,2,3\},\{a,b,c\},\{(a,1),(b,2),(c,1)\},\{(a,2),(b,2),(c,3)\}) \tag{3.3}$$

where the functions corresponding to h and t are defined using the tuple convention from Section 3.2. Figure 3.2 shows a graphical representation for this graph, with the functions h and t depicted by the arcs in the figure. Many different interpretations of this graph are possible, since the symbols in O can represent anything, even whole classes. If these elements from O represented individual sets, then the arcs of A could represent functions such that

$$a \in A \wedge o \in O \Longrightarrow ((h(a) = o \Longrightarrow dom(a) = o) \wedge (t(a) = o \Longrightarrow cod(a) = o))$$

where *dom* and *cod* are the domain and codomain functions respectively from Section 3.2. Another interpretation of the graph from Equation 3.3 is that the elements of O are just the natural numbers 1, 2, and 3, with

$$a \in A \Longrightarrow h(a) \leq t(a)$$

that is, the functions h and t are another representation of the binary relation $\leq$ over the natural numbers. With this representation, we see that the symbol N could be defined as such a (countable) directed graph with $O = \{1,2,...\}$.

Now that we have defined graphs, we need the concept of a mapping, or a *morphism*, between graphs.

Definition III.22 *Given two directed graphs* $G = (O, A, h, t)$ *and* $H = (V, E, s, u)$, *a* directed graph morphism *is a two-tuple* (f, g), *denoted*

$$(f, g) : G \rightharpoonup H$$

such that

$$f : O \rightarrow V$$
$$g : A \rightarrow E$$

where f *and* g *are total functions and*

$$\forall a\,[f(h(a)) = s(g(a)) \wedge f(t(a)) = u(g(a))]. \tag{3.4}$$

If

$$O \subset V$$

and

$$A \subset E$$

then (f, g) *is called a* directed graph imbedding.

The requirement stated in Equation 3.4 is that the directed graph morphism preserves the h (head) and t (tail) functions. Another way to state the assertion in this equation is to claim that the graph of Figure 3.3 *commutes*. In this graph the nodes represent the sets from the two graphs G and H, while the arcs correspond to the functions defined by h and t for G, s and u for H, plus f and g from the graph morphism. Note that there can be multiple arrows between any two vertices, and that these multiple arrows do not have to point in the same direction as they do in Figure 3.3. To claim that this graph commutes means that if we start at any vertex, say A, and follow the arrows to another vertex, say V, then the sequence of arrows does not affect the interpretation we give to the final result. This means that since A is a set, starting at A is equivalent to choosing an arbitrary $a \in A$, and following any arc from A is equivalent to applying the function denoting the arc to

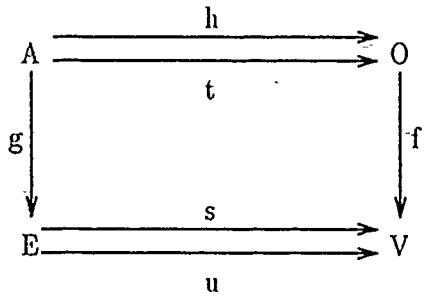

Figure 3.3. Commuting Graph Depicting Graph Morphism

a. Thus moving from A to O along the arrow h represents the evaluation $h(a)$, where $h(a) \in O$. If the next step is to follow the arrow f to V, then we have $f(h(a))$, an element of V. Going back to A, if we instead had followed the arc g, yielding $g(a)$ an element of E, and then the arc s, we would have stopped in V with the element $s(g(a))$. Our claim that the graph commutes implies that

$$f(h(a)) = s(g(a))$$

which is identical to Equation 3.4. Thus a graph that commutes is one where given a start and stop node, the interpretation of the result from traversing the path from start to stop is independent of the actual path followed. So Figure 3.3 depicts the graph that defines the requirement given by Equation 3.4 for graph morphisms. Just as with directed graphs, we can use the term *graph morphism* instead of directed graph morphism without any ambiguity.

If $(f, g) : G \rightarrow H$ is a directed graph imbedding, which is synonymous with a graph imbedding, then all of the nodes from G are also nodes of H, likewise all of the arrows of G are also arrows in H. This, together with the preservation of the heads and tails property, means that H is the graph formed by adding zero or more nodes and/or arrows to G. The identity mapping, whereby $G = H$, is the 'simplest' graph imbedding, in the sense that all other graph imbeddings can be viewed as 'extensions' of the identity mapping.

The following definition of a *category* demonstrates why directed graphs have also been called 'precategories'.

Definition III.23 *A* category *is a six-tuple* $(O, A, s, t, \circ, id)$ *such that:*

1. (O, A, s, t) *is a directed graph.*
2. $\circ$ *is a partial function*

$$\circ : A \times A \rightharpoonup A$$

such that

$$\forall f, g\,[(f \in A \wedge g \in A \wedge s(g) = t(f)) \Longleftrightarrow (\circ(f,g) \in A \wedge s(\circ(f,g)) = s(f) \wedge t(\circ(f,g)) = t(g))]$$

and

$$\forall f, g, h\,[(\circ(f,g) \in A \wedge \circ(g,h) \in A) \Longrightarrow \circ(\circ(f,g),h) = \circ(f,\circ(g,h))]$$

Denote $\circ(f,g)$ *by* gf.

3. *id is a total function*

$$id : O \rightharpoonup A$$

such that

$$\forall a, p\,[(a \in A \wedge p \in O) \Longrightarrow ((s(a) = p \Longleftrightarrow \circ(id(p), a) = a) \wedge (t(a) = p \Longleftrightarrow \circ(a, id(p)) = a))]$$

The two functions ob and mor have the class of all categories as their domain, such that

$$ob((O, A, s, t, \circ, id)) = O$$
$$mor((O, A, s, t, \circ, id)) = A$$

where the elements of O *are called the* objects *of the category, the elements of* A *are called the* morphisms *of the category, and for any category* $(O, A, s, t, \circ, id)$ *denoted by* $\mathbf{C}$, *the function* $hom_{\mathbf{C}}$

$$hom_{\mathbf{C}} : O \times O \rightharpoonup 2^A$$

is given by

$$\forall o, p\,[(o \in O \wedge p \in O) \Longrightarrow hom_{\mathbf{C}}(o,p) = \{a | a \in A \wedge s(a) = o \wedge t(a) = p\}]$$

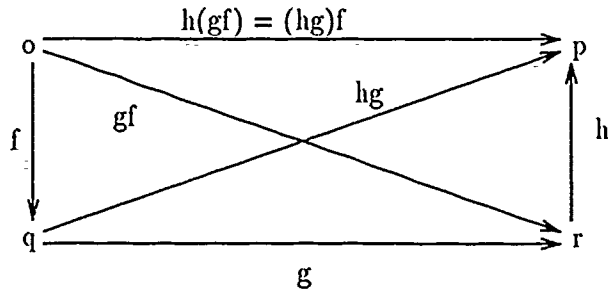

Figure 3.4. Commutative Graph for Composition of Arrows

Note that because of a category's roots as a directed graph, we can interchangably use the terms 'object' and 'node', and also the terms 'arrow', 'arc', and 'morphism'.

A category, then, is a directed graph with two additional functions, the first is a binary operator on the arcs (a composition or concatenation function), while the second 'creates' one additional arc (an identity arc) corresponding to each node than might not otherwise be included in just a directed graph. Item 2 in this definition states that this first function $\circ(f, g)$ is only defined for two arrows such that the tail of f is the same node as the head of g (the first assertion), and also that $\circ$ is associative over those arrows for which the pairwise evaluations of $\circ$ are defined (the second assertion). This is equivalent to claiming that the graph of Figure 3.4 commutes. We base our choice of denoting $\circ(f, g)$ by gf on the standard approach to denoting functional composition or concatenation, so that if $p \in O$, then

$$\circ(f, g)(p)$$

can be denoted by

$$gf(p)$$

or the redundant

$$g(f(p)).$$

Note that the use of the iff in the first assertion of item 2 ensures that the $\circ$ function is only defined for the appropriate pairs of arcs, while the use of the implication in the second

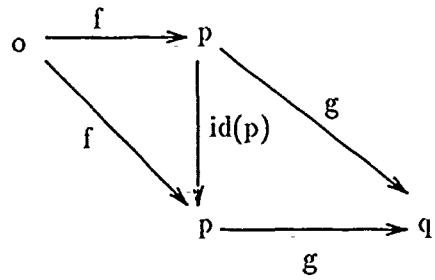

Figure 3.5. Commuting Graph That Corresponds to the Identity Function

assertion allows for the possibility that

$$o(f,g) \notin A \wedge o(g,h) \notin A$$

implying that

$$o(o(f,g),h) = o(f,o(g,h))$$

since both o evaluations would be undefined. Thus we allow that if two function evaluations are undefined, then they are equal. Note that this does not conflict with extensional equality (see Section 3.2). Item 3 asserts that the second function, *id*, ensures that for every element of O there exists one unique arrow (an element of A) that can be interpreted as an *identity* arc, that is it serves as both a left and a right identity with respect to the o operator. This is equivalent to saying that the graph of Figure 3.5 commutes.

For the reader interested in signature algebras (298, 270, 116) another way to view a category is as a two sorted algebra. Given a category C, the two sorts would be $ob(C)$ and $mor(C)$, where each arrow would be an operation of type

$$ob(C) \times mor(C) \rightarrow ob(C)$$

that is for each object and arrow (whose head is that object), this operation yields another object (the tail of the arrow) or is undefined. Although this is just an outline of the idea, the formalism would parallel the definition of a finite automaton as a two sorted algebra (330), where the set of states and the input alphabet are the sorts, such that the 'transition'

operation yields a state for each state and input symbol pair.

We require the following definitions for the remaining chapters.

Definition III.24 *The category* $(O, A, s, t, \circ, id)$ *is called:*

1. *A* small category *if both* O *and* A *are sets.*
2. *A* countable category *if* (O, A, s, t) *is a countable directed graph.*
3. *A* finite category *if* (O, A, s, t) *is a finite directed graph.*
4. *A* locally finite category *if* (O, A, s, t) *is a locally finite directed graph.*

Most of the categories we need are small categories, and indeed are countable categories.

Definition III.25 *The tuple* $(O, A, s, t, \circ, idr)$ *is a* subcategory *of the category* $(V, E, h, u, \bullet, id)$ *if and only if the following are all true:*

1. $(O, A, s, t, \circ, idr)$ *is a category.*
2. $O \subset V$
3. $A \subset E$
4. $s = h|_A$
5. $t = u|_A$

The notation $f|_S$ *denotes function restriction, such that*

$$(dom(g) \subset dom(f) \wedge cod(g) \subset cod(f) \wedge \forall x[x \in dom(g) \Longrightarrow g(x) = f(x)]) \Longleftrightarrow g = f|_{dom(g)}$$

If A *is a subcategory of* B *and*

$$ob(\mathrm{A}) = ob(\mathrm{B})$$

then A *is a* strict subcategory *of* B.

If A *is a subcategory of* B *and*

$$\forall o,p[(o \in ob(\mathrm{A}) \wedge p \in ob(\mathrm{A})) \Longrightarrow hom_{\mathrm{A}}(o,p) = hom_{\mathrm{B}}(o,p)]$$

then A *is a* full subcategory *of* B.

Denote that A *is a subcategory of* B *by*

$$\mathrm{A} \subset \mathrm{B}.$$

In general, a subcategory of a given category B is another category that contains some subsets of the nodes and arrows from B, while retaining all of the requirements for a category with respect to the identity and composition functions. If the subcategory retains all of the nodes from B, then it is a strict subcategory; whereas if it retains only some of the nodes but for each pair of nodes kept, all of the arrows connecting these nodes in B are retained, then it's called a full subcategory. The following examples both serve as examples of these definitions plus present the categories that we use in later chapters.

SET is the category whose objects are sets and whose morphisms are the total (single valued) functions between sets; that is $ob(\mathrm{SET})$ is the class of all sets, while for any two sets A and B, $hom_{\mathrm{SET}}(A,B)$ is the set of all total functions whose domain is A and whose codomain is B. Composition is just function composition, along with the standard identity function on sets. If instead of total functions we allow partial functions, then the corresponding category is called PFN. Thus we have that

$$\mathrm{SET} \subset \mathrm{PFN}$$

plus the fact that SET is a strict (but not a full) subcategory of PFN. This is because all of the objects in PFN are also objects in SET, and all of the arrows in SET are also arrows in PFN, but there exists arrows in PFN that are not arrows in SET (partial functions that are not total). REL is the category whose objects are also sets, but whose morphisms are the relations between these sets. Composition is standard relation composition, while the identity relation is just the identity function. Since all partial functions are also relations,

we have that

$$\text{SET} \subset \text{PFN} \subset \text{REL} \tag{3.5}$$

where PFN is a strict subcategory of REL.

Section 3.2 showed that any relation R between two sets A and B

$$R \subset A \times B$$

can also be represented by a set valued function f

$$f : A \rightarrow 2^B$$

A consequence of Theorem III.2 is that given any set A, the class of all sets also contains its power set 2^A. This means that for any given relation (arrow) within the category REL, there exists a partial function (arrow) within the category PFN that corresponds to this relation. This suggest a category MFN, whose objects are those sets such that

$$\forall A[A \in ob(\text{PFN}) \Longrightarrow (A \in ob(\text{MFN}) \wedge 2^A \in ob(\text{MFN}))]$$

and

$$\forall f, A, B[f \in hom_{\text{MFN}}(A, B) \Longrightarrow \exists C[C \in ob(\text{PFN}) \wedge B = 2^C \wedge f : A \rightarrow B]].$$

So we have that any object in MFN is also an object in PFN, and any arrow in MFN is also an arrow in PFN, which means that

$$\text{MFN} \subset \text{PFN}$$

but also

$$\text{SET} \not\subset \text{MFN}$$

$$\text{MFN} \not\subset \text{SET}$$

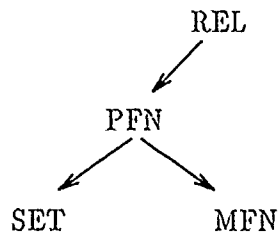

Figure 3.6. Directed Graph Representing a Category of Categories

since SET contains morphisms (whose codomain is not a set of sets) that are not morphisms of MFN, and MFN contains morphisms (which are partial) that are not morphisms of SET. Combining these results with those of 3.5, we can form the directed graph of Figure 3.6 where the arrows represent the subcategory $\subset$ relation. If we include the identity relations, and observe that $\subset$ among categories is transitive (i.e. composition is defined) and associative, then we see that Figure 3.6 also depicts (minus the identity and composite arrows) a *category of categories.*

Categories have certain special objects which prove useful in theoretical computer science, in particular the initial, terminal, and zero objects.

Definition III.26 *Given a category* C, *then* I, *where* $I \in ob(\mathrm{C})$, *is called a* preinitial object *if and only if*

$$\forall A[A \in ob(\mathrm{C}) \Longrightarrow \exists f[f \in hom_{\mathrm{C}}(I, A)]]$$

A preinitial object then, has at least one arrow corresponding to each other object in the category such that the head of the arrow is the preinitial object and the tail of the arrow is the other object.

Definition III.27 *Given a category* C, *then* I, *where* $I \in ob(\mathrm{C})$, *is called an* initial object *if and only if*

$$\forall A[A \in ob(\mathrm{C}) \Longrightarrow \exists! f[f \in hom_{\mathrm{C}}(I, A)]]$$

An initial object differs from a preinitial object in that there is only one arrow corresponding to each other object in the category such that the head of the arrow is the initial object and the tail of the arrow is the other object. In our category SET the initial object is the empty set $\emptyset$, since for any set A, there exists a unique function (156)

$$f : \emptyset \rightarrow A.$$

Although we do not do so, it's possible to construct a category of 'sorted algebras' such that the initial object in this category is the initial algebra of the computer science literature (330), a concept based on the idea of representing an algebra as a dir cted graph (336).

Definition III.28 *Given a category* C, *then* T, *where* $T \in ob(\mathrm{C})$, *is called a* terminal object *if and only if*

$$\forall A[A \in ob(\mathrm{C}) \Longrightarrow \exists! f[f \in hom_{\mathrm{C}}(A,T)]]$$

Any single element set is a terminal object in SET, since for any other set A, the unique morphism is the function that maps all the elements of A onto the single element.

Definition III.29 *Given a category* C, *then* Z, *where* $Z \in ob(\mathrm{C})$, *is called a* zero object *if and only if* Z *is both an initial and a terminal object.*

Although SET does not have any zero objects, the empty set is a zero object in the category PFN, since for any set A, the existence of the completely undefined unique function f

$$f : A \rightharpoonup \emptyset$$

implies that $\emptyset$ is a terminal object, in addition to being an intitial object.

The last category we present is very important to computer science. This category is GRPH, whose objects are countable directed graphs and whose morphisms are directed graph morphisms. Composition is just functional composition based on the definition of graph morphisms (Definition III.22).

Given these definitions, we conclude this section with some observations that other mathematical constructs can be defined as categories. For example, a set is just a category whose only arrows are the identity arrows, and so the objects of the category are the elements of the set.

A monoid is a category with just one object, whose arrows correspond to the elements of the monoid, so that composition of arrows represents the associative binary operator of the monoid, and the one identity arrow represents the identity element of the monoid. Since a monoid is a semigroup with an identity element (157), then given a finite alphabet of symbols Σ, the free monoid Σ^* (see Section 4.1) is a category with one object, an arrow for each finite length word (i.e. element of Σ^*), and one identity arrow that corresponds to the empty word.

The concept of the graph of a relation R, is based on the idea that we can graphically depict that $(a, b) \in R$ by a directed arrow whose head is a and whose tail is b. Since predicates are relations (see Section 3.2), this idea forms the basis for *semantic networks* (195). We can extend this concept so that different types of relations are reflected by their differing graphical characteristics. This results in the motivation for defining types of relations in terms of the most general graphical construct we have available, the category. The following definitions support our intuitive idea that there exists a strong relationship between categories and relations. For reference, the following standard definition of partial, linear, and well orders is given.

Definition III.30 *Given the class S, the relation R*

$$R \subset S \times S$$

is a preorder *if and only if R is reflexive and transitive.*
R is a partial order *if and only if R is reflexive, transitive, and antisymmetric.*
R is a linear order *if and only if R is a partial order and*

$$a, b \in S \Longrightarrow [aRb \vee bRa]$$

R is a strict partial order *if and only if R is transitive, antisymmetric, and*

$$a \in S \Longrightarrow (a,a) \notin R$$

R is a strict linear order *if and only if R is a strict partial order and*

$$a,b \in S \Longrightarrow [aRb \vee bRa] \text{ whenever } a \neq b$$

R is a well order *if and only if R is a strict linear order and for every nonempty subset E of S, there exists a unique element s of E (called the least element), such that*

$$[x \in E \wedge x \neq s] \Longrightarrow sRx$$

Given the preorder R, then the induced equivalence *denoted by S, is given by*

$$xSy \Longleftrightarrow (xRy \wedge yRx)$$

One example of a partial order is the logical implication relation denoted by $\Longrightarrow$ as applied to any set of well formed formulas from the predicate calculus (see Appendix A). Thus

$$(P,Q) \in \Longrightarrow \quad \text{iff} \quad P \Longrightarrow Q$$

That $\Longrightarrow$ is reflexive follows from

$$P \Longrightarrow P$$

Since

$$P \Longrightarrow Q$$

and

$$Q \Longrightarrow R$$

implies that

$$P \Longrightarrow R$$

then $\Longrightarrow$ is transitive. Antisymmetry results from defining the equality of two predicates as the 'if and only if' $\Longleftrightarrow$, that is if

$$P \Longrightarrow Q$$

and

$$Q \Longrightarrow P$$

then

$$P \Longleftrightarrow Q$$

and the two predicates P and Q are considered equal. So the $\Longrightarrow$ relation over a set of formulas is reflexive, transitive, and antisymmetric, thus forming a partial order over the set. Note that the equivalence induced by the $\Longrightarrow$ is just the $\Longleftrightarrow$, and that for any partial order the induced equivalence is the equality defined with respect to the antisymmetry property.

Definition III.31 *A category* P *is called a:*

Preorder *iff*

$$\forall A, B[(A \in ob(\mathrm{P}) \wedge B \in ob(\mathrm{P})) \Longrightarrow card(hom_{\mathrm{P}}(A, B)) \leq 1]$$

Partial order *iff* P *is a preorder and*

$$\forall A, B[(A \in ob(\mathrm{P}) \wedge B \in ob(\mathrm{P}) \wedge card(hom_{\mathrm{P}}(A, B) = card(hom_{\mathrm{P}}(B.A) = 1) \Longrightarrow A = B]$$

Linear order *iff* P *is a partial order and*

$$\forall A, B[(A \in ob(\mathrm{P}) \wedge B \in ob(\mathrm{P})) \Longrightarrow (card(hom_{\mathrm{P}}(A, B) = 1 \vee card(hom_{\mathrm{P}}(B.A) = 1)]$$

Well order *iff* P *is a linear order such that every full subcategory of* P *contains an initial object.*

card is the cardinality function whose domain is the class of all sets and whose codomain is the set of cardinality symbols.

The cardinality function generalizes the concept of how many elements a set contains, so that $card(S)$ evaluates to a member of N if the set S is finite, and evaluates to other nonnumeric symbols (such as $\aleph_0$) if the set is infinite (82).

Within a category that satisfies one of these order definitions, each arrow corresponds to an element of a relation R, so that if A is the head of the arrow, and B is the tail of the arrow, then $(A, B) \in R$ holds. The composition of arrows reflects the transitive property of the relation, while the identity arrows signify that the relation is reflexive, since the identity arrow for the object A corresponds to $(A, A) \in R$. Thus any category satisfies the basic requirements for a preorder, that is the arrows are reflexive and transitive. But we must restrict our category to have at most one arrow that corresponds to $(A, B) \in R$ for each pair of objects A and B, which is stated in the preorder item of Definition III.31. Thus our definition of a preorder satisfies the standard set based definition if we restrict our category such that the class of objects constitutes a set.

The additional constraint for the partial order is just the statement that the arrows must satisfy the antisymmetry property, which also agrees with the standard set based definition of a partial order. Note the equality in the statement for the partial order item contains an equality that is defined in terms of the class that contains the objects for the category.

The linear order constraint states that for any two objects within the category there exists an arrow whose head is one of the objects and whose tail is the other. This is equivalent to the set based statement that any two elements of a linearly ordered set are comparable with respect to the linear order relation. Note that if a category represents a linear order, then a preinitial (actually initial, since in a linear order category there is at most one arrow between two objects) object is the 'first' or minimum element in the linearly ordered class of the objects, and the terminal object is the 'last' or maximum element.

Since the difference between a set based strict linear order and a linear order is just the lack of the reflexive property (305), we can convert a linear order category into an object that we could call a strict linear order by simply removing the identity arrows. As a result, we do not differentiate between a linear order and a strict linear order in the following chapters. If there is some requirement for a strict linear order such that a linear order would not suffice, then we explicitly state that a strict linear order is needed, and we treat the strict linear order as a category without identity arrows. Thus the motivation for our definition of a well order as retaining the reflexive identity arrows, in contrast to the standard definition of a well order based on a strict linear order (305). Our constraint for a well order states that every full subcategory must contain an initial object, which is equivalent to saying that a set based well order satisfies the requirement that every subset contains a first element. As a result we have the following categorical wording of the Well Ordering Principle which states that any set can be well-ordered (305):

> Any small category can be mapped onto a well ordered category using a total bijective mapping on the objects and a partial surjective mapping of the arrows.

We use the word *mapping* in this statement since there may not exist a Turing computable function that satisfies this claim, because this principle is derivable from the Axiom of Choice for sets, and as we show in Appendix B, the choice function is not necessarily a Turing computable function.

Because a partial order can be treated as a category, many of the concepts from computer science (and other fields) can be easily recast as categories, thus providing additional formalism and structure. For example, consider the set of all CSP processes (see Section 4.3). One partial order on this set is denoted by

$$P \sqsubseteq Q$$

and means that the process P is more nondeterministic than Q, that is the set of execution sequences of P contains the set of the execution sequences of Q, and may contain execution sequences that are not possible for Q. This partial order has a least element, the process denoted by CHAOS (47), which is the process that can behave like any other process.

This partial order forms a category, such that the objects are the CSP processes, and the morphisms represent some type of reduction (or no change) in nondeterminism. Note that this ordering of the CSP processes is akin to Shannon's partial ordering of information representation (315), such that

$$P \leq Q$$

would denote that P is an abstraction of Q.

The standard definition of a sequence is as the codomain of a function whose domain is the natural numbers (8). Implicit in this definition is that the codomain set is linearly ordered by the $\leq$ order from N on the subscripts given to each element. Thus we can define a sequence as a linear order countable category.

Another example of a linear order countable category is any *temporal logic* (see Appendix A), since we adopt the assumption that time is countable (300). Thus we can use the tools from temporal logic to prove assertions about sequences, since both can be modeled with the same type of category (see Section 4.2).

We present this discussion regarding ordering relations for two reasons, the first being that we use ordered sets in the following chapters, and the second is that we can now present a more formal definition of what we previously defined in terms of sets. We start by denoting with a boldface natural number a linear order category that contains n objects if the number is n, such that for any arrow a, $h(a) \leq t(a)$. Thus our previous Figure 3.2 represents the category 3 without the identity arrows. Following this reasoning, we can define N to be the linear order category that contains one object for each natural number, along with the order $\leq$. In a similar manner we can define the categories R (a category with an uncountable number of objects), Z, Q, and C.

Consider a (proposed) countable category which contains an initial object, labeled F, and a terminal object T, such that $F \neq T$. We could interpret each object as an assertion of the predicate calculus, and each arrow as a logical implication. Thus each identity arrow would represent the fact that

$$P \Longrightarrow P$$

where P is any assertion. The initial object corresponds to the concept of 'false', such that for any assertion P

$$F \Longrightarrow P$$

is a unique implication, and the terminal object corresponds to 'true', where

$$P \Longrightarrow T$$

is a unique implication for any assertion P. Since $\Longrightarrow$ is transitive then composition of arrows is satisfied. But what of the associativity of arrows? Unfortunately, if P, Q, and R are all false, then

$$F \Longrightarrow (Q \Longrightarrow R)$$

evaluates to true, but

$$(P \Longrightarrow Q) \Longrightarrow R$$

is false. (This is why we define $\Longrightarrow$ to be right associative)

We can correct this lack of associativity for the arrows by defining the objects to be only theorems, the arrows to represent derivations, and adding the unique object A that represents the set of all axioms. This means that F is no longer an object. Thus, given a set of axioms and the rules of logical inference, we can *construct* the category such that each object is a provable truth (i.e. theorem), and each arrow represents a derivation. For example, Figure 3.7 depicts a 'slice' from such a category, so that

$$(P \longrightarrow R) \wedge (Q \longrightarrow R)$$

where $\longrightarrow$ means 'leads to the derivation of'. The basic axiom that relates $\longrightarrow$ to predicate logic is

$$(A \longrightarrow B) \Longleftrightarrow ((A \Longrightarrow B) \wedge A)$$

where the application of resolution (260) to the right hand side of the iff yields that both A and B are true. Note that the similarity of the $\longrightarrow$ operator with the symbology used for the 'type' of the implication operator is not entirely accidental (19). Note that this article

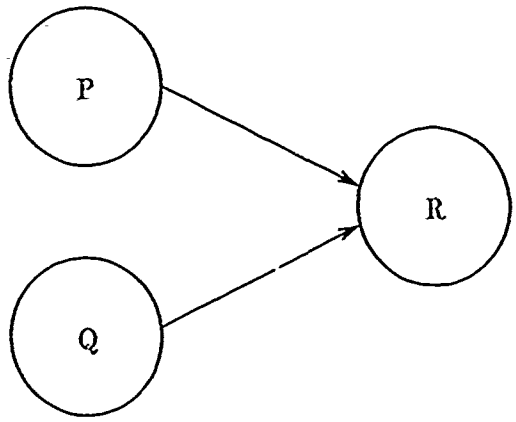

Figure 3.7. Arrows in a Consistent Theory

by Backhouse on constructive type theory contains a rigorous definition of the 'type' of

$$\neg P$$

as equivalent to the type of

$$P \Longrightarrow \emptyset$$

where $\emptyset$ denotes the *empty type*. This is not equivalent to saying that

$$\neg P \Longleftrightarrow (P \Longrightarrow false)$$

(a mistake that periodically appears in the computer science literature) since the unary operator $\neg$ cannot be defined in terms of any binary operator that has a different binding strength.

Another way to interpret Figure 3.7 is that either the truth of P or the truth of Q leads to the derivation of the truth of R. Thus we have the same kind of graphical interpretation that led to finite automata being labeled as 3 automata (222), or 'or' automata. Note that there can be multiple derivations of one theorem from another, so there can be more than one arrow whose head is one object and whose tail is another. This category has no (in general) initial object, since F is no longer the initial object, and we cannot assume a

unique derivation of any theorem from the axioms. The axioms object is a preinitial object, since we assume that every theorem can be derived from the axioms. We can retain the object T as the terminal object by assuming that the derivation of T given any theorem (or the axioms object) is unique.

We can correct the lack of an initial object by requiring the arrows to form a well ordering on the objects so that our category can be interpreted as a collection of objects that can be enumerated in some well ordered manner. That is, each arrow denotes the existence but not uniqueness of a derivation. The arrows would also represent the concept of a temporal ordering of the theorems, consistent with the idea of starting with only the axioms and then enumerating each theorem in a linear time ordered manner. Thus the axioms object is now the initial object, since it represents the object that 'comes first'. We use the word 'theorem' to denote derivable true statements, versus 'truths' which are true, but not derived. So that although there may be an uncountable number of truths that could be derived for a given set of axioms and rules of inference, there will only ever be a countable number derived by means of any given process (Turing machine). Now (assuming that the collection of objects are consistent (225)) since each object represents a true assertion,

$$P \longrightarrow Q \longrightarrow R$$

evaluates to true regardless of the parantheses grouping, so that the arrows are both composable and associative. This category corresponds to the concept of a *consistent theory* (225). Note that the true implications from our first attempt have not been lost, since if the axioms and rules of inference are those from the predicate calculus, then if $P \Longrightarrow Q$ is true, even though P may not be, there is an object of this category that represents this assertion. With respect to such a category that represents the theory of predicate logic, the Deduction Theorem (199) can be stated in the following manner:

Deduction Theorem The existence of an object representing

$$P \Longrightarrow Q$$

is not a sufficient condition for the existence of an object representing

$$P$$

Unfortunately, this approach to defining a *theory* as a category suffers from two major shortcomings. The first results from our assumption that we can represent any countable number of theorems as a well ordered set. Although the Axiom of Choice implies that we can (see Appendix B), there exists countable sets that cannot be well ordered by any Turing machine, such as the set of all second order predicate calculus theorems, or the set of all computable numbers (Appendix B). So, although we can claim the existence of such well-ordered categorical theories, we have no means to construct such categories. Within this thesis though, we restrict our analysis to only those theories that can be constructed by some Turing machine, so that this problem will not restrict our results. As we return to the concept of a theory as a category, we shall present additional definitions.

The second shortcoming is that we do not have a concise representation of the rules of inference for our theory category. We can simply write them out as they normally appear (i.e. equationally, or as formulas), but then we have the inconsistency that when applying them to the axioms object we must go 'inside' the object and pull out individual axioms, whereas when applying them to the derived objects we treat those objects as one entity. What we need is a technique for representing and applying these rules of inference that is as formal as the structure of the category itself. The following definition satisfies this second problem, although not the first.

Definition III.32 *A* **theory** *is a category* $\mathbf{T}$*, whose objects are categories such that:*

1. *There exists a preinitial object of* $\mathbf{T}$ *called the* **axioms**, *which is a countable category whose only arrows are the identity arrows.*
2. *Every morphism of* $\mathbf{T}$ *is a directed graph imbedding; the morphisms of* $\mathbf{T}$ *are called* **rules of inference**.
3. *Excluding the axioms object, every other object of* $\mathbf{T}$ *is a well ordered category whose initial object is the image of the axioms object with respect to one of the arrows of* $\mathbf{T}$.

A theory T *is called:*

Consistent *iff the false object is not an object of any category within* T.

Finite *iff* T *is a finite category whose objects are all finite categories.*

The definition of the axioms object ensures that the axioms are always a countable set, but not necessarily finite (see Ginzburg (138) for an example of an axiomatic theory that requires a countably infinite number of axioms). Since more than one rule of inference could be applied to the axioms to yield a given collection of theorems, the axioms is only a preinitial object, not necessarily an initial object.

We shall use the traditional interpretation of a theory, so that each object of the theory represents a collection of axioms and theorems organized as a well ordered category, such that the theorems can be derived from the axioms using the rules of inference. Additionally, we only require theories such that each object of the theory is a well ordered category containing an initial object that represents the axioms. With few exceptions, we shall analyze theories whose objects are small categories, that is the collection of axioms and theorems forms a set.

Note that Definition III.32 is slightly more general than those definitions that require a theory to be consistent. For example, Loeckx (209) defines a theory as a set of well formed formulas from the predicate logic that satisfies the following two conditions:

1. The set is consistent with respect to some semantic interpretation of the formulas.
2. Given any formula w from the set, all other formulas that can be formally derived from w are also members of the set.

Our definition of a consistent theory depends upon a semantical interpretation of the object F, the *false* object. For theories based upon some standard (see Appendix A) form of predicate (temporal) logic, the absence of the object F is equivalent to saying that if P is a theorem, then $\neg P$ cannot be a theorem. This results from the derivation in any standard logic of the theorem

$$(P \wedge \neg P) \longrightarrow false$$

which, if P and $\neg P$ are both theorems, implies that there exists an object within the category that contains F as an object.

If we informally define a *complete theory* as one such that for any given predicate P, we can prove that either P is an object of some category within the theory, or we can prove that it's not, then we have implied the existence of a *theory of theories.* This results from the interpretation of the proof of either P or $\neg P$ being true as a derivation of the theorem P or $negP$ within this theory of theories, where each other theory would be an object of this theory. Unfortunately, we run up against the same types of paradoxes that result from a set of all sets (see the discussion of Russell's paradox in the beginning of the first chapter), so that we refrain from attempting such a definition within this thesis. What we will do is to follow the standard practice of considering a complete theory as one within which all truths can be proven, or equivalently that every syntactically correct string of symbols within our logic can be proven either true or false, but not both. This means that for any wff (see Appendix A) P, either P is an object of one of the categories that comprises the theory, or $\neg P$ is, but not both. More formally, we present the following definition, which generalizes the concept of a complete theory so that we are not restricted to just wffs of the modal (temporal) logic.

Definition III.33 *Given the theory* T, *the class* C, *the atomic symbols true and false, the unique false object* F, *and the function*

$$- : \{false\} \times C \rightharpoonup C$$

such that for any two objects t *and* u *of* T

$$(c \in ob(\mathrm{t}) \wedge -(false, c) \in ob(\mathrm{u})) \longrightarrow F$$

then T *is* complete *with respect to* C *iff*

$$\forall c[c \in C \Longrightarrow (\exists \mathrm{t}[\mathrm{t} \in ob(\mathrm{T}) \wedge c \in ob(\mathrm{t})]$$
$$\oplus \exists \mathrm{u}[\mathrm{u} \in ob(\mathrm{T}) \wedge -(false, c) \in ob(\mathrm{u})])]$$

The statement $P \oplus Q$ reads 'P exclusive-or Q', and is true if either P is true or Q is true but not both are true, as opposed to the standard 'inclusive-or' of modal logic which is true if both arguments are true. (see Appendix A)

The function $-$ represents the concept of a complement with respect to the false object. This means that if the element c represents a true formula, then

$$-(false, c) \tag{3.6}$$

represents a formula that could be called the complement of the original formula. Within the modal logic (Appendix A), if c is a wff, then 3.6 evaluates to the wff $\neg c$. Thus this definition states that for every element of the base class C, a complete theory contains either that element or its complement. This implies that a theory could be incomplete if either certain elements of the base class or their complements were not contained within the theory (i.e. there exists formula which can neither be proven true or false), or if the theory contained both elements and their complements (i.e. there exists formula which can be proven both true and false, which is an inconsistent theory), or both.

This definition implies that the concept of a complete theory actually contains two essential ideas. The first is that every element of the base class must be proven either true or false, so that some means must exist to perform these proofs or derivations. And the second is that for any given formula within the base class there exists some method to determine what the complement of that formula is (if it exists within the class). For example, the propositional logic (non modal) along with the interpretation of functions as having domains and codomains the natural numbers is a complete theory. Consider that within this theory (and class) the following wff is true

$$\exists f[f(3) = 3]$$

since such an f is the identity function. whereas the wff

$$\forall f[f(3) = 3]$$

is false, as can be seen if f equals the successor function. It is not the wffs alone which are true or false, but the wffs as interpreted within the context of some theory, since if we consider the theory of all functions over the natural numbers that have fixed points at 3, then both of these wffs are true. Likewise we can find theories that make both of these wffs false.

Note that the preceding definition of a theory implies that there exists a theory that satisfies the standard definition of a 'logic' (139). This follows from the observation that if the morphisms within the well ordered categories that comprise the objects of the theory are equivalent to the $\longrightarrow$ operator, and the objects within the well ordered categories represent wffs from the modal logic (see Appendix A), then the theory satisfies the requirement that the set of formulas that constitute a logic satisfy the *rule of detachment*:

If F and $F \Longrightarrow G$ are members of the logic then G is a member of the logic.

The remaining chapters attempt to show that our 'theory' of theories is not devoid of practicality. That is, we do not want our 'theory' to fall into that 'category' described by Knuth in the following quote (194). (Emphasis is the author's)

> Some theory is developed which is very *beautiful*, and too often it is therefore thought to be *relevant*.

3.4 Summary

With the increased use of *category theory* in computer science, there is a need for collections of things which do not satisfy the traditional definitions of sets. This is because there are categories whose objects consist of all possible sets. As shown by Russell's paradox (132), the collection of all sets is itself not a set! Thus the need for collections that are not constrained by the axioms of set theory. The *class* concept of Lewis Carroll (61) supplies the necessary collections used in this research.

A class is defined to be a collection of things, such that other collections (classes) can be formed from any given nonempty class. No additional constraints are imposed on

classes in general, so that if a given class must satisfy the axioms of set theory, then that class must be identified as a set. The operations defined for classes are identical with those defined for sets: union, intersection, complement, and cartesian product. The concept of class equality (between classes) is also based on the set based definition (although not identical), while the definition of equality within a class is dependent upon the specific class. The informal definition of a univeral class that contains all of the things needed for this research effort is given, although this universal class is not explicitly used in the subsequent chapters.

The definitions given for *relations, functions,* and *predicates* parallel those from standard analysis based on sets, except that the class replaces the set, and *partial* functions replace the *total* functions assumed in many analysis texts. This means that certain definitions, such as those for a *sequence* and *countable* sets, are slightly different than the standard ones from a text such as Apostol (8), because partial functions are possible. The definitions relating to relations, functions, and predicates given in this chapter are used extensively throughout this research. For example, most of the concepts presented in the next Chapter are based on relations and functions, while predicates are used extensively in all of the following chapters to formalize the concept of assertions that are true.

The definition given for a *category* is equivalent to the standard definition, but is worded in a slightly different manner to draw on the analogy with *directed graphs*, a concept more familiar to computer and software engineers. Although categories are not strictly required by the subsequent chapters, one reason for including this material is the prevelance of category based research in the literature. Additionally, standard texts on categories include analysis of those categories whose objects are complete metric spaces. The next chapter shows that different types of computational models actually generate complete metric spaces, so that the results from these texts can be applied to the analysis of computational models.

IV. Imbedding Computational Models Within the Category of Complete Metric Spaces

This chapter presents the topological analysis of computational models. The mathematical tool that enables this topological view is the complete metric space, that is, the objects that comprise the category COMP of complete metric spaces defined in the last chapter (Section 3.3). This approach based on metric space topology, while not new (95, 6, 7), does present an alternative to the more 'standard' approach based on domains and resursive equations, which was pioneered by Scott (313), Stoy (325), and Strachey (324, 314).

Specifically, this chapter demonstrates that the finite automaton, the CSP, and the UNITY models can all be recast as metric spaces, based on the metric given in the next section. The same techniques can also be applied to other major computational models that support concurrency, such as Milner's CCS (242), Petri Nets (279), and Hennessy's EPL (154), so as to create other complete metric spaces that permit the type of topological analysis performed in this chapter. Section 4.4, which presents the metric space based on UNITY programs, also lays the groundwork for the transformational techniques of program development given in Chapter VI.

This chapter comprises the first major division of this research effort, the relationship between computation theory and the topological analysis of the computational models. Section 4.1 presents the metric on strings from the *free monoid* Σ^*, where Σ is the alphabet of symbols, that forms the basis for the metrics used in the remaining sections. Section 4.2 addresses the basic model of computation, the finite automaton, and demonstrates the correspondence between the metric space of finite automata, its completion, and computational power. Section 4.3 also develops a metric space, based on the CSP model of computation. The primary purpose for choosing CSP is that it is significantly different (syntactically) than the primary model used in Chapter VI, UNITY. Also, whereas the philosophy behind CSP is closely related to an applicative approach, UNITY is basically an imperative model. Section 4.4 then completes the chapter by first defining an alteration to the UNITY execution model called the *standard execution model*, and secondly showing

that a metric space of UNITY programs under both the UNITY execution model and the standard execution model can be defined in terms of the CSP metric space.

The motivation for the topological analysis of computational models stems from the requirement to construct a framework within which to develop the transformations of Chapter VI. But this topological analysis was also inspired by the work of authors such as Day (92), who presented (at that time) the tools of topology as a new way of studying computational models. This early topological analysis followed from the observation that a semigroup based on the composition operator $\circ$, can be defined as a topological space S, plus the continuous associative function $\circ$

$$\circ : S \times S \rightarrow S$$

The third section develops a complete metric space based on Hoare's concurrent computational model Communicating Sequential Processes (CSP) (165). CSP is chosen as a representative from a class of concurrent models that are based on the idea of modeling the *behaviour* of the computation. This behaviour concept common to all of the models within this class is summarized by the following definition from Milner's book on a Calculus of Communicating Systems (CCS) (242):

> We define a program to be a closed behaviour expression, i.e. one with no free variables.

Thus an informal definition of this class would be that it contains those computational models that represent the behaviour of the computation, such that given any model the complete instantiation of the variables (model variables, not program variables) results in the specification of a program. The representation of the behaviour includes a representation of both the initial state (see Appendix A for a definition of state) of the computation, and the resulting sequence of states that follow the intial state until the computation halts (assuming it halts). Within this class exists two natural subclasses, the first consisting of those models whose representations are primarily algebraic, such as CCS (although CCS does straddle between both subclasses), CSP, and Hennessy's Example Process Language

(EPL) (154). Those models whose representations are primarily graphic make up the second subclass, and includes such models as Petri nets (279).

4.1 *Complete Metric Spaces and the Star Closure*

This section lays the foundation for the topological analysis of computational models and the categories based on these models. Our topologies result from metric spaces, where a metric space is a combination of a set and a function called a metric. Within this section we develop a metric space based on the star closure that both serves as the basis for later analysis, and also serves as an example of the topological concepts. First we give the formal definition of a metric space.

Definition IV.1 *A* metric space *is a two-tuple* (X, d) *where* X *is a nonempty set, and* d *is a function*

$$d : X \times X \to \mathrm{R}^+ \cup \{0\}$$

where R^+ *denotes the set of positive real numbers, such that*

1. *(Strictly Positive)* $\forall x, y\,[(x \in X \wedge y \in X) \Longrightarrow (d(x,y) = 0 \Longleftrightarrow x = y)]$.
2. *(Symmetry)* $\forall x, y\,[(x \in X \wedge y \in X) \Longrightarrow d(x,y) = d(y,x)]$.
3. *(Triangle Inequality)* $\forall x, y, z\,[(x \in X \wedge y \in X \wedge z \in X) \Longrightarrow d(x,y) \leq d(x,z) + d(z,y)]$.

A function d that satisfies the requirements of Definition IV.1 is called a *metric*, and is interpreted as a measure of the 'closeness' or distance between two elements of the set. The elements of the set X are called the *points* of the metric space. If we replace the third item in the list of requirements for d in Definition IV.1 with

$$\forall x, y, z\,[(x \in X \wedge y \in X \wedge z \in X) \Longrightarrow d(x,y) \leq \max\{d(x,z), d(z,y)\}]$$

then the function d is called a non-Archimedian or an ultra metric (43). Note that if d is an ultra metric then it is also a metric, but the converse is not necessarily true. The second defintion we need is for a *Cauchy sequence.*

Definition IV.2 *Given a metric space* (X, d), *the set*

$$\{x_n\}_{n=1}^{\infty} \subset X$$

which is called a sequence of points in the metric space (X, d) *and is denoted by* $\{x_n\}$, *is a* Cauchy sequence *if and only if:*

For every δ, $\delta \in \mathrm{R}^+$, *there is an* N, $N \in \mathrm{N}$, *such that*

$$\forall n, m[(n \in \mathrm{N} \wedge m \in \mathrm{N} \wedge n \geq N \wedge m \geq N) \Longrightarrow d(x_n, x_m) < \delta]$$

where

$$\mathrm{N} = \{0, 1, 2, ...\}$$

A Cauchy sequence (within some metric space) then, is an ordered sequence of points, such that the distance between them as defined by the metric decreases as we go further into the sequence. As an example of a Cauchy sequence, consider the metric space (Q, d), where Q is the set of rational numbers, and

$$d(x, y) = |x - y| \text{ where } x, y \in \mathrm{Q}.$$

The sequence defined by

$$x_n = \sum_{i=0}^{n} 1/i! \tag{4.1}$$

is a Cauchy sequence of rational numbers.

Although Definition IV.2 implies that Cauchy sequences must have an infinite number of points, sequences that only have a finite number of points can still be covered by this definition by simply extending the finite sequence indefinitely by repeating the 'last' point. For example, the sequence

$$\{3, 2, 1\}$$

becomes the Cauchy sequence

$$\{3, 2, 1, 1, 1, 1, ...\}.$$

The Cauchy sequence is a sequence whose elements become increasingly 'closer' (with respect to the distance concept of the metric) to something that may or may not be an element of the metric space that the Cauchy sequence lies in. This 'something' is called

the *limit* of the sequence, a concept formalized in the following definition. This definition also defines exactly what types of sequences have limits.

Definition IV.3 *A sequence* $\{x_n\}$ *of points in a metric space* (X, d) *is* convergent *if and only if there exists an* x, $x \in X$, *such that:*

For every δ, $\delta \in \mathrm{R}^+$, *there exists an* N, $N \in \mathrm{N}$, *such that*

$$\forall n \left[(n \in \mathrm{N} \wedge n \geq N) \Longrightarrow d(x_n, x) < \delta \right].$$

This x *is called the* limit *of the sequence and is denoted by*

$$\lim_{n \to \infty} x_n = x$$

A sequence that is not convergent is divergent.

Another nomenclature for the limit point of a Cauchy sequence $\{x_n\}$ is x_∞, if such a limit exists, since not all Cauchy sequences have limits that are elements of the set X. If a Cauchy sequence has a limit, then the elements of the sequence become arbitrarily 'close' (with respect to the metric) to this limit, and the sequence converges. A theorem from topology states that this limit is unique if it exists (8).

Return to the example of the sequence of rational numbers generated by Equation 4.1. Although this sequence is a Cauchy sequence within the metric space (Q, d), the limit point of the sequence is not a rational number, and so is not an element of the metric space. This means that the sequence is divergent. However, within the metric space (R, d), using the same metric d, this sequence is convergent, since the limit point of the sequence is the irrational number denoted by e. That this sequence converges in the metric space (R, d), is a consequence of another theorem from topology that states that all Cauchy sequences in (R, d) are also convergent (256). Any metric space with this property that all Cauchy sequences are convergent is called a *complete* metric space.

Definition IV.4 *A metric space* (X, d) *is* complete *if and only if every Cauchy sequence in the metric space is also convergent.*

Within any complete metric space, if we take a sequence of points that become progressively closer together with respect to the metric, then we are assured that there is an element of the metric space that the elements of the sequence eventually have zero distance from. For infinite sequences this eventuality may take a countably infinite time.

We conclude this section with an example of a complete metric space that will not only serve as an example, but will supply some needed tools for later analysis. This example is based on the *star closure* of a finite set, where the finite set often represents the individual symbols or *atoms* of a given language.

Given a finite set of symbols, Σ, we present a metric for the space of all strings of symbols from Σ of finite length. Then we motivate the definition of infinite strings of symbols from Σ as the completion for this metric space.

Definition IV.5 *Given a finite set* Σ, *the* star closure *of* Σ, *denoted by* Σ^*, *is the set that contains only those elements given by:*

1. $\Lambda \in \Sigma^*$
2. $\forall\alpha[\alpha \in \Sigma \Longrightarrow \alpha \in \Sigma^*]$
3. $\forall\beta,\gamma[(\beta \in \Sigma^* \wedge \gamma \in \Sigma^*) \Longrightarrow \beta\gamma \in \Sigma^*]$

where

$$\Lambda \notin \Sigma$$

and

$$\forall\delta[\delta \in \Sigma^* \Longrightarrow (\Lambda\delta = \delta\Lambda = \delta)]. \tag{4.2}$$

From this definition we see that Σ^* is the set that contains all possible finite strings that can be formed by concatenating the symbols representing the elements of Σ plus the symbol for the empty string Λ (138). For example, if $\Sigma = \{a, b\}$, then

$$\Sigma^* = \{\Lambda, a, b, aa, ab, ba, bb, aaa, aab, ...\}.$$

The Λ symbol represents the concept of an identity with respect to the concatenation operation. So for our sample $\Sigma = \{a, b\}$, we have

$$\Lambda\Lambda = \Lambda \quad \Lambda a = a \quad ab\Lambda = ab \text{ etc.}$$

Sometimes we need to restrict our analysis to strings of symbols from a set Σ such that each string has at least one such symbol. We denote this set by Σ^+.

Definition IV.6 *Given a finite set Σ, the set denoted by Σ^+ is defined by*

$$\Sigma^+ = \Sigma^* - \{\Lambda\}$$

where Λ is that unique element of Σ^* that is the concatenation identity.

efore we define the metric for the set Σ^*, we first need to define two functions len and prefix, where

$$\text{len} : \Sigma^* \rightarrow \mathbb{N}$$

such that

$$\text{len}(\Lambda) = 0$$
$$\forall x \left[x \in \Sigma^+ \Longrightarrow \exists \alpha, u \left[(\alpha \in \Sigma \wedge u \in \Sigma^* \wedge \alpha u = x) \Longrightarrow \text{len}(x) = 1 + \text{len}(u) \right] \right].$$

Our function len is just the length of the string, that is the number of symbols from Σ that comprise the string, with the convention that the empty word Λ has zero length. Next we define the set valued function prefix, where

$$\text{prefix} : \Sigma^* \rightarrow 2^{\Sigma^*}$$

such that

$$\text{prefix}(\Lambda) = \{\Lambda\}$$
$$\forall x \left[x \in \Sigma^+ \Longrightarrow \forall u, v \left[(u \in \Sigma^* \wedge v \in \Sigma^* \wedge uv = x) \Longrightarrow u \in \text{prefix}(x) \right] \right].$$

Given these definitions for len and prefix, a metric for the set Σ^* is the function denoted by σ, where

$$(x \in \Sigma^* \wedge y \in \Sigma^*) \Longrightarrow \sigma(x,y) = \begin{cases} 0 & \text{if } x = y \\ \inf\{1/2^k | k = \text{len}(u) \wedge u \in \text{prefix}(x) \cap \text{prefix}(y)\} & \text{else} \end{cases} \tag{4.3}$$

for all such x and y that are elements of Σ^*.

Since the len function has $\mathbb{N}$ for its codomain, k is nonnegative and this σ function does map pairs of elements of Σ^* into the nonnegative reals. Also, $\sigma(x,y) = 0$ whenever $x = y$. So, to prove that σ is indeed a metric we need to show that:

1. $\sigma(x,y) = 0 \Longrightarrow x = y \qquad \forall x,y \in \Sigma^*.$
2. $\sigma(x,y) = \sigma(y,x) \qquad \forall x,y \in \Sigma^*.$
3. $\sigma(x,z) \leq \sigma(x,y) + \sigma(y,z) \qquad \forall x,y,z \in \Sigma^*.$

It has already been proven that σ satisfies all three of these requirements (43), but we repeat here the proof of the first item from the list to demonstrate an important concept.

Consider two elements from Σ^*, say x and y. If $\sigma(x,y) = 0$ and we do not know that $x = y$, then it must be true that

$$\inf\{1/2^k | k = \text{len}(u) \wedge u \in \text{prefix}(x) \wedge u \in \text{prefix}(y)\} = 0.$$

But this implies that k is unbounded, since if k were bounded then the inf would simply be the minimum of the set, a fixed nonzero number. Having k unbounded means that the two elements x and y have identical prefixes of unbounded length, which means that x and y are identical for any finite number of symbols. Our interpretation of such an x and y is that they are equal (305). So σ as given by 4.3 is a metric, and (Σ^*, σ) is a metric space for any finite set Σ.

Although the metric σ is the only one required for this analysis, other metrics exist for the set Σ^*, each with its own topological properties. For example, versus the metric σ which results in open balls that are also closed, the metric $\hat{\sigma}$ generates open balls that are

not closed, where $\hat{\sigma}$ is given by

$$\hat{\sigma}(x,y) = \begin{cases} 0 & \text{if } x = y \\ \sum_{k=1}^{\infty} 1/2^k \chi_k(x,y) & \text{else} \end{cases}$$

where

$$\chi_k(x,y) = \begin{cases} 1 & x_k \neq y_k \\ 0 & x_k = y_k \end{cases}$$

and for $z \in \Sigma^*$, z_k denotes the kth symbol of z, such that $z_k \in \Sigma$.

Given a finite set Σ, Σ^* is the set that contains all possible strings of symbols from Σ of *finite length*. We emphasize the finiteness of the length of the strings because it is possible to consider strings that have infinite length. To demonstrate this concept, consider the following example based on the real numbers. If

$$\Sigma = \{-, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, .\}$$

then we can form strings of symbols from Σ that represent real numbers. So some elements of R, the set of all real numbers, can be represented by the elements of Σ^*. If we temporarily (for the duration of this example) denote the bijection between the real numbers and their representations with the equality symbol, then we can ask whether $\Sigma^* = \mathrm{R}$. The answer is no, since $-1.1-$ is an element of Σ^* but does not represent a real number. A more subtle question then, is if $\mathrm{R} \subset \Sigma^*$? The answer to this question is also no, since the real number we denote with π, which is the ratio of a circle's circumference to its diameter, cannot be represented with any string of symbols from Σ of finite length. Indeed, to represent π with a finite number of symbols we are forced to use symbols that are not from Σ. So if we wish to have a set based on Σ that we can claim is equal to the set of real numbers, we must allow strings of digits that are infinite. This results from the fact that the real numbers, with the standard metric $d(x,y) = |x - y|$, forms a complete metric space.

So the next question is whether our (Σ^*, σ) with σ given by 4.3 is a complete metric space. To show that the answer is no, consider the sequence from Σ^* where $\Sigma = \{0, 1\}$,

given by

$$x_i \quad \underbrace{01\text{01}...0101}_{\text{i symbols}} \qquad\qquad i \geq 1.$$

This is a Cauchy sequence with respect to the metric σ, but the limit of this sequence, which is an infinite string of alternating 0's and 1's, is not an element of Σ^* since it's not a finite string. However, a theorem from topology states that given any metric space we can form a unique complete metric space from it by adding in those elements which are the limit points of each Cauchy sequence (256). The complete metric space formed in this manner is called the *completion* of the given metric space. We now define the completion of our (Σ^*, σ) metric space.

Definition IV.7 *Given a finite set Σ, and the metric σ defined by 4.3, the set Σ^∞ contains only those elements such that*

$$(\Sigma^\infty, \sigma)$$

is the completion of the metric space (Σ^*, σ).

Based on Definition IV.7, the set Σ^∞ contains all of the elements of Σ^*, plus all of the limit points of all of the Cauchy sequences that can be formed from the elements of Σ^*. If we arbitrarily select any infinite string that can be formed from the symbols of Σ,

$$a_1 a_2 a_3 ... \qquad\qquad a_i \in \Sigma \qquad 1 \leq i,$$

then we can form a Cauchy sequence from this string,

$$\{x_n\} = \{a_1, a_1 a_2, a_1 a_2 a_3, ...\},$$

whose limit point is the original infinite string. This means that Σ^∞ contains all finite strings of symbols from Σ, plus all infinite strings of such symbols. Consequently, Σ^∞ can be shown to contain an uncountable number of elements (305), which, since there are only a countable number of computable sequences, implies the truth of the following theorem.

Theorem IV.8 *Given a finite set Σ, and the metric σ given by 4.3, there exists Cauchy sequences of elements of Σ^* that cannot be generated by any Turing machine.*

Proof: Since there are only a countable number of finite length strings, there must be an uncountable number of infinite length strings, each of which is unique. Therefore, since each infinite string is the limit point for at least one Cauchy sequence, and the limit point for a given convergent sequence is unique, there are an uncountable number of distinct Cauchy sequences, whereas there can only be a countable number of Turing computable Cauchy sequences (334). ■

That there are only a countable number of Turing computable Cauchy sequences follows from a theorem in Turing's 1937 paper on computability, which states "The limit of a computably convergent sequence is computable." (334)

Although Σ^∞ contains the infinite strings needed to complete the metric space (Σ^*, σ), we do not have a need for an uncountable number of such strings. This thesis deals with computation, and so instead of all infinite strings we only need those infinite strings that are the limit points of Cauchy sequences that can be generated by Turing machines. We call such sequences *Turing computable*. Accordingly, we define the set needed to form a pseudo complete metric space, based on only those Cauchy sequences that are Turing computable.

Definition IV.9 *Given a finite set Σ, and the metric σ defined by 4.3, the set denoted by Σ^C contains only the following elements:*

1. *All of the elements of Σ^*.*
2. *The limit points of all Turing computable Cauchy sequences that can be formed from the elements of Σ^*.*

As a consequence of this definition we have exactly our *Turing computable complete metric space.*

Definition IV.10 *Given a finite set* Σ, *the set* Σ^C *defined by Definition IV.9, and the metric* σ *defined by 4.3, then the two-tuple*

$$(\Sigma^C, \sigma)$$

forms a **Turing computable complete metric space.**

The concept behind the Turing computable complete metric space is that for any Cauchy sequence that can be generated by a Turing machine, the limit point for that sequence is an element of the space. This means that the metric space is 'effectively' complete with respect to any analysis that requires sequences of strings to be generated by Turing machines.

In one of the landmark papers dealing with the mathematical properties of computation (311), Scott claimed that the set Σ^∞ satisfies the following property:

$$\Sigma^\infty = \Sigma \cup (\Sigma^\infty)^*.$$

Unfortunately, the paper did not present a rigorous proof, and also did not define what is meant by the concatenation of two infinite strings. However, if we do define concatenation of infinite strings in terms of Turing machines (the actual definition does not matter), then we can prove the following result.

Theorem IV.11 *Given a finite set* Σ, *the set* Σ^C *defined by Definition IV.9 satisfies the following equality:*

$$\Sigma^C = \Sigma \cup (\Sigma^C)^*.$$

Proof: The equality holds by definition for all possible finite strings of symbols from Σ, so only infinite strings need be considered. For any given infinite string x, where

$$x \in \Sigma \cup (\Sigma^C)^*$$

another infinite string can be formed by concatenation with either a finite or an infinite string, and any such concatenation is Turing computable. The result of any

countable number of such concatenations can be computed by a Turing machine, so that any strings generated in this manner are Turing computable (note the star closure is a countable number of such string concatenations). Additionally, given any computable infinite string x,

$$x \in \Sigma^C$$

a computable Cauchy sequence can be formed using the technique described in the paragraph following Definition IV.7. This means that the operations on the right hand side of the equality can only generate additional elements of Σ^C (as long as string concatenation is defined for any combination of finite and/or infinite strings). Thus

$$(\Sigma \cup (\Sigma^C)^*) \subset \Sigma^C$$

Since the right hand side of the equality includes the star closure of the left hand side, then

$$\Sigma^C \subset (\Sigma \cup (\Sigma^C)^*)$$

since any element of the set Σ^C is also an element of the set on the right hand side of the equality. This proves the equality. ■

Corollary IV.12 *The set Σ^* can be represented as a theory (see Section 3.3).*

Proof: Since any finite Σ can be represented with the two element set, the proof assumes that

$$\Sigma = \{a, b\}.$$

The axioms object consists of three objects representing the symbols a, b, and Λ (the empty word), while the morphisms correspond to:

$$\forall x[x \in \Sigma^* \Longrightarrow \alpha x \in \Sigma^*]$$

where

$$\alpha \in \{\Lambda, a, b\}$$

■

Corollary IV.13 *The set Σ^C can be represented as a theory.*

Proof: Take the theory representing Σ^* and add all of the limit points of the Turing computable Cauchy sequences to the axioms object, along with the rule of inference that represents infinite with infinite (and infinite with finite) string concatenation. ■

That neither of the theories representing Σ^* or Σ^C are complete theories follows from the analogy between these sets and the real numbers. If Σ is the set of decimal digits and the period, then there exists real numbers from the interval $[0, 1]$ that cannot be represented with either elements of Σ^* or Σ^C.

Corollary IV.14 *Given that*

$$\Sigma = \{., 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

then

$$\mathrm{C} \subset \Sigma^C$$

where C *is the set of Turing computable numbers (see Appendix B).*

Proof: Since a Turing computable number is a string of decimal digits that has the property that a Turing machine can produce any finite number of consecutive digits in finite time (246), then the limit point of a Turing computable Cauchy sequence of real numbers is exactly a Turing computable number. This follows from considering each successive element in the Cauchy sequence as the next iteration of the digits of the computable number. ■

In some sense (excluding strings that are not numbers), the set Σ^C corresponds to the set C of computable numbers. Thus the set Σ^* corresponds to the set of all numbers with finite decimal expansion, and the set Σ^∞ corresponds to the set R of all the real numbers.

Since the set of reals with finite decimal expansion is dense in the set of reals (standard absolute value metric), the statement that Σ^* is 'dense' in the set Σ^∞ has some meaning. Also note that since 'computer science' addresses the science of what is possible within the realm of Turing machines, then Σ^C is sufficient for computer science (as opposed to Σ^∞), since any string that can by computed by a Turing machine will be an element of Σ^C.

Note that the definition of Σ^C is based on the topological concepts of a complete metric space. However, Theorem IV.11 shows that Σ^C also satisfies a strictly set-theoretic specification. So this theorem states that there exists a relationship between the concept of computation and that of topology. The next section addresses this issue in more detail.

4.2 A Complete Metric Space Based on Finite Automata

Extending the groundwork laid in the last section, this section further demonstrates the relationship between computational and topological concepts. In particular, this section shows that the concept of computational power can be analyzed using the topology of complete metric spaces. This analysis is based on completing a metric space whose elements are finite automata, such that the completion includes elements that are not finite automata. This section considers these elements from a computational power point of view, and not as 'infinite automata', which are closely related to temporal predicates (191) (and not to be confused with infinite strings associated with finite automata (235)).

The basic model of computation is the finite automaton, a computing machine based on discrete states and the transitions between states (see the Introduction for a definition of state). The finite automaton is at the bottom of a hierarchy of computing machines with ever increasing abilities or *power* (138, 217). Although the finite automaton is at the bottom, it is the only type of machine in the hierarchy that can be physically realised, in that all real computers are finite automata (with the possible exception of Man) (249). In addition to the computing machines consituting this hierarchy, many other computational models are based on the finite automaton (5, 168, 360), and this section gives an example of one of these models, called the *computation system* (238).

There are two basic types of finite automaton, the *nondeterministic* and the *deterministic*. Although there is no difference in the computational power between the two, this section gives the definitions for both.

Definition IV.15 *A* **nondeterministic finite automaton** *is a five-tuple* (S, A, R, s, E), *where*

$$S = \{s_1, ..., s_n\}$$

is a finite set whose elements are called the **states**.

$$A = \{a_1, ..., a_m\}$$

is a finite set whose elements are called the inputs, *and*

$$R = \{r_1, ..., r_m\}$$

is a finite set whose elements are relations over S, that is

$$\forall r_i [r_i \in R \Longrightarrow r_i \subset S \times S]$$

and

$$s \in S$$

is called the initial state, *and*

$$E \subset S$$

is a finite set whose elements are called the final *or* accepting states.

Since for any nondeterministic automaton (S, A, R, s, E) there are exactly the same number (both A and R have m elements) of relations in R as inputs in A, the standard practice is to label each relation with a symbol from A. One interpretation of a nondeterministic finite automaton (S, A, R, s, E) is that of a directed graph, with the nodes labeled with the elements of S, and the arcs labeled with the elements of A, such that each relation in R corresponds to exactly one element from A (one or more of the relations from R may be empty sets). Given the relation that corresponds to the symbol α (where $\alpha \in A$) which contains zero or more ordered pairs, then for each pair (s_i, s_j) that is an element of this relation there is an arc labeled with α whose head is s_i and whose tail is s_j. With this interpretation, we can consider paths through the directed graph that start with the initial state s. Inputs to the graph are *words*, which are strings of symbols from A, and as each successive (starting with the leftmost symbol) symbol from a word is read, we traverse the arc labeled with that symbol whose head is the current node to the node that is the tail of the arc. After the last symbol in the word is processed in this manner, and if the final node is an element from the set E, then that word is said to be *accepted* by the automaton, otherwise the word is *rejected*. By repeating this process for all possible words (of finite length) from A^*, we can construct the set of all words that are accepted by a

given nondeterministic finite automaton. If we denote the automaton with the symbol T, then $accept(T)$ denotes this set of all words from A^* that are accepted by the automaton, while $reject(T)$ denotes the set of all words from A^* that are rejected by T. Note that

$$accept(T) \cup reject(T) = \Sigma^*.$$

Whereas the nondeterministic automaton has transistions between states based on relations, the *deterministic finite automaton* has transitions based upon total functions.

Definition IV.16 *A* **deterministic finite automaton** (S, A, R, s, E) *is defined such that the sets* S, A, E *and the element* s $(s \in S)$ *are exactly the same as in Definition IV.15; whereas the set* R *contains total functions, that is for* $m \geq 1$

$$R = \{f_1, ..., f_m\}$$

such that

$$\forall f_i [f_i \in R \Longrightarrow f_i : S \rightarrow S]$$

where each f_i *is defined for all of the elements of the domain* S *(138).*

Just as with the nondeterministic finite automaton, the number of elements in the set R equals the number of elements in the set A. The directed graph interpetation is also the same as for nondeterministic finite automata, with one exception. For the deterministic automaton the arcs are determined by functions, not relations. Given a node s_j, the arc that is labeled with α (where $\alpha \in A$) whose head is s_j, has as its tail the node s_k, where

$$f_\alpha(s_j) = s_k.$$

A consequence of this difference in the elements of the set R between the nondeterministic and the deterministic finite automata is that for each node in the directed graph representation of a deterministic finite automaton, there is exactly one arc labeled with each symbol from A whose head is that node, while for a nondeterministic finite automaton there can be zero or more arcs labeled with a symbol from A whose head is a given node.

Two nondeterministic (or deterministic) finite automata that have identical sets of input symbols are said to be equivalent if they both accept exactly the same set of *words*. Note that if two automata accept the same set of words, then they also reject the same set of words, since for any automaton T we have that $reject(T) = S^* - accept(T)$, where S is the set of input symbols. Also, since it has been shown that any nondeterministic finite automaton is equivalent to a deterministic finite automaton, we refer to both deterministic and nondeterministic finite automata as simply finite automata (217). If we define two finite automata to be equal if and only if (iff) their directed graph representations differ only in the symbols chosen for the arcs and nodes, then this equivalence relation between finite automata is not equality, since two finite automata can be equivalent but have different numbers of states (138). We can partition the class of all finite automata that share the same set of input symbols into equivalence partitions, such that all automata in a partition accept the same set of words. Each equivalence partition can be denoted by the set (which is described in a finite representation using a *regular expression* (217)) of all words accepted by the finite automata in that partition.

Now we present an example of a computational model based on the finite automaton, which is called a *computation system* (238).

Definition IV.17 *A* computation system *is the ordered two-tuple* $((\Sigma, D, x), F)$, *where*

$$\Sigma = \{\alpha_1, ..., \alpha_n\}$$

is a finite set, D is a countable set, x is an element of D, and F is a total function whose domain is Σ^* and whose codomain is the set of all partial functions with domain D and codomain D, such that

$$\forall\alpha[\alpha \in \Sigma^* \Longrightarrow F(\alpha) = \bar{\alpha}]$$

where

$$\bar{\alpha} : D \rightharpoonup D$$

and function composition is given by

$$\forall \beta, \gamma, y \left[(\beta \in \Sigma^* \wedge \gamma \in \Sigma^* \wedge y \in D) \Longrightarrow \overline{\beta\gamma}(y) = \bar{\beta}(\bar{\gamma}(y)) \right].$$

If Λ *denotes that unique element of* Σ^* *that is the concatentation identity, then*

$$F(\Lambda) = \bar{\lambda}$$

where $\bar{\lambda}$ *denotes the identity function on the elements of* D*, that is*

$$\forall y [y \in D \Longrightarrow \bar{\lambda}(y) = y]$$

such that λ *is not an element of either* Σ *or* D*.*

A computation system consists of the triple (Σ, D, x) along with the function F that maps elements of Σ^* into partial functions from the elements of D into the elements of D. Additionally, concatenation of elements of Σ into strings of symbols (which are elements of Σ^*), corresponds to composition of the functions represented by the individual symbols. As an example of a computation system, consider $((\Sigma, D, x), F)$ where

$$\begin{aligned} \Sigma &= \{a, b\} \\ D &= \{0, 1\} \end{aligned}$$

and

$$\begin{aligned} F(a) = \bar{a} &= \{(0,0), (1,1)\} \\ F(b) = \bar{b} &= \{(0,1), (1,0)\} \end{aligned}$$

with the functions $F(a)$ and $F(b)$ denoted using the two tuple nomenclature from Definition III.10. With the notation of Definition IV.17, we have

$$\bar{a}(0) = 0 \text{ and } \bar{a}(1) = 1$$

with

$$\bar{b}(y) = 1 - y \qquad y \in D.$$

Although F is defined for all of its domain Σ^*, the definition of function composition in Definition IV.17 implies that we only need specify F evaluated for the elements of Σ.

In Definition IV.5 we define Σ^* so that it includes the distinct element Λ which is not an element of Σ (and for any computation system $((\Sigma, D, x), F)$, Λ is not an element of D). This Λ represents the concept of the *empty word* (217). Definition IV.5 states that this empty word serves as an identity when composing strings of elements from Σ. Since the empty word is not an element of Σ, and there is not necessarily an identity function on the elements of D that corresponds to an element of Σ, our definition of a computation system includes the identity function $\bar{\lambda}$ which corresponds to the empty word Λ.

Since for any computation system $((\Sigma, D, x), F)$ there exists a collection of partial functions that maps elements of D to elements of D, one for each element of Σ, we can collect these functions into a set, say M. For example, consider the computation system $((\Sigma, D, x), F)$ where

$$\Sigma = \{a, b\}$$
$$D = \{1, 2, 3\}$$
$$x = 1$$
$$F = \{(\Lambda, \bar{\lambda}), (a, \bar{a}), (b, \bar{b}), \ldots\}$$

such that

$$\bar{a} = \{(1, 2), (2, 3)\}$$
$$\bar{b} = \{(1, 3), (2, 1)\}.$$

Then we can form

$$M = \{\bar{\lambda}, \bar{a}, \bar{b}\}$$

such that

$$\bar{a}(1) = 2 \quad \text{and} \quad \bar{a}(2) = 3$$

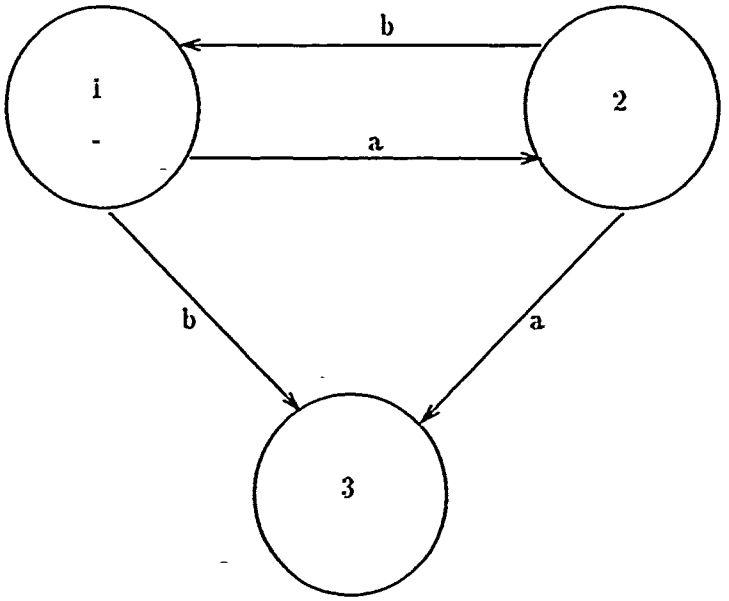

Figure 4.1. Directed Graph Representation of Example Computation System

$$\bar{b}(1) = 3 \quad \text{and} \quad \bar{b}(2) = 1.$$

In this example both $\bar{a}$ and $\bar{b}$ are partial functions, since neither $\bar{a}(3)$ nor $\bar{b}(3)$ is defined. Also note that F can be completely specified with just those elements that correspond to the symbols from Σ, along with $(\Lambda, \bar{\lambda})$.

We can represent this sample computation system using a directed graph. Figure 4.1 shows such a graph, where the elements of D are the nodes, and the elements of M are the directed arcs. Each arc is labeled with the element from Σ that corresponds to the element of M that the arc represents. So the arc whose head is node 2 and whose tail is node 1, and is labeled with the symbol b, represents the function evaluation $\bar{b}(2) = 1$. Just as is the common practice with finite automata, we do not include the arcs labeled with Λ that would have the same node for their head and tail, one such arc per node (138). This directed graph representation suggests that a computation system corresponds to an automaton (101). Indeed, in Figure 4.1 we have labeled node 1 with a '-' to signify that it is the node x in the computation system $((\Sigma, D, x), F)$, just as we label the initial node in the graphical representation of a finite automaton. We can call the nodes of the graph the

states, with the arcs representing the transitions. If we choose the state x to be the *initial* state, and choose zero or more states collected in a set E to be the *final* states, then our computation system augmented with E would satisfy our definition of a nondeterministic finite automaton (Definition IV.15). Since E can be empty, then any computation system can be considered as a finite automaton without any accepting states.

Now let us show that a finite automaton is a special instance of a computation system. Recall that we labeled each function in M with a symbol that corresponded to a symbol from Σ in our sample computation system $((\Sigma, D, x), F)$ (along with the implicit set M). In our example computation system, the set D corresponds to the set S of states, the set Σ corresponds to the set A of inputs, and the set M corresponds to the set R of relations. The common interpretation (101, 138, 217) for a finite automaton is that of moving from one state to another on receiving one of the input symbols. So for our example (see Figure 4.1), if we are currently in state 2 and receive an input of a, then we move to state 3. These moves between states are called transitions.

Although our definition of finite automaton requires the existence of a set whose elements are relations, our set M of partial functions satisfies this requirement, because as we saw in Section 3.2 the class of all functions (which includes partial functions) is a species of the class of all relations. So any partial function is also a relation, which is also true for the set-based definitions of partial functions and relations that the definition of finite automaton is based upon (138). Since a computation system $((\Sigma, D, x), F)$ requires that the set Σ be finite, then this set M of partial functions will also be finite, since there is only one partial function corresponding to each element of Σ. So Σ satisfies the requirements of the set A, and M satisfies those for the set R in our definition of a finite automaton (S, A, R, s, E). If we take the element x to be the initial state s, then except for the set E the only difference between the requirements imposed on the sets for a computation system and those for a finite automaton is the cardinality of the set D from the computation system, which corresponds to the set S for the finite automaton. This means that all computation systems whose (possibly countably infinite) set D is finite, and that have a set of states (possibly empty) considered to be final or accepting states (so as to make up the set E), also satisfy Definition IV.16 of finite automata. Note that for any

computation system that doesn't have a candidate set of final or accepting states for the set E, the set E can be considered as empty.

So it is established that the class of all computation systems whose set of states is finite is a species of the class of all finite automata. But the objective of this section is to further establish a link between computational power and the topology of complete metric spaces. So the next step is to choose the appropriate metric space to complete, based on a model of known computational power, the finite automaton. This implies that we need a metric for finite automata. And the metric that we choose will depend upon a bijection between finite automata and *binary automaton trees.*

Definition IV.18 *A* **binary automaton tree** *is a four-tuple* (N, P, n, M)*, where* N *is a finite set whose elements are called the* **nodes**, P *is a set whose two elements are total functions with domain* N *and codomain* N, n *is an element of* N *called the* **root node**, *and* M *is a subset of* N *whose elements are called the* **accepting nodes**.

Since we wish to define a bijection between finite automata and binary automata trees, we first observe the similarities between the two. A finite automaton is a five-tuple, say (S, A, R, s, E). If A contains just two elements, then R contains just two total functions whose domain and codomain is the set S. And since s is an element of S, and E is a subset of S, then we have that any finite automaton (S, A, R, s, E) with just two inputs (the set A contains elements called the inputs), leads to the four-tuple (S, R, s, E) that satisfies the definition of a binary automaton tree. Since any finite input symbol set can be represented by just two distinct symbols (138), we will drop the caveat that the set A only contain two symbols, so that what follows applies to any finite automaton.

For example, if we are given the finite automaton (S, A, R, s, E) where

$$S = \{1, 2, 3\}$$
$$A = \{a, b\}$$
$$R = \{r_a, r_b\}$$
$$s = 1$$
$$E = \{3\}$$

such that

$$r_a = \{(1,2),(2,2),(3,2)\}$$
$$r_b = \{(1,1),(2,3),(3,3)\}$$

then we can form the binary automaton tree (S, R, s, E). Figure 4.2 shows the standard graphical representation of the finite automaton (S, A, R, s, E), while Figure 4.3 graphically depicts the binary automaton tree (S, R, s, E) in a tree-like manner, hence the term 'binary automaton tree'. In Figure 4.2 we label node 1 with a '-' to signify that this is the initial node, and node 3 with a '+' because it is an accepting node. In Figure 4.3 we label node 1 with a '-' since it is the root node of the tree, and node 3 (which appears twice) with a '+' to show it's an accepting node. Our four-tuple representation for a binary automaton tree does not include the set of input symbols from the finite automaton because if we did there would only be two symbols in the set. What the two symbols actually are does not matter, so we can use any two distinct symbols to label the arcs of the binary automaton tree's graphical representation, where each symbol corresponds to one of the two functions that make up the set R. Given a graphical representation of a finite automaton, use the following technique to construct the graphical representation of the corresponding binary automaton tree. For an example, refer to Figures 4.2 and 4.3.

1. Start with the initial node from the finite automaton and make it the root node of the binary automaton tree.
2. Given any *parent node* p of the binary automaton tree, we draw an arc labeled with the input symbol i (We could use any two distinct symbols to differentiate between the arcs) whose head is p and whose tail is the *child node* c, if the arc i had the node p as its head and the node c as its tail in the graphical representation of the finite automaton, and if the parent node has not previously appeared in the tree.
3. A parent node that has already appeared in the binary automaton tree is a *leaf node* and has no child nodes, that is there are no arcs whose head is this node.
4. The binary automaton tree is constructed in this manner starting with the root node, and whenever there is more than one child node that can be drawn next the leftmost

node is drawn first, where all of the leftmost child nodes are the the tail nodes of arcs labeled with the same symbol (i.e. each symbol corresponds to one of the two total functions that define the arcs).

The last item in our constructive description guarantees that given any finite automaton we can construct a unique representation of a corresponding binary automaton tree, since any possible arbitrary choice has been removed.

Conversely, we can take any given binary automaton tree and graphically construct the representation for a unique finite automaton using the following procedure.

1. Start with the root node of the binary automaton tree and make it the initial node of the finite automaton.
2. Given any arc labeled with the symbol i from the binary automaton tree, add the parent node p and the child node c from the tree to the finite automaton if they are not already present, and draw an arc labeled with the input symbol i whose head is p and whose tail is c.
3. The finite automaton is constructed in this manner starting with the arcs that connect the root node to its two child nodes, and then repeating the second step once for each arc in the binary automaton tree.

This constructive technique ensures that there will be the same number of arcs in the graphical representation of the finite automaton as in the binary automaton tree, which is the desired result. Also note that given an arbitrary graphical representation of a finite automaton, we can generate a unique five-tuple representation for the automaton, just as for any five-tuple representation we can construct a unique graphical representation (138). This means that our use of 'finite automaton' can mean either the five-tuple or the graphical representation.

Consequently, given the graphical representation of either a finite automaton or a binary automaton tree, we can uniquely construct the other, and given either graphical representation, we can generate both the four-tuple representation of the binary automaton tree and the five-tuple representation of the finite automaton. So we see that for any given

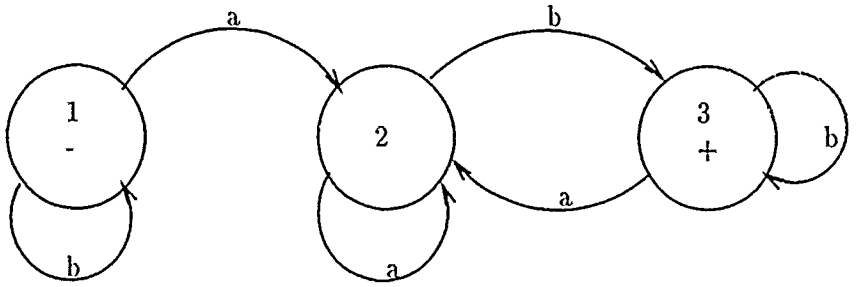

Figure 4.2. Graphical Representation of Finite Automaton (S, A, R, s, E)

finite automaton, we can construct the unique binary automaton tree that represents it, and for any binary automaton tree there exists a unique corresponding finite automaton (with the caveat that in generating the finite automaton given a four-tuple representation of the binary automaton tree we have an arbitrary choice as to the two input symbols, but we can force uniqueness by simply specifying that these two symbols will always be the unique symbols a and b which are not members of any of the other sets needed for the representations). Thus the motivation for the existence of a bijective function between finite automata and binary automaton trees, and for the next theorem, which requires the following definition.

Definition IV.19 *Two binary automaton trees* (T, P, t, U) *and* (V, Q, v, W) *are* equal, *if and only if, there exists a bijective total function* ϕ,

$$\phi : T \to V$$

such that

$$\forall x, y, p, q[(x \in T \wedge y \in V \wedge p \in P \wedge q \in Q) \Longrightarrow (q(y) = \phi(p(x)) \Longleftrightarrow y = \phi(x))]$$

$$\phi(t) = v$$

$$\forall u, w \left[(u \in U \Longrightarrow \phi(u) \in W) \wedge (w \in W \Longrightarrow \phi^{-1}(w) \in U)\right];$$

Theorem IV.20 *There exists a total bijective function whose domain is the set of all finite automata and whose codomain is the set of all binary automaton trees. There also*

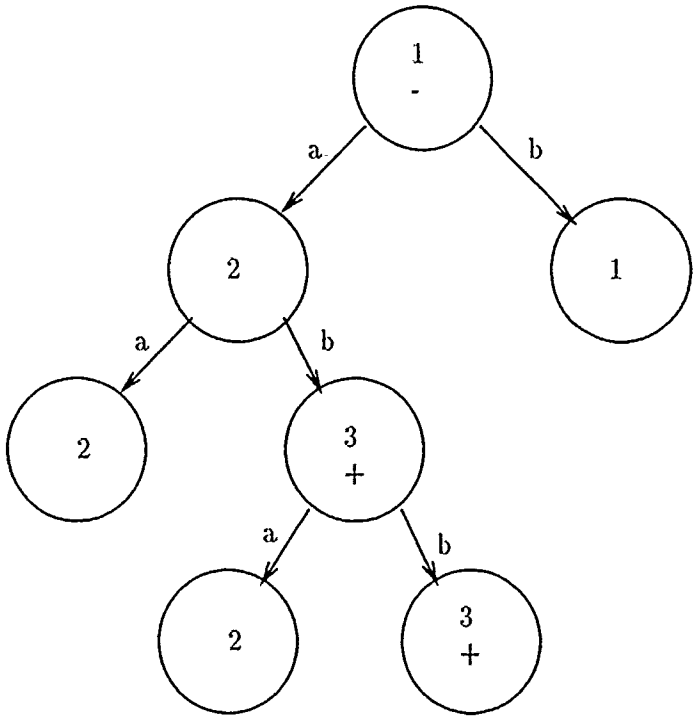

Figure 4.3. Graphical Representation of Binary Automaton Tree (S, R, s, E)

exists a total bijective function whose domain is the set of all binary automaton trees and whose codomain is a specified set of graphical representations of binary automaton trees.

Proof: The requirement for binary automaton tree equality states that two trees are equal iff their graphical representations based on the proceeding construction technique are identical except for a relabeling of the nodes and/or arcs, that is they are isomorphic with respect to the initial and accepting nodes, and the functions that are represented by the connecting arcs.

The preceeding construction demonstrates the existence of such a bijective function for the set of all finite automata with only two input symbols. Since any finite set of input symbols can be represented with just two distinct symbols, the existence claim holds for all finite automata. ■

The *specified set* in this theorem refers to the set 'specified' by the constructive technique described above. Note that if we had not designed our constructive technique so as to be deterministic then the second part of this theorem would not have been true.

If we extend the definition of the function *accept*, so that its domain includes all binary automaton trees, with the enumeration of the accepted set of words for a binary automaton tree performed just as for a finite automaton, then we have the following corollary to this theorem. (Note that we use 'enumeration' in the sense of Turing-enumerable)

Corollary IV.21 *There exists a bijective function Φ, whose domain is the set of all finite automata and whose codomain is the set of all binary automaton trees, such that*

$$accept(m) = accept(\Phi(m))$$

for any finite automaton m, and

$$accept(b) = accept(\Phi^{-1}(b))$$

for any binary automaton tree b.

Proof: The function Φ is just the bijection from Theorem IV.20. Given a finite automaton m, and any arbitrary nonempty word w, such that

$$w \in accept(m)$$

we can decompose w into a sequence of individual symbols from the input alphabet of m, that is

$$w = s_1 s_2 ... s_n \qquad n \geq 1.$$

Both m and $\Phi(m)$ have an initial node that processing of w starts in, and for each symbol s_i, the transition from one state to another in m has a corresponding transition in $\Phi(m)$, such that a given input symbol leads to an accepting state for m iff it leads to an accepting state for $\Phi(m)$, which implies

$$accept(m) \subset accept(\Phi(m)).$$

Since Φ is total and bijective, then for any binary automaton tree b, there exists a finite automaton $\hat{m}$, such that

$$b = \Phi(\hat{m}) \qquad \hat{m} = \Phi^{-1}(b).$$

This, along with the above reasoning, implies that

$$accept(\Phi^{-1}(b)) \subset accept(b)$$

for any binary automaton tree b. These subset containments hold for $accept(m)$ nonempty, and such that the empty word is not an element of $accept(m)$. But if m accepts the empty word then the initial node of m is accepting, which will also be true for $\Phi(m)$, thus $\Phi(m)$ accepts the empty word. The converse is also true, if $\Phi(m)$ accepts the empty word, so does m. If $accept(m) = \emptyset$ then m has no accepting states, and neither does $\Phi(m)$. so that $accept(\Phi(m)) = \emptyset$; the converse is also true.

By repeating the above argument for any given binary automaton tree b, and for any word w, such that

$$w \in accept(b)$$

the implications yield

$$accept(\Phi(m)) \subset accept(m)$$
$$accept(b) \subset accept(\Phi^{-1}(b))$$

which establishes the equalities. ■

Figure 4.4 shows two such isomorphic trees that meet the definition for binary automaton tree equality from Theorem IV.20, and Figure 4.5 graphically depicts the relationships specified by this definition of equality. For Figure 4.4 the bijective function ϕ is given by

$$\phi(1) = a \qquad \phi(2) = b.$$

Note that the definition for equality given in Theorem IV.20 does not address the labels on the arcs, since the placement of the arcs suffices to identify which function they represent. In Figure 4.4 however, we have labeled the arcs so that the functions represented by the 'a' and the 'b' correspond to the functions represented by the '0' and the '1' respectively.

Before we proceed we need to define the concept of how far away from the root node is any given node in our graphical representation of a binary automaton tree, which is called the *level* of the node (46). Also, because of the second part of Theorem IV.20, we no longer need to differentiate between a binary automaton tree and its graphical representation using our construction method. So whenever we use the term 'binary automaton tree', we mean either the four-tuple designation or the graphical representation.

Definition IV.22 *Given a binary automaton tree, the* level *of any node within that tree is given by:*

1. *The level of the root node is 1.*

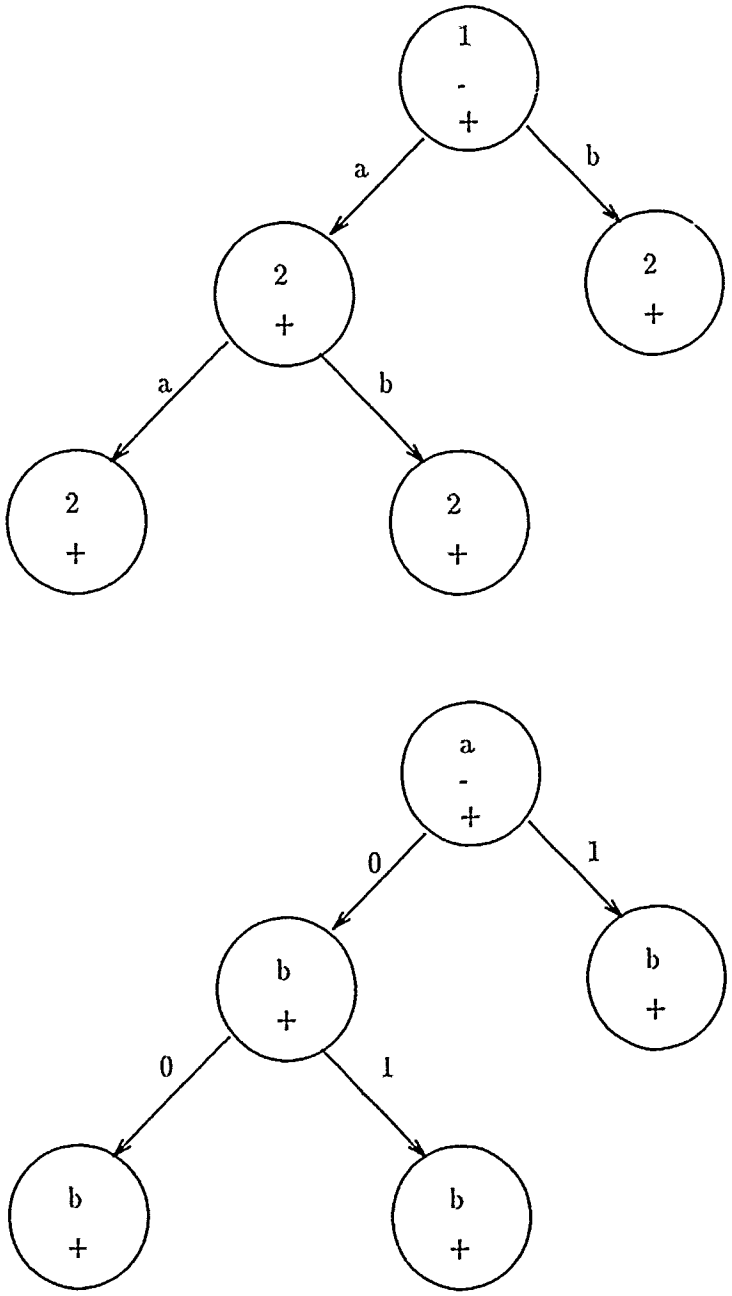

Figure 4.4. Two Equal Binary Automaton Trees

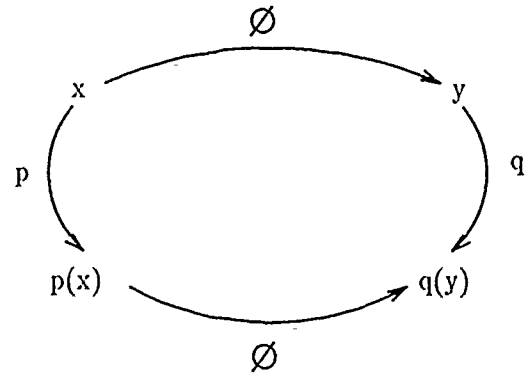

Figure 4.5. Relationships Specified by Binary Automaton Tree Equality

2. *Given that the level of a parent node is j, then the level of any child node of that parent node is $j+1$.*

Definition IV.23 *A* **level k restricted binary automaton tree** *contains only nodes whose level is less than or equal to k.*

Definition IV.24 *Given a binary automaton tree T, construct the* **level k restriction** *of T, denoted by T_k, by deleting all nodes of level $k+1$ or greater, and any arcs whose heads or tails are these nodes.*

Next we give a metric for binary automaton trees, which will serve as the basis for our metric for finite automata. Just as the metric σ given by 4.3 of Section 4.1 measured how 'far into' two strings the equality of their prefixes would hold, our binary automaton tree metric measures how far into two trees the equality of their level k restrictions will hold. Note that level k restriction equalities are just those given by Theorem IV.20 for any binary automaton tree. Given any two binary automaton trees T and V, we define a function d_{tr} that maps two tuples of binary automaton trees into the nonnegative real numbers by

$$d_{tr}(T,V) = \begin{cases} 0 & \text{if } T = V \\ \inf\{1, 1/2^k | T_k = V_k\} & \text{otherwise} \end{cases} . \qquad (4.4)$$

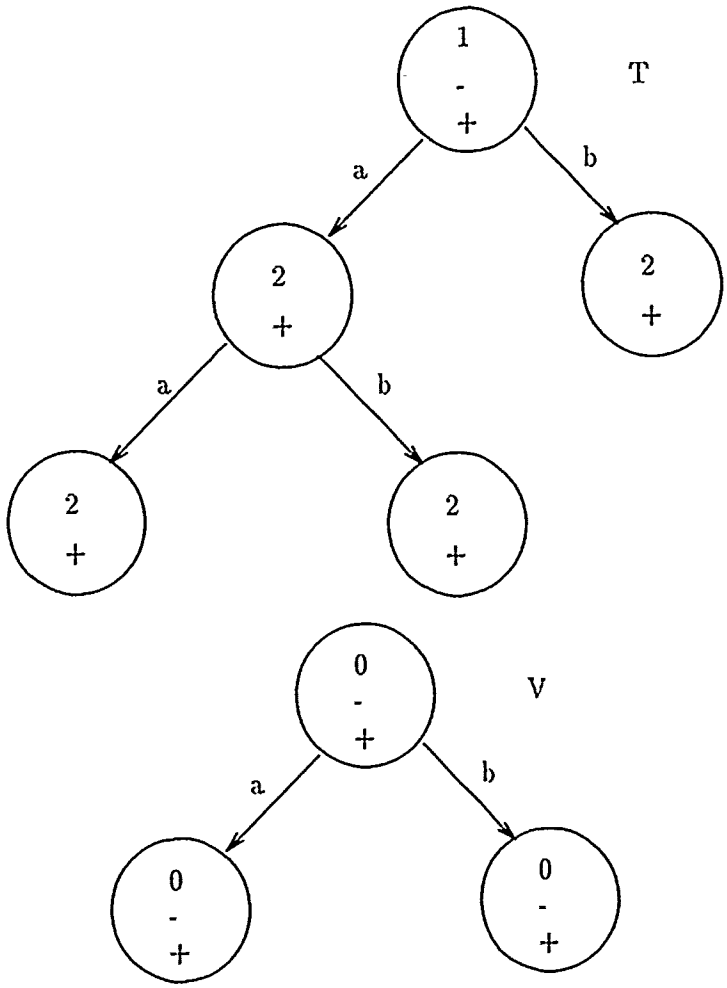

Figure 4.6. Binary Automaton Trees T and V

If two binary automaton trees T and V do not have equal level 1 restrictions, then we have that

$$\begin{aligned} d_{tr}(T,V) &= \inf\{1\} \\ &= 1 \end{aligned}$$

since there is no value of k for which $T_k = V_k$.

Consider the two binary automaton trees T and V depicted in Figure 4.6. If we consider their level 1 restrictions, then we have that $T_1 = V_1$, because there exists a bi-

jection ϕ such that $\phi(1) = 0$, and both states 1 and 0 are initial and accepting states in the corresponding restricted binary automaton trees. But when we consider the level 2 restrictions we find that there does not exist any such bijection that satisfies the requirement of Theorem IV.20 for (restricted) tree equality. Consequently, we have $T_2 \neq V_2$, and also that $T_3 \neq V_3$, so that

$$\begin{aligned} d_{tr}(T, V) &= \inf\{1, 1/2\} \\ &= 1/2 \end{aligned}$$

The proof that d_{tr} given by 4.4 is a metric for the set of all binary automaton trees follows the proof that σ (see 4.3 in Section 4.1) is a metric for the set Σ^* (43). (Note that Degano and Montanari show how the concept behind this metric can be extended to more complicated structures that represent distributed systems (99)) Since Theorem IV.20 states that there is a one-to-one correspondence between finite automata and binary automata trees, then it follows that we can use our metric d_{tr} to define a metric on the set of all finite automata.

Theorem IV.25 *Given the bijection Φ from Corollary IV.21 and the function d_{tr} from 4.4, then the function d_{fa} that maps the set of all finite automata into the nonnegative real numbers given by*

$$d_{fa}(A, B) = d_{tr}(\Phi(A), \Phi(B))$$

for any two finite automata A and B, is a metric for the set of all finite automata.

Proof: The proof that d_{tr} is a metric follows the proof from Section 4.1 for the metric σ on Σ^*. Consider any three finite automata A, B, and C. Since Φ is a total bijection, then

$$d_{tr}(\Phi(A), \Phi(B)) \leq d_{tr}(\Phi(A), \Phi(C)) + d_{tr}(\Phi(C), \Phi(B)) \Longrightarrow d_{fa}(A, B) \leq d_{fa}(A, C) + d_{fa}(C, B).$$

Since d_{tr} is a metric, then this implication states that d_{fa} satisfies the triangle inequality requirement of a metric (see Definition IV.1). The constructive technique

used to prove Theorem IV.20 provides the bijection Φ between finite automata and binary automaton trees such that

$$(\Phi(A) = \Phi(B)) \Longleftrightarrow (A = B)$$

which implies that since d_{tr} satisfies the strictly positive requirement for a metric, then d_{fa} also satisfies this requirement. Further,

$$d_{tr}(\Phi(A), \Phi(B)) = d_{tr}(\Phi(B), \Phi(A)) \Longrightarrow d_{fa}(A, B) = d_{fa}(B, A)$$

so that d_{fa} satisfies the symmetry requirement; therefore d_{fa} satisfies all the requirements for a metric. ■

One question we can ask is whether the metric d_{fa} accurately represents any intuitive concept of 'closeness' of two finite automata. In Figure 4.6 we have two binary automaton trees T and V such that $d_{tr}(TV) = 1/2$. But both the finite automaton that produced T and the one that produced V accept the same set, which is $\{a + b\}^*$. So in this case the measure of closeness given by the metric does not reflect the concept of computational capability that is represented by the accepted sets. We can partially circumvent this problem though, by modifying the set of all finite automata. Instead of including every possible finite automaton, we can include only the canonical homomorphic images of all possible automata. We do this by collecting all of the finite automata together into equivalence classes based on the accepting sets, so that each equivalence class contains only those automata that accept a given set. Then we generate a canonical homomorphic image of all the automata in the equivalence class, which will be a finite automaton that accepts the same set as all the automata in the class, but has the minimum number of states of any automaton in that equivalence class. It has been shown that this unique automaton exists for each equivalence class, and can be generated from any automaton in the class thru a homomorphism, and will be isomorphic (with respect to relabeling of the nodes and arcs) to the finite automata in the equivalence class that have the fewest number of states (138). This means that we can form a metric space whose set contains each canonical homomorphic image (from each equivalence class), and whose metric is the function d_{fa}.

This metric space now has the desired property that for any two finite automata a and b,

$$d_{fa}(a,b) \neq 0 \Longrightarrow accept(a) \neq accept(b)$$

and for any regular set that would have been accepted by some automaton in the original set of all finite automata, there exists a finite automaton in this new set that accepts the same set. So we have a metric space whose elements have all of the required computational power (i.e can accept all regular sets), and whose metric has to some degree the desired intuitive property.

Theorem IV.26 *Given a finite set Σ, and the function d_{fa} defined in Theorem IV.25, there exists a set M of finite automata such that (M, d_{fa}) forms a metric space, and*

$$\forall a,b\,[(a \in M \wedge b \in M) \Longrightarrow (accept(a) = accept(b) \Longleftrightarrow d_{fa}(a,b) = 0)]\,.$$

Additionally, for every regular set S such that $S \subset \Sigma^*$, there exists a finite automaton $m \in M$ such that

$$accept(m) = S.$$

Proof: For each regular set that can be formed from Σ^*, create one equivalence class of all finite automata that accept that set, and form the set M by choosing from each class one automaton that has the minimum number of states for all the automata in the class. (Disregarding isomorphisms due to relabeling of the nodes and/or arcs this set M would be unique) Since each element of M accepts a different set,

$$accept(a) = accept(b) \Longrightarrow a = b$$
$$a = b \Longrightarrow d_{fa}(a,b) = 0$$

for any $a, b \in M$. Since d_{fa} is a metric for any set of finite automata,

$$d_{fa}(a,b) = 0 \Longleftrightarrow a = b$$
$$a = b \Longrightarrow accept(a) = accept(b)$$

due to the positive definiteness of the metric and the fact that the definition of automaton equality given in Theorem IV.20 implies that any two automata that are equal must be members of the same equivalence class; this is proved by having the Φ of Corollary IV.21 be the isomorphism between equal finite automata defined by the function ϕ of Theorem IV.20, which implies that $accept(m) = accept(\Phi(m))$ whenever $m = \Phi(m)$.

The construction of M ensures that every regular set is accepted by some element of M. ■

The remainder of this section deals with the metric space (M, d_{fa}) given in this theorem.

Now that we have formed a metric space (M, d_{fa}) (see Theorem IV.26) for which we have elements with well defined computational power (the accepting sets), and for which the metric imparts intuitive meaning in terms of closeness with respect to this computational power, our next goal is to answer the following question: Does the completion of this metric space of finite automata yield elements that lie higher up on the Chomsky hierarchy (327, 69, 70) of computing machines? As we shall prove, the answer is *yes*!

Given a finite alphabet $\Sigma = \{a, b\}$, consider the set $S \subset \Sigma^*$ given by

$$S = \{a^n b^n a^n | n \geq 0\}$$

which is a set generated by a type-1 grammar, but not by a type-2 grammar (101). If we fix an upper bound to n, say $n \leq 3$, then we would have the regular set that is accepted by the finite automaton shown in Figure 4.7. To increase clarity in this figure, we have only included the arcs that eventually lead to an accepting state. All of the arcs not shown have the node labeled 'S' as their tail, and from this node there is no string of symbols that will lead to an accepting state. If we had chosen $n = 10$ instead, then the structure of the finite automaton would follow that of Figure 4.7, with just more added states to handle those words with $3 < n \leq 10$. We can continue this process indefinitely, so that for any fixed finite value of n, there exists some finite automaton that can accept the set. So even though there is no finite automaton that can accept the set S, since the value of n is not fixed, what could we say about an *infinite automaton* constructed in the manner

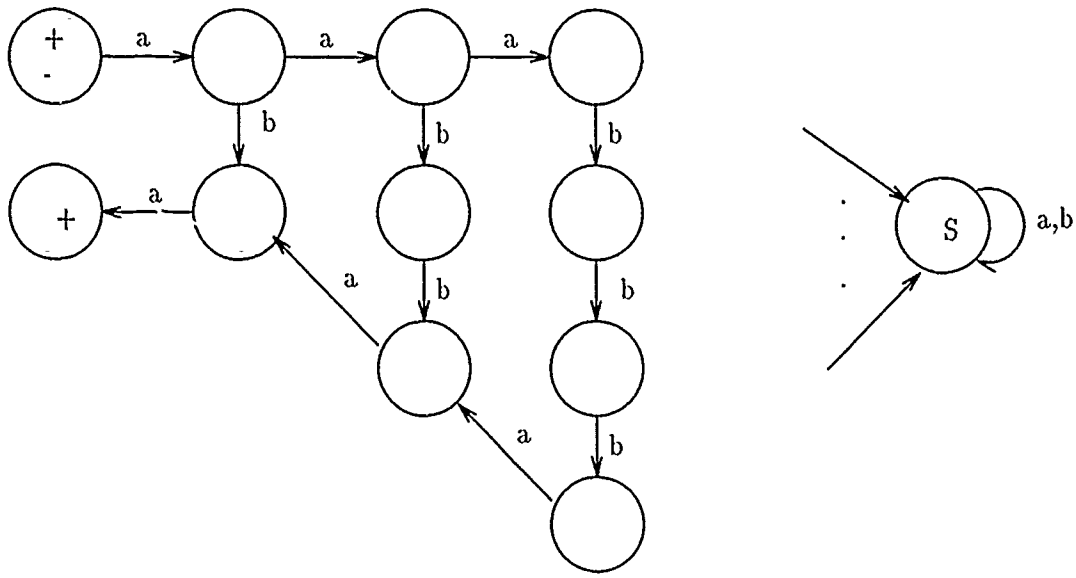

Figure 4.7. Finite Automaton that Accepts $\{a^n b^n a^n | 0 \leq n \leq 3\}$

of Figure 4.7 that would have an infinite number of states? This concept of an infinite automaton corresponds to that of a *transition system* (12). We can't answer this question by constructing such an automaton, but we can answer it using topological concepts and our metric space (M, d_{fa}).

Before stating the primary theorem of this section, we need some intermediate results regarding the relationships between *recursive* sets, the metric space (M, d_{fa}), and binary automaton trees. A recursive set, say R, of words formed from some finite set of symbols Σ, has the property that both the set and its complement $\Sigma^* - R$ are Turing enumerable (101). This means that we could order the words of Σ^* by length using the function len of Section 4.1, along with another function to choose between words of equal length; then starting with the word Λ (the concatenation identity, or empty word) and continuing for each successive word σ of Σ^* (initially $\sigma = \Lambda$), concurrently enumerate both R and $\Sigma^* - R$ so as to decide if σ is an element of R. Since both enumerations will generate any given element of either R or $\Sigma^* - R$ in finite time, the decision of whether or not σ is an element of R can also be done in finite time, since we only have to wait to see which enumeration produces σ. Note that one of the two enumerations will eventually produce σ since either

$\sigma \in R$ or $\sigma \notin R$. Proceeding thru the length-ordered version of Σ^* in this manner allows us to generate a unique length-ordered version of R. Thus we have that the elements of any recursive set can be uniquely ordered by length (190).

Given any finite subset S of a length-ordered recursive set R (R is not generated by any context sensitive grammar),

$$R = \{\sigma_1, \sigma_2, ..., \sigma_n, ...\}$$

we can construct the finite automaton that accepts only those elements of S (327). Extending this concept, we can construct a sequence of finite sets

$$S_1 \subset S_2 \subset S_3 \subset \cdots \subset S_n \subset \cdots$$

such that

$$\forall i \, [i \in \mathrm{N} \Longrightarrow S_i \subset R]$$

with the (proposed) property that

$$\lim_{n \to \infty} S_n = R \tag{4.5}$$

and

$$\forall i \, [i \in \mathrm{N} \Longrightarrow S_i = \{\sigma_1, \sigma_2, ..., \sigma_i\}]. \tag{4.6}$$

Note that R must be countably infinite, since if it was finite then it would be a regular set. Equation 4.5 states that the limit of a sequence of finite (regular) sets can be a recursive set! Our next task is to prove this claim.

Consider a sequence of finite automata from the metric space (M, d_{fa})

$$F_1, F_2, ..., F_n, ...$$

such that

$$\forall i \, [i \in \mathrm{N} \Longrightarrow accept(F_i) = S_i].$$

We can use the bijection Φ from Corollary IV.21 to construct a corresponding sequence of binary automaton trees

$$T_1, T_2, ..., T_n, ...$$

such that

$$\forall i \, [i \in \mathrm{N} \Longrightarrow T_i = \Phi(F_i)]$$

and

$$\forall i \, [i \in \mathrm{N} \Longrightarrow accept(T_i) = accept(F_i) = S_i] \, . \tag{4.7}$$

Our sequence of finite automata $\{F_i\}_{i \in \mathrm{N}}$ corresponds to Reeker's concept of *acceptor series* (297), except that we do not use Reeker's technique of substituting graphs into graphs. Note that Corollary IV.21 and Theorem IV.26 imply that given a set S_i, there is one unique element of (M, d_{fa}) denoted by F_i, such that $accept(F_i) = S_i$, and consequently there is one unique (disregarding isomorphic relabeling) binary automaton tree T_i, such that $accept(T_i) = S_i$. A consequence of our length-ordering of R is that this sequence $\{T_i\}_{i \in \mathrm{N}}$ of binary automaton trees is a Cauchy sequence with respect to the metric d_{tr}.

Lemma IV.27 *The sequence of binary automaton trees given in Equation 4.7 is a Cauchy sequence within the metric space of all binary automaton trees with the metric* d_{tr}.

Proof: For this proof and the remaining ones of this section we will use the modal operators $\Box$ and $\Diamond$ with respect to the indicies of the sequences. For any sequence, the indicies form a total linearly ordered set (177), so that the indicies can be considered as discrete time, and the operators as from linear time temporal logic (see Appendix A).

Consider two sets from the sequence of sets of accepted words, say S_i and S_j. If $S_i = S_j$ then Theorem IV.26 implies that $F_i = F_j$, where $accept(F_i) = S_i$ and $accept(F_j) = S_j$. Further, the bijection Φ from Corollary IV.21 implies that

$$F_i = F_j \Longrightarrow T_i = T_j \Longrightarrow d_{tr}(T_i, T_j) = 0$$

where $T_i = \Phi(F_i)$ and $T_j = \Phi(F_j)$.

If $S_i \neq S_j$, and $j > i$, then

$$\forall \sigma_k, \sigma_l [(\sigma_k \in S_i \wedge \sigma_l \notin S_i \wedge \sigma_l \in S_j) \Longrightarrow \text{len}(\sigma_l) \geq \text{len}(\sigma_k)]$$

which follows from the length-ordering of R and the assertion given in Equation 4.6. Since the number of distinct words in R is not finite, and there are only a finite number of words of a given length (the symbol set Σ is finite), then with respect to any given index i,

$$\Diamond_{>i} (\exists l \forall k [\sigma_l \in S_j \wedge (\sigma_k \in S_i \Longrightarrow \text{len}(\sigma_l) > \text{len}(\sigma_k))]) .$$

(Here the $\Diamond_{>i}$ along with the S_j means 'there exists an index j greater than i such that the following is true') Given the index j for which this assertion is true, then it follows that

$$\Box_{>j} (S_j \subset S_k)$$

which implies that given any index i,

$$\Diamond_{>i} \Box (\exists l \forall k [\sigma_l \in S_j \wedge (\sigma_k \in S_i \Longrightarrow \text{len}(\sigma_l) > \text{len}(\sigma_k))]) . \qquad (4.8)$$

For any such distinct sets S_i and S_j, with $j > i$, then T_j $(\Phi(F_j))$ is equal to T_i $(\Phi(F_i))$ plus the additional nodes and arcs needed so as to accept the additional words in $S_j - S_i$ (since $S_i \subset S_j$). If some of the words in $S_j - S_i$ are the same length as the words of greatest length from S_i, then T_j and T_i differ at the level of the nodes that accept these 'longest' words from S_i. If any of the words from S_j have greater length than the words of greatest length in S_i, then T_j has one or more nodes of greater level than T_i, and if all the words in $S_j - S_i$ are longer than the words of greatest length from S_i, then T_j and T_i differ at a greater level than that of the nodes that accept the longest words from S_i. A result of the assertion given in 4.8 plus the fact that for any given length there are only a finite number of words (from Σ^*) of that length, is the following. Let N be the maximum possible number of words of length $\text{len}(\sigma_1)$, where σ_1 is the word of shortest length from R. Then for any set S_l, where

$l > N$, there exists another set S_k, such that $S_l - S_k$ contains only words longer than those of greatest length from S_k. Conversely, given a set $S_{\hat{k}}$, then:

1. There exists a set S_k whose longest word is equal in length to the longest word of $S_{\hat{k}}$.
2. There exists a set S_l such that $S_l - S_k$ contains only words longer than those of greatest length from S_k (and thus from $S_{\hat{k}}$).

The above arguments imply the following conclusion. Given $K \in \mathrm{N}$, there exists a set S_k whose longest words have length K, and there also exists a set S_l such that

$$d_{tr}(T_k, T_l) \leq 1/2^{K+1} < 1/2^K$$

where $S_k = accept(F_k)$, $S_l = accept(F_l)$, and $T_k = \Phi(F_k)$, $T_l = \Phi(F_l)$. This implies that with respect to some index N,

$$\forall\delta[\delta \in \mathrm{R}^+\} \Longrightarrow \Diamond_{>N}\Box d_{tr}(T_n, T_m) < \delta].$$

■

As a natural consequent of this result we have the following.

Corollary IV.28 *The sequence $\{F_i\}_{i\in\mathrm{N}}$ given by 4.7 is a Cauchy sequence within the metric space (M, d_{fa}).*

Proof: Theorem IV.25 implies that if $\{T_i\}_{i\in\mathrm{N}}$ is a Cauchy sequence, then so is $\{\Phi^{-1}(T_i)\}_{i\in\mathrm{N}}$, where $F_i = \Phi^{-1}(T_i)$. ■

Having completed these preliminary results, we can now prove the claim that the Cauchy sequence of regular sets $\{S_i\}_{i\in\mathrm{N}}$ converges to the given recursive set R.

Theorem IV.29 *Given a length-ordered recursive set R (not a finite set)*

$$R = \{\sigma_1, \sigma_2, \cdots, \sigma_n, \cdots\}$$

along with the sequence $\{S_i\}_{i\in\mathrm{N}}$ *given by 4.7 such that*

$$S_1 \subset S_2 \subset \cdots \subset S_n \subset \cdots$$

with

$$\forall i[i \in \mathrm{N} \Longrightarrow S_i \subset R]$$

and

$$S_i = \{\sigma_1, \ldots, \sigma_i\}$$

for each i*, then there exists a metric for the set of all regular sets such that*

$$\lim_{n\to\infty} S_n = R.$$

Proof: Given any regular set S, Theorem IV.26 implies that there exists a function f whose domain is the set of all regular sets and whose codomain is the set M from the metric space (M, d_{fa}), such that

$$accept(f(S)) = S.$$

Define the metric d_{rs} on regular sets such that for any two regular sets S and T,

$$d_{rs}(S,T) = \begin{cases} \inf\{1/2^{l+1} | l = \mathrm{len}(\sigma) \wedge \sigma \in T \wedge \sigma \notin S\} & \text{if } S \subset T \\ 1 & \text{else} \end{cases}$$

(Note that l is the length of the shortest word σ from T that is not a word from S) The arguments presented in the proof of Lemma IV.27 imply that

$$d_{rs}(S,T) = d_{fa}(f(S), f(T))$$

for any regular sets R and S. Although this equality holds for all regular sets, it would not have served as the definition of d_{rs} since $f(U)$ is undefined for any U that is not a regular set but is an element of the completion of this metric space. This equality is sufficient to prove that d_{rs} is a metric, following the reasoning of

Theorem IV.25, since d_{fa} is a metric. This equality and Corollary IV.28 imply that the sequence $\{S_i\}_{i\in \mathrm{N}}$ is a Cauchy sequence with respect to the metric d_{rs}.

Completing the metric space of all regular sets with the metric d_{rs} implies that

$$\lim_{n\to\infty} S_n$$

exists and is unique (189, 108). It follows from the proof of Lemma IV.27 that

$$\forall\delta\,[\delta \in \mathrm{R}^+ \Longrightarrow \Diamond\Box d_{rs}(S_n, R) < \delta]$$

which implies that

$$\begin{aligned} d_{rs}(\lim_{n\to\infty} S_n, R) &= \inf\{1/2^{l+1} | l \in \mathrm{N}\} \\ &= 0 \end{aligned}$$

and so

$$\lim_{n\to\infty} S_n = R$$

since the limit of any Cauchy sequence is unique. ■

This next corollary shows that the Turing complete metric space formed from the metric space of regular sets, includes all of the recursive sets.

Corollary IV.30 *The metric space of all recursive sets with the metric d_{rs} is a subspace of the Turing computable completion of the metric space of all regular sets with the metric d_{rs}.*

Proof: Theorem IV.29 implies that for any recursive set there exists a Turing computable Cauchy sequence of regular sets that converges to the recursive set. ■

This corollary implies that for any given recursive set R, there is a Turing computable Cauchy sequence of finite automata such that the limit point of this sequence is a 'machine' capable of accepting exactly the recursive set. Although this proof required the Turing

completion of the metric space of all regular sets with the metric d_{rs}, it did not address any other Turing computable Cauchy sequences except those generated in the manner of Equation 4.7. Are there possibly other Turing computable Cauchy sequences that converge to sets which are not recursive sets? The following theorem implies (but does not prove) that the answer to this question is yes. This theorem involves the *oracle*, which is a computing device that can solve the halting problem for Turing machines (152).

Theorem IV.31 *The Turing computable completion of the metric space of all regular sets with the metric d_{rs} is a subspace of the metric space of all oracle recursive sets with the metric d_{rs}.*

Proof: What needs to be proven is that all Turing computable Cauchy sequences of regular sets converge to oracle recursive sets.

The definition of d_{rs} from Theorem IV.29 implies that for any Cauchy sequence $\{S_i\}_{i\in\mathbf{N}}$ of regular sets, that

$$\lim_{n\to\infty} S_n = S \Longrightarrow (\sigma \in S \Longrightarrow \Diamond\Box\sigma \in S_i) \tag{4.9}$$

since if this wasn't true, then it would follow that

$$\Box\Diamond\sigma \notin S_i$$

which implies that

$$\Box\Diamond d_{rs}(S_j, S_k) \not< 1/2^{l+1}$$
$$l = \text{len}(\sigma)$$

which is not true because $\{S_i\}_{i\in\mathbf{N}}$ is a Cauchy sequence. The same reasoning implies that

$$\sigma \notin S \Longrightarrow \Diamond\Box\sigma \notin S_i. \tag{4.10}$$

For a set to be recursive both the set and its complement (with respect to Σ^*) must be Turing enumerable (359). Although this argument does not show that any Turing

machine could enumerate either S or its complement, assertions 4.9 and 4.10 imply that S and it's complement are 'oracle enumerable', and thus S is oracle recursive. This is based on the idea that an oracle (90, 1) that could solve the halting problem (for Turing machines), could decide for any given word σ, whether σ is or is not an element of S. This would not have been true if either

$$\sigma \in S \Longrightarrow \Box\Diamond\sigma \in S_i$$

or

$$\sigma \notin S \Longrightarrow \Box\Diamond\sigma \notin S_i$$

instead of 4.9 and 4.10. ■

Another related result is that any Turing computable string of symbols can also be generated by the limit point of a Turing computable Cauchy sequence of finite automata, i.e. the Turing completion of the metric space of finite automata includes machines that can generate all of the computable symbol strings (equivalent to computable numbers).

Theorem IV.32 *Given any Turing computable symbol string, there exists an element of the Turing computable completion of the metric space (M, d_{fa}) of finite automata that accepts only this symbol string.*

Proof: Since the string is computable, then it is the limit point of a Cauchy sequence of finite length strings within the metric space (Σ^*, σ) from Section 4.1 (291). Corollary IV.28 implies that there exists a Turing machine, such that for each of these finite length strings, a finite automaton that accepts only that string can be constructed using the techniques of this section. Thus this sequence of finite automata is a Turing computable Cauchy sequence, and the unique limit point is the machine that accepts only the computable string. ■

The previous results of this chapter plus Theorem IV.32 supply the necessary justification for the claim that the computational power of the 'machines' included in the completion of the metric space of finite automata is no greater than that of Turing machines.

Oracle Recursive
$\cup$
Recursively Enumerable
$\cup$
Turing Completed Regular
$\cup$
Recursive
$\cup$
Regular

Figure 4.8. Hierarchy of Sets

Theorem IV.33 *The set of Turing machines includes all of the elements of the Turing computable completion of the metric space of finite automata* (M, d_{fa}).

Proof: Consider the set S_i accepted by each finite automaton F_i in an arbitrary Cauchy sequence of such machines. The sequence of these sets corresponding to the Cauchy sequence can be produced by a Turing machine, since the Cauchy sequence of finite automata is Turing computable. This means that the cumulative union of these sets for a given j

$$\bigcup_{i \leq j} S_i \qquad j \in \mathrm{N}$$

is also Turing computable. Just as for the symbol strings of Theorem IV.32, if this cumulative union is computable for any arbitrary natural number j, then the machine that generates this cumulative union is the machine represented by the limit point of the Cauchy sequence. ■

A consequent of these results is the hierarchy of sets shown in Figure 4.8, where 'completed regular' refers to those sets that correspond to the computational power of the Turing completion of the metric space (M, d_{fa}) of finite automata. Neither the strictness of the inclusion of recursive sets in completed regular sets, nor the strictness of the inclusion of completed regular sets in recursively enumerable sets is proven here.

4.3 A Complete Metric Space Based on CSP

The last section demonstrated that the computational properties of the sequential computational model of finite automata can be related to the topological properties of a complete metric space based on the model and an appropriate metric. Additionally, the section showed that the computational concepts embodied in Chomsky's hierarchy of computational languages (and therefore models), can be recast in terms of a topological hierarchy based on the concept that the completion of a metric space can be 'higher' up the hierarchy than the other elements of the space. This section continues this tact, developing a complete metric space for a concurrent model of computation, Hoare's Communicating Sequential Processes (CSP) (165), and then using the standard topological tools to present alternative proofs to two of the main theorems from Hoare's book. This section closes with the demonstration that Hoare's proof of program concept based on the sat operator is equivalent to proofs based on the modal logic presented in Appendix A.

The first goal of this section is to show that CSP inherently generates a complete metric space and to relate the topological properties of this metric space to the algebraic and computational properties developed by Hoare. As explained in the introduction to this chapter, CSP is chosen as a representative of that class of concurrent computational models that are based on the *behaviour* of the computation. Before developing the metric space based on the *processes* that underly the CSP concept, another metric space based on the CSP *traces* needs to be presented.

CSP treats a process as either defined by its possible behaviour, or as defined by a system of equations that relate outputs (which along with the inputs constitute the 'behaviour') to inputs. These processes can be either *deterministic* or *nondeterministic*, but for both cases any process can be characterized by the set of possible input/output histories, which are called *traces*. Hoare defines a deterministic process as one that "can never refuse any event in which it can engage", and a nondeterministic process as one that "does not enjoy this property, i.e., there is at some time some event in which it can engage: but also (as a result of some internal ... choice) it may refuse to engage in that event, even though the environment is ready for it" (165). Within this section and those that follow any reference to CSP or other processes implies that the processes are deterministic, unless

stated otherwise.

Thus each process P generates a set denoted by

$$traces(P)$$

that contains all possible traces of P, where each trace is a sequential history of the input/output behaviour of P. A trace is denoted by

$$\langle a, b, c \rangle$$

where a, b, and c denote individual events, and their ordering within the brackets signifies the sequence in which they occur. By convention no two events occurs simultaneously, and every trace contains a finite number of events (165). For example, the process given by

$$VMS = (coin \rightarrow (choc - VMS))$$

is one which can perform the event called *coin*, then performs another (unnamed) process which first does the event *choc* then continues with the process VMS repeated anew. Thus the notation

$$x \rightarrow P$$

denotes the process that first does the event x, then follows with the process P.

Given this process VMS defined in an equational manner, another equivalent definition of VMS is given by the set of all possible traces of VMS, that is

$$traces(VMS) = \{\langle\rangle, \langle coin \rangle, \langle coin, choc \rangle, \langle coin, choc, coin \rangle, ...\}$$

where $\langle\rangle$ denotes the empty trace, that is the sequence that contains no events. Since for every process (by convention) there exists a time interval that precedes the start of the process, then every process contains the empty trace in its set of traces, even the process that does nothing. Note that the set of traces for a given process can be countably infinite, even though each trace only contains a finite number of events.

Thus a process can be defined either from its set of all possible traces or from one or more equations (possibly recursive). Whereas the previous section developed a metric for strings of symbols that are similar to the sequence of events in a trace (see the metric defined by Equation 4.3), this section first presents a metric for the space of all possible traces that can be formed from a given finite set of events. This metric is then used to define a metric on the space of all possible processes that can be defined (in the CSP sense of a process definition) given a finite set of events.

Since a trace is an ordered sequence (i.e. countable) of discrete events, the metric σ defined by Equation 4.3 can be modified to create a metric for the space of all (finite) traces. Consider the two traces denoted by

$$\langle x_1, x_2, ..., x_n \rangle$$

and

$$\langle y_1, y_2, ..., y_m \rangle$$

for some n and m elements of the natural numbers. Since the metric σ on strings of symbols from Σ^* is based on the idea of how many symbols at the beginning of each string are identical, the same concept can be used to define the following metric on traces denoted by π.

$$\pi(x,y) = \begin{cases} 0 & \text{if } x = y \\ \max\{1/2^k | x_k \neq y_k\} & \text{else} \end{cases} . \tag{4.11}$$

If

$$x = \langle x_1, ..., x_n \rangle$$

and

$$y = \langle y_1, ..., y_n, y_{n+1}, ..., y_m \rangle$$

where

$$x_i = y_i \qquad \text{for } 1 \leq i \leq n$$

then

$$\pi(x, y) = 1/2^{n+1}$$

based on the convention that since there is no x_{n+1} symbol, then

$$x_{n+1} \neq y_{n+1}.$$

That π is a metric follows from the same argument presented for the metric σ defined by Equation 4.3. Since traces must be of finite length, only the *max* function is required instead of a sup.

This metric π also satisfies the intuitive requirement that two traces x and y that are 'close' with respect to the metric are 'close' in terms of having more of their leading symbols identical. That is, as the metric distance between the two traces decreases, the number of events (starting from the first event) that must occur before the two sequences differ increases. Note however, that other intuitive concepts of closeness do not necessarily apply to this metric. For example, consider the two traces given by

$$x = \langle 0, 1, 0, 1, 0, 1, 0 \rangle$$

and

$$y = \langle 0, 0, 0, 1, 0, 1, 0 \rangle$$

which yields

$$\pi(x, y) = 1/4.$$

Compare this to the two traces

$$u = \langle 0, 1, 1, 1, 1, 1, 1 \rangle$$

and

$$v = \langle 0, 0, 0, 0, 0, 0, 0 \rangle$$

which also gives

$$\pi(u, v) = 1/4$$

Although the metric states that these two pairs of traces are equally 'far apart', an intuitive notion of closeness would conclude that x and y are closer than u and v.

Thus the set of all finite traces along with the metric π given by Equation 4.11 constitute a metric space. A natural question then, is whether this metric space is complete or not. Consider the Cauchy sequence of traces given by

$$\langle 0 \rangle, \langle 0, 1 \rangle, \langle 0, 1, 0 \rangle, \langle 0, 1, 0, 1 \rangle, \langle 0, 1, 0, 1, 0 \rangle, ...$$

Is the limit point of this sequence an element of the metric space? The answer is no, since no finite string can represent this limit. To prove this claim, consider that there does exist a finite string x

$$x = \langle x_1, ..., x_n \rangle$$

that is the limit point of this Cauchy sequence. If $x_n = 1$ the traces given by

$$y = \langle x_1, ..., x_n, 0 \rangle$$

and

$$z = \langle x_1, ..., x_n, 0, 1 \rangle$$

are both elements of the Cauchy sequence such that

$$\pi(y, z) = 1/2^{n+2}$$

But for any element u of the Cauchy sequence,

$$\pi(x, u) \geq 1/2^{n+1}$$

which means that x cannot be the limit point of the Cauchy sequence, since there are other elements of the sequence that are 'closer together' than x is to any other element.

This is also seen by considering each trace as the binary expansion of a number based on negative powers of 2, that is

$$\langle 0,1\rangle = 0\times 2^{-1} + 1\times 2^{-2} = 1/4$$

$$\langle 0,1,0,1\rangle = 0\times 2^{-1} + 1\times 2^{-2} + 0\times 2^{-3} + 1\times 2^{-4} = 1/4 + 1/16 = 5/16$$

Thus the limit point is given by

$$\sum_{k=1}^{\infty} \frac{1}{4^k}$$

which evaluates to 1/3, a number that cannot be represented with any finite string of zeros and ones.

Thus the completion of this metric space requires the inclusion of 'traces' that contain an infinite number of events. One logical question is whether the limit traces of Cauchy sequences represent in some manner unique processes, so that these limit points can be put into a one-to-one correspondence with the set of all processes (over some finite alphabet). In the previous example the Cauchy sequence of traces (ignoring the empty trace) can be generated by the following process named ONETHIRD:

$$\text{ONETHIRD} = (0 \rightarrow (1 \rightarrow \text{ONETHIRD}))$$

But this same Cauchy sequence can also be generated by the process OT:

$$\text{OT} = (0 \rightarrow (1 \rightarrow (0 \rightarrow (1 \rightarrow \text{OT}))))$$

So there is no one-to-one correspondence between processes and the limit points of arbitrary Cauchy sequences of traces.

Just as the symbol string metric given by Equation 4.3 formed a basis for the metric on sets of strings represented as binary automaton trees and given by Equation 4.4, the metric π on traces can form the basis for a me' ,n sets of traces represented as trees.

Consider the process ONETHIRD defined above, where

$$traces(ONETHIRD) = \{\langle\rangle, \langle 0\rangle, \langle 0,1\rangle, \langle 0,1,0\rangle, \langle 0,1,0,1\rangle, \langle 0,1,0,1,0\rangle, ...\}$$

Note that trace theory (299) provides a more efficient representation for this set of traces. Disregarding the commas between the events of a trace, and considering each trace as a symbol string from A^* (the empty trace corresponds to the empty word) without the angle brackets, where A is the alphabet of the process, then the unary prefix operator $pref$ yields the result

$$traces(ONETHIRD) = pref(\{01\}^*)$$

where

$$pref(X) = \{t \in A^* | \exists u[u \in A^* \wedge tu \in X]\}$$

Since each trace can be treated as a symbol string, which is exactly the idea behind the metric π, then the set of traces can be treated as a set of strings that are represented as a modification to the binary automaton tree. Just as the construction of the binary automaton tree was based on the length ordering of an arbitrary recursive set (see the discussion following Theorem IV.26), the construction of a tree representation for the set of traces requires that the set of traces be ordered by trace length. Before presenting the following theorem establishing that such sets can be length ordered, some additional notation must be defined. Note that since any finite alphabet can be represented with just the two symbol alphabet Σ

$$\Sigma = \{0,1\}$$

then all of the following results apply to processes and traces defined over this alphabet.

Definition IV.34 *Given a function F that maps processes to processes, then*

$$\mu X.F(X)$$

denotes the process (if it exists) that is the solution to the recursive equation

$$X = F(X)$$

Further, if F is of the form

$$F(X) = (x \rightarrow G(X))$$

where x is an event from the alphabet of the process, and G is a function that maps processes to processes (and could also be a function of x), then F is guarded.

For example, the process ONETHIRD can be denoted as

$$ONETHIRD = \mu X.(0 \rightarrow (1 \rightarrow X))$$

Definition IV.34 yields the following theorem.

Theorem IV.35 *Given the recursively defined process denoted by*

$$\mu X.F(X)$$

such that $F(X)$ is a guarded expression, then the set

$$traces(\mu X.F(X))$$

can be well ordered by trace length.

Proof: Denote the process by P, that is

$$P = \mu X.F(X)$$

Since for any process its set of traces contains the empty trace, which with length zero is the shortest trace, then the empty trace is the first element in the set of length

ordered traces for P. Denote the guarded expression $F(X)$ by

$$F(X) = (x \rightarrow G(X))$$

This implies that the next trace in the length ordering is (given that x comes before any other trace of the same length)

$$\langle x \rangle.$$

There are only three possibilities for $G(X)$, either it is recursively defined in terms of F, it is recursively defined independently of F, or it terminates (see Chapter 1 of Hoare (165)). If G terminates, then the finite number of traces generated can be length ordered. If G is recursively defined independently of F then the countable number of traces generated can be length ordered, since these traces form by definition a recursive set (although not explicitly stated in Hoare's book, this thesis assumes, as it seems that Hoare did, that the definition of such a G must be a total recursive function). If G is recursively defined in terms of F, then the trace generation process can be continued with the base case for G determining the next trace, and then the recursive definition generating the next trace from the definition (which is based on a guarded expression) for F.

Continuing in this manner the successive generation of traces can be continued indefinitely, with any arbitrary provision for ordering traces of the same length (of which there can only be a finite number for any given finite length). ■

Hoare's development of recursively defined processes actually implies that the process defined by

$$\mu X.F(X)$$

is a total recursive function, and so generates a set of all possible traces that is a recursive set. Although this alone would have proved the theorem, the given proof demonstrates the relationship between the process as defined by F, and the type of the function denoted by G (which to prove the theorem need only be total recursive). Since Theorem IV.35 forms the basis for the following metric on the space of those processes whose definitions satisfy

the requirements of the theorem, then the metric is only valid for deterministic processes (at this point), since the constraints on the process definition given in Theorem IV.35 are those for deterministic processes from Hoare's book (165).

Given the alphabet Σ, the technique used to construct the binary automaton tree of Section 4.2 can be used to construct a *binary trace tree* from any given deterministic process. The primary difference between the binary trace tree and the binary automaton tree is that the nodes marked with a '+' in the binary trace tree designate traces that are actually generated by the process, while the other nodes are included to simply populate each level and standardize the graphical representation of the tree. Figure 4.9 depicts the first three levels of the binary trace tree constructed from the process defined by ONETHIRD above. Note that this particular tree can be completely represented with a finite number of levels since the process ONETHIRD is defined recursively in terms of itself, which means that every trace longer than some fixed length must 'return' to the root node. The node on the third level labelled '$\langle 0, 1 \rangle$' could have its output arc labelled with '0' return to the root node, since any trace generated by ONETHIRD that passes thru this node could be considered as starting over again from the root node. The other nodes on the third level never lead to traces generated by ONETHIRD, so they can be terminated at this level also. But this termination technique cannot be used in the formal definition of the binary trace tree, since it it was, the binary trace trees for the processes ONETHIRD and OT would be different, whereas Corollary IV.38 (to follow) requires that they be identical. Note that every process generates the empty trace, so that every binary trace tree will have the root node (the empty trace) labeled with a '+'.

Definition IV.36 *Given the alphabet*

$$\Sigma = \{0, 1\}$$

and any process P defined over this alphabet in the formulation given by Theorem IV.35, the binary trace tree *corresponding to P is a binary tree such that the paths represent*

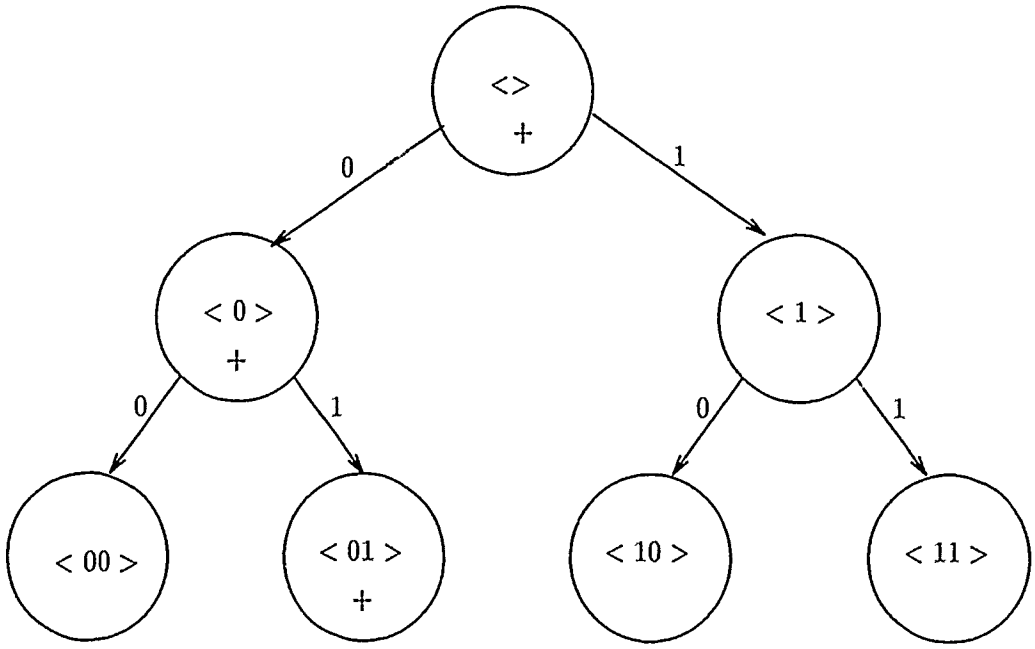

Figure 4.9. Partial Binary Trace Tree Representing ONETHIRD

elements of Σ^*, *and those paths corresponding to the elements of*

$$traces(P)$$

contain nodes differentiated from those nodes that are not contained in such paths.

Thus for a given process, each path in the binary tree that represents the binary trace tree corresponds to an element of Σ^*, since the concatentation of the symbols (0 or 1) on the arcs constituting the path forms a symbol string from Σ^*. And if all of the nodes on a given path are elements of the 'differentiated' set, then the symbol string from this path represents the sequence of events making up one of the possible traces that can be generated by the process. The empty word Λ represents the empty trace $\langle\rangle$, and corresponds to the root node of the binary trace tree. As shown in Figure 4.9, the differentiation of the nodes contained in the paths representing traces that can be generated by the process is by marking these nodes with a '+'. The ordering of the traces of equal length (here chosen as the natural lexicographic ordering) determines uniquely the actual graphical representation

of the binary trace tree, in that this ordering gives the sequence of the nodes from left to right on any given level.

Following the development of the metric given by Equation 4.4 for binary automaton trees, the next theorem gives a metric for binary trace trees, and thus a metric for deterministic processes.

Theorem IV.37 *Given the set of all processes defined by recursive formulas of the form*

$$\mu X.F(X)$$

such that $F(X)$ is a guarded expression, then the function d_{tr} given by Equation 4.4 is a metric for the set of all binary trace trees corresponding to these processes.

Proof: That d_{tr} is a metric for the set of all binary trace trees follows from the proof that it is a metric for the set of all binary automaton trees, since the only difference between the two types of trees is the marking convention for the nodes. The theorem then follows from the existence of the constructive technique used to generate unique binary trace trees from any given deterministic process, plus the fact that for any process P, the set

$$traces(P)$$

contains only traces of finite length, and this set can be put into a one-to-one correspondence with the set Σ^*. ■

Corollary IV.38 *Given the class of all processes that satisfy the requirements from Theorem IV.37, such that for any two processes P and Q*

$$P = Q \iff traces(P) = traces(Q)$$

then the function whose domain is the cross product of this class with itself, given by

$$d_p(P,Q) = d_{tr}(\hat{P},\hat{Q})$$

where $\hat{P}$ and $\hat{Q}$ are the binary trace trees constructed from the sets traces(P) and traces(Q) respectively, is a metric for this class.

Proof: Given any deterministic process P, the set

$$traces(P)$$

is uniquely determined, thus uniquley determining the associated binary trace tree. This implies that two binary trace trees are equal if and only if the corresponding sets of all process traces are equal. Thus

$$d_p(P,Q) = 0$$

implies that

$$P = Q$$

under the class definition of the processes comprising the domain of d_p. ■

Figure 4.10 depicts the first three levels of the binary trace tree generated by the process

$$\text{ONESEVENTH} = \mu X.(0 \rightarrow (0 \rightarrow (1 \rightarrow X)))$$

Just as ONETHIRD in a sense produced the binary expansion for the number 1/3, ONESEVENTH generates the binary expansion for the number 1/7. Comparing the binary trace tree for ONESEVENTH to that for ONETHIRD (Figure 4.9), yields

$$d_p(ONETHIRD, ONESEVENTH) = 1/4$$

since the two trees are identical up to the second level, but not at the third.

The equality definition given in Corollary IV.38 for processes circumvents the problem manifested in the previous observation that both of the processes ONETHIRD and OT generate the same binary trace trees, but the two processes do not have identical for-

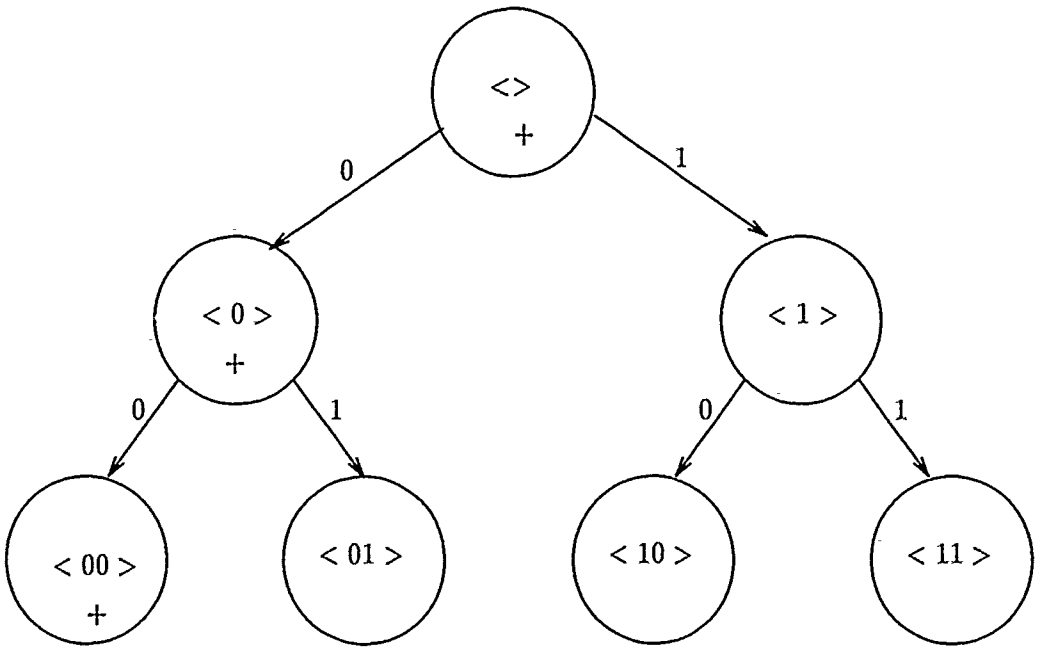

Figure 4.10. Partial Binary Trace Tree Representing ONESEVENTH

mulations. Thus the equality used here is not dependent upon formulation, but only upon behaviour.

Once the metric space of the class of processes and the metric d_p from Corollary IV.38 is established, the two primary results concerning deterministic processes from Hoare's book can be proven using the standard topological tools of complete metric spaces. Before proceeding with these derivations, the following definition and theorem are needed to demonstrate that the CSP definition of *continuity* of functions is equivalent to the standard metric space based definition. The particular wording of this definition and theorem is from Apostol's book (8). The primary reason for deriving Corollary IV.38 is to establish the existence of a metric space for determinsitic processes that can be used in conjunction with the definition and theorem.

Definition IV.39 *Let* (S, d_S) *and* (T, d_T) *be metric spaces and let*

$$f : S - T$$

be a function from S to T. The function f is said to be continuous *at a point p in S if for every $\epsilon > 0$ there is a $\delta > 0$ such that*

$$d_T(f(x), f(p)) < \epsilon \qquad \text{whenever } d_S(x, p) < \delta.$$

If f is continuous at every point of a subset A of S, we say f is continuous on A.

The following theorem, based on this definition, actually serves as an equivalent definition of continuity which is used in the remainder of this section.

Theorem IV.40 *Let*

$$f : S \to T$$

be a function from one metric space (S, d_S) to another (T, d_T), and assume $p \in S$. Then f is continuous at p if, and only if, for every sequence $\{x_n\}$ in S convergent to p, the sequence $\{f(x_n)\}$ in T converges to $f(p)$; in symbols,

$$\lim_{n\to\infty} f(x_n) = f\left(\lim_{n\to\infty} x_n\right)$$

This next definition presents the CSP definition of continuity (165).

Definition IV.41 *A function F from one set of all processes with a given alphabet into another set of all processes with a given alphabet is* continuous *if*

$$F(\bigsqcup_{i\geq 0} P_i) = \bigsqcup_{i\geq 0} F(P_i) \text{ if } \{P_i | i \geq 0\} \text{ is a chain}$$

Section 2.8.2 of Hoare's book presents the background and explanation of symbology used in this definition. Although Hoare's formal definition uses complete partial orders instead of sets of processes, any set of all processes with a given alphabet does form a complete partial order, so this wording is used in Definition IV.41 since the other results are based on such sets of processes. The next theorem equates these two definitions of continuity within the metric space defined by Corollary IV.38.

Theorem IV.42 *Let M denote the metric space given in Corollary IV.38, then a function F,*

$$F : M \rightarrow M$$

is continuous with respect to Theorem IV.40 if and only if it is continuous with respect to Definition IV.41.

Proof: The first step is to show that any chain of processes is also a Cauchy sequence with respect to the metric space given in Corollary IV.38. Consider the chain

$$\{P_0, P_1, ..., P_n, ...\}$$

then

$$n \leq m \Longrightarrow traces(P_n) \subset traces(P_m)$$

Since any finite sequence is Cauchy (that converges to the last element in the sequence), only infinite sequences need be considered. If

$$\alpha = d_p(P_0, P_1)$$

and

$$\alpha = 1/2^k$$

then

$$i > 1 \Longrightarrow d_p(P_1, P_i) \leq \alpha$$

and each P_i has a trace of length k or greater (by definition of a chain). Either there exists a finite l such that

$$i > l \Longrightarrow d_p(P_i, P_l) = 0$$

which implies that the chain is a Cauchy sequence that converges to P_l, or else there doesn't, in which case the lengths of the traces must continually increase (since for any given level there is only a finite number of nodes). Since the traces continue to get longer, and each trace remains in the processes that occur 'later' in the chain.

then for any natural number h, there exists another natural number N such that

$$(n > N \wedge m > N) \Longrightarrow d_p(P_n, P_m) \leq 1/2^h$$

Since for any positive real number ϵ there exists a natural number N such that

$$1/2^N < \epsilon$$

then for any positive real number ϵ there exists a natural number N such that

$$(n > N \wedge m > N) \Longrightarrow d_p(P_n, P\ ) \leq \epsilon \tag{4.12}$$

and the chain forms a Cauchy sequence.

The next step is to show that any chain with a least upper bound is convergent, such that the limit of the chain is the limit of the Cauchy sequence formed by the chain. Let P_∞ denote the limit of the chain (as defined by Hoare), then

$$traces(P_\infty) = \bigcup_{i \geq 0} traces(P_i)$$

Since the limit of the Cauchy sequence must be a process that contains all of the traces of those processes whose indices are greater than N for a given ϵ (see Equation 4.12), then for any chain the limit of the Cauchy sequence must contain all of the traces for all of the processes, and thus this limit is exactly that process denoted by P_∞.

Given that any chain with a limit is a convergent sequence, then this theorem can be proved by showing that two conclusions follow.

1. If F distributes across all convergent sequences then F distributes across all chains with limits (this shows that Theorem IV.40 implies Definition IV.41).

2. If F distributes across all chains with limits then F distributes across all convergent sequences (this shows that Definition IV.41 implies Theorem IV.40).

The first item is a consequent of the fact that any chain with a limit is also a convergent sequence, thus the final step is to prove the second item.

Consider the convergent sequence of processes given by

$$\{Q_0, Q_1, \ldots\}$$

Form a chain of processes from this sequence in the following manner. Let N_1 be the minimum N such that Equation 4.12 holds for $\epsilon = 1/2$. Then let N_2 be the minimum N such that the equation holds for $\epsilon = 1/4$. Continue this process such that N_i is the minimum N such that Equation 4.12 holds for $\epsilon = 1/2^i$. The resulting sequence

$$\{Q_{N_1}, Q_{N_2}, \ldots\}$$

has the property that

$$traces(Q_{N_i}) \subset traces(Q_{N_{i+1}})$$

for any natural number i, which makes this sequence a chain. Since the sequence has a limit Q_∞, the chain also has the same limit, because

$$s \in traces(Q_{N_i}) \Longrightarrow s \in Q_\infty$$

and

$$\lim_{i \to \infty} Q_{N_i} = Q_\infty$$

since only a finite number of the elements from the original convergent sequence can be missing from the chain. Additionally, since only a finite number of the processes from the original convergent sequence can be missing from the constructed chain, then if F distributes over the resultant chain it must also distribute over the original sequence. Thus distribution over all possible chains with limits implies distribution over all possible convergent sequences. ■

A consequent of Theorem IV.12 is that the basic fixed point theorem for deterministic CSP processes (see Section 2.8.2 of (165) for the actual proof) can be proven using the

metric space based definition of continuous functions instead of the chain based definition. This theorem states that

$$\mu X.F(X) = F(\mu X.F(X)) \tag{4.13}$$

which means that a recursively defined deterministic process (based on Γ guarded) has a solution that satisfies the recursive formulation. Additionally, by using the metric space continuity definition, Theorem IV.40 implies that a continuous function need not necessarily have a complete metric space for either the domain or the codomain. Contrast this with the chain based definition which requires that both the domain and the codomain be complete partial orders (which correspond to complete metric spaces based on the metric d_p).

The second major result regarding determinsitic processes is that for F a *constructive* function that maps processes to processes, the equation

$$X = F(X)$$

has a unique solution for X, given the following definition of a constructive function. In the following definition the $\restriction$ denotes the CSP restriction operator, so that $s \restriction A$, which represents the trace s restricted to the set A, is defined by

$$\langle a, b, c, d \rangle \restriction \{d, b\} = \langle b, d \rangle$$

Definition IV.43 *A function F whose domain and codomain are a set of processes is* constructive, *if F is monotonic with respect to the partial order that defines chains, and*

$$F(X) \restriction (n+1) = F(X \restriction n) \restriction (n+1) \tag{4.14}$$

for all processes X.

This unique solution result for processes defined in terms of constructive functions can also be obtained using the *contraction mapping theorem* for complete metric spaces, based on the following preliminary definition and results.

Definition IV.44 *Given the metric space* (X, d), *then the function* f,

$$f : X \to X$$

is a **contraction**, *or a* **contraction mapping**, *if there exists a real number* c,

$$0 \leq \epsilon < 1$$

such that

$$x, y \in X \Longrightarrow d(f(x), f(y)) \leq cd(x, y)$$

The function f *is* **nonexpansive** *if*

$$x, y \in X \Longrightarrow d(f(x), f(y)) \leq d(x, y)$$

Since the contraction mapping theorem requires contractions whose domain and codomain are the same complete metric space, the next theorem supplies such a complete metric space.

Theorem IV.45 *The metric space defined in Corollary IV.38 is complete.*

Proof: Consider the following Cauchy sequence

$$\{P_0, P_1, ...\}$$

For any natural number k (not zero), there exists another natural number N, which is the least natural number such that

$$n, m \geq N \Longrightarrow d_p(P_n, P_m) \leq 1/2^k$$

This implies that all traces of length k or less that are generated by P_n are also generated by P_m. Now consider the set formed by including these traces of length k

or less for every such k. This set, which can be length ordered by considering

$$k = 1, 2, ...$$

and is denoted by Q, satisfies the property that for every positive real number δ, there exists a natural number M, such that

$$n > M \Longrightarrow d_{tr}(\hat{P}_n, \hat{Q}) < \delta$$

where $\hat{P}_n$ is the binary trace tree corresponding to P_n, and $\hat{Q}$ is the binary trace tree formed from the traces contained in the length ordered set Q. Thus the set Q represents those processes (which are all equal to one another within the class defined by Corollary IV.38) which are the limit point of the Cauchy sequence. Since Q can be length ordered by a Turing machine, then Q is a recursive set that can be generated by a (deterministic) recursive process. Additionally, since any finite representation of the Cauchy sequence must be formulated in terms of a guarded expression (since every element of the sequence is either a guarded recursive expression or a finite representation), then the set Q can only be formulated by a guarded recursive (or finite) expression, and the limit point is an element of the metric space. ■

An interesting implication of the combination of this proof that Q is a recursive set generated by a Cauchy sequence within a complete metric space, and Corollary IV.30, is the following.

Corollary IV.46 *There exists an injective total function whose domain is the metric space of deterministic CSP processes defined by Corollary IV.38, and whose codomain is the Turing computable completion of the metric space of finite automata defined in Theorem IV.26.*

Proof: The existence of such a one-to-one correspondence is based on the existence of a one-to-one correspondence between the metric space of finite automata given by Theorem IV.26 and the metric space of regular sets given by Theorem IV.29 (see

the proof of Theorem IV.29), along with the statement from Theorem IV.29 that a recursive set is the limit point of a Cauchy sequence of regular sets. Since a deterministic CSP process corresponds to a recursive set of traces for the process, then the process can be represented as the limit point of a Cauchy sequence of finite automata that correspond to the regular sets. ■

This corollary implies that every deterministic CSP process can be put into a one-to-one correspondence with the limit point of a Cauchy sequence of finite automata, which follows from the recursive nature of the sets of traces generated by deterministic CSP processes. Thus in some sense, deterministic CSP processes are not as 'powerful' as Turing machines. As shown in later paragraphs, this does not hold true for nondeterministic CSP processes.

The final preliminary result is to show that the CSP concept of a constructive function from processes to processes is equivalent to that of a contraction mapping over the metric space of Corollary IV.38.

Theorem IV.47 *A function F whose domain and codomain are the class of processes given in Corollary IV.38 is constructive if and only if F is a contraction over the metric space given by Corollary IV.38.*

Proof: Only if proof:

For arbitrary processes X and Y,

$$d_p(X,Y) = 1/2^k \Longrightarrow X \uparrow k = Y \uparrow k$$

which implies that

$$F(X \uparrow k) = F(Y \uparrow k)$$

and that

$$F(X \uparrow k) \uparrow (k+1) = F(Y \uparrow k) \uparrow (k+1)$$

Since F is constructive, then by definition (see Equation 4.14)

$$F(X) \uparrow (k+1) = F(Y) \uparrow (k+1)$$

and so

$$d_p(F(X), F(Y)) \leq 1/2^{k+1} \leq (1/2)d_p(X, Y)$$

If proof:

(The CSP convention is for $n \geq 0$ in $P \uparrow n$, while the metric d_p is based on the convention that $n \geq 1$) For arbitrary k,

$$d(X, X \uparrow k) = 1/2^k$$

so that if F is a contraction, then

$$d(F(X), F(X \uparrow k)) \leq \alpha(1/2^k)$$

Since α must be strictly less than 1, then

$$d(F(X), F(X \uparrow k)) \leq 1/2^{k+1}$$

which implies that

$$F(X) \uparrow (k+1) = F(X \uparrow k) \uparrow (k+1)$$

■

With these preliminary results, what Hoare calls the 'fundamental theorem' of deterministic processes (see Section 2.8.3 of (165)), that

$$X = F(X)$$

has a unique solution for a constructive F, can be reworded as the contraction mapping theorem, a major topological result. This particular wording for the contraction mapping theorem is from Naylor and Sell (256).

Theorem IV.48 (Contraction Mapping Theorem) *Let (X, d) be a complete metric space and let*

$$f : X \rightarrow X$$

be a contraction. Then there is one and only one point x_0 in X such that

$$x_0 = f(x_0)$$

Moreover, if x is any point in X and x_n is defined inductively by

$$x_1 = f(x)$$
$$x_2 = f(x_1)$$
$$\vdots$$
$$x_n = f(x_{n-1})$$

then

$$x_n \rightarrow x_0 \text{ as } n \rightarrow \infty$$

Theorem IV.49 *The fundamental theorem of deterministic processes from CSP is equivalent to the contraction mapping theorem.*

Proof: Compare the wording of the contraction mapping theorem with the fundamental theorem from Hoare (165). The previous results complete the proof. ■

The application to CSP processes of the second part of the contraction mapping theorem (compare to the Recursion Theorem of Kleene (192)), which states that the unique fixed point associated with the contractive F can be found by inductively applying F to any initial element (process), can be domonstrated using the process ONETHIRD. Consider that the definition of ONETHIRD uses the function F, such that

$$F(X) = (0 \rightarrow (1 \rightarrow X))$$

To show that F is contractive, consider that if

$$d_p(X, Y) = 1/2^k$$

then

$$d_p(F(X), F(Y)) = 1/2^{k+2}$$

and in general

$$d_p(F(X), F(Y)) = (1/4)d_p(X, Y)$$

So the contraction mapping theorem states that with any arbitrary choice of a process X, repeated application of F to this process yields that single unique process that is the fixed point for F. Thus if

$$X_1 = F(X) = (0 \rightarrow (1 \rightarrow X))$$

$$X_2 = F(F(X)) = (0 \rightarrow (1 \rightarrow (0 \rightarrow (1 \rightarrow X))))$$

then as n becomes arbitrarily large, X_n becomes arbitrarily 'close' to the process ONETHIRD, such that

$$\lim_{n \to \infty} X_n = ONETHIRD$$

Note that this is true even if the process X is recursive, which means that X can produce traces of unbounded length. For any arbitrary positive real number ϵ, there exists a natural number k such that

$$1/4^k \leq \epsilon$$

and so

$$d_p(X_k, X) \leq \epsilon$$

Just as a constructive function is equivalent to a contraction mapping, the nonexpansive function defined in Definition IV.44 is equivalent to the CSP concept of a *nondestructive* function. a function that satisfies

$$F(P) \upharpoonright n = F(P \upharpoonright n) \upharpoonright n \text{ for all } n \text{ and } P$$

Theorem IV.50 *A function F whose domain and codomain are the class of processes given in Corollary IV.38 is nondestructive if and only if F is nonexpansive over the metric*

space given by Corollary IV.38.

Proof: Follows the reasoning for the proof of Theorem IV.47, with the difference that

$$d_p(X, X \uparrow k) = 1/2^k$$

implies that if F is contractive, then the $\leq$ in

$$d_p(F(X), F(X \uparrow k)) \leq 1/2^k$$

implies that only

$$F(X) \uparrow k = F(X \uparrow k) \uparrow k$$

can be stated. ■

Another consequent of the topological analysis of CSP processes is that results from topology can be translated into their CSP equivalents. For example, since any contraction is also uniformly continuous (256), the following theorem could be added to Hoare's book.

Theorem IV.51 *If F is a constructive function over a complete partial order, then F is continuous.*

Further, the requirement that a constructive function be monotonic is redundant, since the equivalency between contractive and constructive functions shown in Theorem IV.47 did not require the monotonicity property.

4.4 A Complete Metric Space Based on UNITY

The last section demonstrated that the process based CSP model of computation can be used to develop a complete metric space, such that the metric gives a somewhat intuitive notion of the 'closeness' between two programs, where a program is defined as one or more processes (see Definition II.5). The purpose of this section is to show that an imperative shared variable model for computation, Chandy and Misra's UNITY (64), can be mapped into an equivalent process based CSP representation. Unfortunately, CSP is based on the nonsimultaneous execution of atomic events, whereas UNITY has an assignment component operator ||, that denotes the simultaneous execution of the two argument components. There are two ways to address this disparity, one being the restructuring of the analysis of a UNITY program so that atomic actions include multiple assignment components, and the other being a modification to the UNITY execution model to prevent the simultaneous execution of assignment components. This section presents approaches utilizing both techniques, along with additional formalism regarding what is meant by an atomic action. Both approaches are used to develop metric spaces of UNITY programs based on the metric space of CSP processes from the last section.

Definition IV.54 gives a modified UNITY execution model that eliminates the simultaneous execution problem, called the *standard execution model*. With both execution models defined (Chandy and Misra define the UNITY execution model (64)), Definition IV.57 presents the formalism regarding atomic actions within the UNITY execution model, while Definition IV.56 formalizes exactly what pieces of a UNITY program should be considered atomic with respect to this standard execution model. These definitions form the basis for the primary results of this section, which are equivalences between UNITY programs and CSP programs that can be used to define a metric space of UNITY programs under either execution model.

Since the definition of 'equivalent' from Chapter I (Definition II.9) depends upon a reference function, this section will present equivalence inducing mappings that are defined with respect to the identity function on states. Thus the input program to these mappings is called 'equivalent' to the image (under the mapping) without explicit reference to this identity function. This implies that given two UNITY programs, application of such a

mapping permits the use of the metric d_p from Section 4.3, thus giving an intuitive measure of the 'closeness' of the UNITY programs. The resulting metric space of UNITY programs provides the framework for the topological analysis of UNITY programs, and also leads to the investigation into whether this space is complete without having to add objects that are not UNITY programs (just as Section 4.2 showed that completing the metric space of finite automata required 'machines' that are not finite automata).

Although Corollary IV.65 concludes the material regarding the metric space of UNITY programs, this section contains additional material relating the UNITY weak fair choice operator on statements to the CSP concurrency operator on processes, culminating with Theorem IV.71. This additional material appears here because of the close analogy between Theorem IV.71 and Lemma IV.63, a result which is needed for the proof of Corollary IV.65. Theorem IV.71 also establishes the transition into Chapter VI.

Although this modified execution model for UNITY, called the standard execution model, is different from the UNITY execution model of Chandy and Misra (64), all of the reasoning techniques presented for UNITY can still be applied to the standard model. This is because any assignments that would yield different results if executed sequentially, instead of simultaneously, can be rewritten as a single assignment component, and thus still executed simultaneously.

Before examining the first mapping of UNITY to CSP, the CSP process to process operators for *choice*, denoted by $|$, and for *concurrency*, denoted by $\|$, need explaining. If x and y denote events (or possibly event variables) such that $x \neq y$, and P and Q denote processes, then the notation

$$(x \rightarrow P | y \rightarrow Q) \tag{4.15}$$

represents the process that either performs x followed by the events of P, or else performs y followed by the events of Q. The choice is deterministic, based on Hoare's definition of deterministic given at the beginning of Section 1.3. Although the traditional definition of deterministic implies that there exists a Turing machine whose input is a description of both processes (i.e. the process $(x \rightarrow P)$ and the process $(y \rightarrow Q)$), along with possibly other input information, and whose output is exactly one of these processes, the definition

of deterministic here is based on the concept of weak fairness (see Definition I.1). The factors that influence the choice are typically either some input from another process (such as a 'user'), or environmental effects such as the details of the hardware that the process might be running on. In the general case, where the choice of the first event can be made from those elements constituting a set, there is the notation for a *general choice* operator given by

$$(z : A \rightarrow R(z)) \tag{4.16}$$

This process permits the deterministic choice of an instantiation for z from the set A as the first event, while the remaining events are given by the formulation $R(z)$. For the choice given by Equation 4.15,

$$A = \{x, y\}$$

and

$$R(z) = \begin{cases} P & \text{if } z = x \\ Q & \text{if } z = y \end{cases}$$

If the set A from Equation 4.16 is empty, then the resultant process is the one which performs no events, that is

$$(z : \{\} \rightarrow R(z)) = STOP \tag{4.17}$$

The general choice operator can be used to define the concurrency operator, as given by

$$(P \| Q) = (z : C \rightarrow \dot{P} \| \dot{Q}) \tag{4.18}$$

where

$$P = (x : A \rightarrow P(x))$$

$$Q = (y : B \rightarrow Q(y))$$

and

$$C = A \cap B \cup (A - \alpha(Q)) \cup (B - \alpha(P))$$

such that $\alpha(P)$ denotes the set of all possible events of P, called the alphabet of P (likewise for Q), and

$$\hat{P} = \begin{cases} P(z) & \text{if } z \in A \\ P & \text{otherwise} \end{cases}$$

and

$$\hat{Q} = \begin{cases} Q(z) & \text{if } z \in B \\ Q & \text{otherwise} \end{cases}$$

The $\|$ operator is both commutative and associative, which, along with Equation 4.18 implies the following:

$$(c \rightarrow P)\|(c \rightarrow Q) = (c \rightarrow (P\|Q)) \quad \text{if } \alpha(P) = \alpha(Q)$$
$$(c \rightarrow P)\|(d \rightarrow Q) = STOP \quad \text{if } \alpha(P) = \alpha(Q) \text{ and } c \neq d$$

Thus whenever P and Q have identical alphabets,

$$traces(P\|Q) = traces(P) \cap traces(Q) \quad \text{if } \alpha(P) = \alpha(Q)$$

In the more general case, which can be stated by

$$\alpha(P) \neq \alpha(Q)$$

and

$$a \in (\alpha(P) - (\alpha(P) \cap \alpha(Q)))$$
$$b \in (\alpha(Q) - (\alpha(P) \cap \alpha(Q)))$$
$$c \in (\alpha(P) \cap \alpha(Q))$$
$$d \in (\alpha(P) \cap \alpha(Q))$$

it follows from Equation 4.18 that

$$(a \rightarrow P)\|(c \rightarrow Q) = (a \rightarrow (P\|(c \rightarrow Q)))$$
$$(c \rightarrow P)\|(b \rightarrow Q) = (b \rightarrow ((c \rightarrow P)\|Q))$$

$$(a \rightarrow P)\|(b \rightarrow Q) = ((a \rightarrow (P\|(b \rightarrow Q)))|(b \rightarrow ((a \rightarrow P)\|Q)))$$

and

$$traces(P\|Q) = \{t | t \restriction \alpha(P) \in traces(P) \wedge t \restriction \alpha(Q) \in traces(Q) \wedge t \in (\alpha(P) \cup \alpha(Q))^*\}$$

Given two processes P and Q, the notation $P\|Q$ represents a process whose traces are the interleavings of the individual traces of P and Q, with the constraint that whenever any event common to both alphabets occurs, it occurs for both processes (instead of being repeated). Thus these common events from both alphabets represent events that occur simultaneously within both processes (the interpretation being that these simultaneous events are the same event), and these common events synchronize the two processes at the instant of time they occur.

UNITY programs consist of assignment statements along with the fair choice operator on statements, and assignment components along with the composition operator on components. The symbol || will be used here for the composition operator, which is the same one used in Chandy and Misra's book, but instead of their symbol for fair choice the one used here is the |. Since, as Chandy and Misra state, the always section of a UNITY program is unnecessary, it will not be considered in the following analysis (64). And since the initially section consists of a *proper* set of assignments (see Chapter 2 of Chandy and Misra's book for a definition of proper), the following theorems regarding assignments also apply to those of the initially section. Note that UNITY requires the set of statements in a given program to remain unchanged by the execution of the program (See Section 22.7.4 of Chandy and Misra), so that the proofs of the following theorems do not need to consider sets of ordered n-tuples of events that change with time, a concept which is expressed in the following lemma.

Lemma IV.52 *There exists a total function f whose domain is the set of all UNITY statements, such that for any UNITY statement S, the evaluation f(S) is a unique finite set of n-tuples of events.*

Proof: See Section 22.7.4 of Chandy and Misra (64). ■

This lemma states that for any UNITY statement there exists a mapping of that statement into a set of sequences of events. Since the assignment components of a UNITY program are considered to execute simultaneously (64), whereas the definition of 'event' from Chapter I does not permit true simultaneity, these sequences of events represent the different possible execution sequences for the assignment components assuming that these assignments must execute sequentially. For a finite number of assignment components, which is mandated by the UNITY model, there will only exist a finite number of these possible execution sequences, or *interleavings.*

The first step is to show that the basic unit of an assignment component, represented by

$$var := expr \tag{4.19}$$

where *var* denotes a variable and *expr* denotes an expression, can be mapped into an equivalent CSP process.

Lemma IV.53 *The UNITY assignment denoted by Equation 4.19 can be mapped by a total function f into an equivalent (see Definition II.9) CSP process.*

Proof: Define the state of a UNITY program as a vector representing the values of all program variables, of which there can only be a finite number. If the assignment of Equation 4.19 is considered an event, then there exists a total function that maps an event of this type into a unique state, given the previous state (before this event). Section 3.3 of Chandy and Misra details the UNITY program execution model that supports this claim. Thus the equivalent CSP process is one whose only nontrivial trace is

$$\langle e, \surd \rangle$$

where $\surd$ denotes the CSP termination event, and e denotes the event that generates the identical state as does the UNITY assignment. ■

The development of a metric space of UNITY programs based on the UNITY execution model (see Chapters 2 and 3 from Chandy and Misra (64)), is intermixed with

the development of a metric space of UNITY programs based on the modified execution model. Many of the following results (the wordings specify which ones) actually apply to both execution models and both metric spaces. This is because the primary difference between the two execution models is the difference between what an atomic action is in each model. Before presenting the two definitions of the atomic actions, the following definition and theorem define the modified execution model and present an important consequent of the definition.

The basic tenet of this modified execution model is the resolution of the conflict between the nonsimultaneous property of events (which CSP assumes), and the simultaneous execution of assignment components in the UNITY execution model. This next definition presents the idea of a modified UNITY execution model, such that no two assignment components can occur simultaneously.

Definition IV.54 *Given two UNITY assignments $S1$ and $S2$, both of the form given in Equation 4.19, then under the* standard execution model *the UNITY process*

$$S1 \| S2$$

denotes that the two assignments can execute in either order.

Unless otherwise stated, the remainder of this section assumes the standard execution model.

Although this standard execution model for UNITY programs is different than the one given by Chandy and Misra (64), is it useful? There is more than one answer to this question, but one of the answers is stated in the following theorem.

Theorem IV.55 *The set of all UNITY programs that execute under the UNITY execution model is a proper subset of the set of all UNITY programs that execute under the standard execution model.*

Proof: Since Definition IV.54 implies that any UNITY program that executes under the UNITY execution model also executes under the standard model, then what must

be shown is the existence of at least one UNITY program that cannot execute under the UNITY model, but can execute under the standard model. One such program is given by:

```
Program E
        declare
                x : integer
        initially
                x = M
        assign
                x := x+1 || x := x-1
end
```

Not only does program E execute under the standard model, but it has a fixed point, which is that x equals its initial value. So program E demonstrates that there are syntactically correct UNITY programs that do not even execute under UNITY, but can execute and also have a fixed point under the standard execution model. It is true that program E could be rewritten as two statements so that it would generate the same sequence of states as does program E, that is it would contain

x := x+1 if x = M-1 or x = M | x := x-1 if x = M+1 or x = M

but this modified program would not have a fixed point.

This simple example demonstrates the major difference between the UNITY execution model and the standard execution model. Under UNITY, forced sequencing of assignments cannot be achieved within a single statement. This means that the only technique to force two or more assignments to execute sequentially is to move the assignments into separate statements. However, this poses two problems. First, fixed point analysis is based on statements, and second, the guidelines for mapping UNITY programs to different architectures often call for partitioning the statements among the processors. This can cause problems if the sequentially executing assignments end up on different processors, mainly because of the possible lack of control over the sequencing, and problems with variables that are common to the multiple assignments. The standard execution model can

help remedy these problems, because it is based on the concept of a 'hierarchy of arbitrary sequential execution', that is the execution model at the assignment component level is very similar to that at the statement level.

The concept of simultaneous assignments still exists within the standard execution model. Consider assignments of the form

x,y := M,N

which under both models executes both individual assignments at the same time. The philosophy of the standard model is that this assignment can be represented as a single atomic action within the CSP model. This action corresponds to the assignment of the new values to the state vector (a vector denoting all the values of the named program variables). The obvious question then, is what exactly is an atomic action? For example, the UNITY fragment

x,locky := f(x,y),false if locky || requesty := true if ¬locky

can be rewritten as

x,locky,requesty := x+(f(x,y)-x)locky,false,1-locky

with the change of type of requesty and locky from boolean to [0,1]. Thus the following definition.

Definition IV.56 *Within the standard execution model, an* atomic action *(for CSP purposes) corresponds to a single assignment component.*

And this next definition defines the atomic action for the UNITY execution model.

Definition IV.57 *Within the UNITY execution model, an* atomic action *(for CSP purposes) corresponds to a single assignment statement.*

Thus under each model, an atomic action represents one or more variable assignments that occur simultaneously.

Can the atomic actions under one execution model be mapped into atomic actions in the other, while still retaining the program properties? As seen in Theorem IV.55, there exists UNITY programs that execute under the standard execution model that do not (without modification) execute under the UNITY execution model. So the answer for the

mapping from standard execution to UNITY execution models is that such a mapping, if it exists, can be a function of the program to be mapped. This means that few generalizations can be made about such a mapping. However, for the mapping in the other direction, the following result can be shown.

Theorem IV.58 *Given a UNITY program P that executes under the UNITY model, and a set S whose elements are the (predicates) properties of P, then there exists another UNITY program Q that executes under the standard model, such that each element of S is a property of Q.*

Proof: The UNITY execution model (Chapters 2 and 3 of Chandy and Misra (64) requires that simultaneous execution of assignment components within a single statement generates a unique state vector, given a unique state vector prior to the execution. This means that the program P can be mapped into the program Q, so that each statement of P corresponds to a statement in Q consisting of one assignment component that assigns values to the state vector. The proof of Theorem IV.61 contains the details substantiating the implications of the UNITY execution model. ■

An intuitive proof for this theorem is that since the program P can only compute a partial recursive function (217), then this function can be rewritten as an algorithm than uses only sequential composition (120, 230, 258). Thus the program Q can be written without multiple simultaneous atomic actions (corresponding to the mutiple assignment components). This tact is not used in the proof because the standard theorems about what is necessary to compute any algorithm or function utilize fixed sequential execution. instead of the arbitrary sequential composition allowed under the standard execution model.

Another answer to the question of the usefulness of the standard execution model follows from considering the relationship between Dijkstra's guarded commands (104), which form the basis for parallel programming models and languages such as CSP (165). Distributed Processes (150), and UNITY. The execution model for these guarded commands states that a construct of the form

*[guard1 — command1 [] guard2 — command2]

will randomly (with the fair choice assumption, see Chapter I) choose to execute the command whose guard is *true* until all guards evaluate to *false*. Thus the UNITY execution model for a program consisting of the two statements

s | t

corresponds to

*[*true* → s [] *true* → t]

in the guarded command syntax.

Consider the guarded structure

[guard1 → command1 [] guard2 → command2]

whose execution is one random choice of any command with a *true* guard. This corresponds to the UNITY construct

s1 if b1 | s2 if b2

where b1 (b2) is *true* if guard1 (guard2) is *true*, but once either b1 or b2 evaluates to *true* then the other immediately becomes permanently *false*. Additionally, the statements s1 and s2 perform whatever assignments command1 and command2 do, plus any additional assignments to implement the requirements on b1 and b2. As an example, consider

[x ≥ y → z := 1 [] x ≤ y → z := 2]

which will assign either a 1 or a 2 to z, deterministically if x is not equal to y, or arbitrarily if they are equal. A corresponding UNITY program is

z,flag := 1,false if x ≥ y ∧ flag | z,flag := 2,false if x ≤ y ∧ flag

where the boolean variable 'flag' is initialized to 'true'. However, note that this correspondence is not as strong as in the previous example. This is because in the guarded construct there is a sense of locality in evaluating the guards, that is, no other statements external to those between the [and the] can affect the truth or falsity of each guard. However, in the UNITY program if there are other statements besides these two then this locality concept can be violated. Such a violation would occur if the variable 'flag' were somehow assigned a value in another statement in the UNITY program that could execute after one of these two statements but before the other. The standard execution model though, does provide a corresponding UNITY program that preserves this locality. This program is

z,flag := 1,false if x ≥ y ∧ flag || z,flag := 2,false if x ≤ y ∧ flag

This program retains the locality concept because these two components will execute in immediate succession, and the idea of a random choice is preserved because the sequencing of the components is arbitrary. Note that this program could not even execute under the UNITY model because of the possibility of simultaneously assigning both 1 and 2 to z.

A direct result of Lemma IV.53 and Definitions IV.54 and IV.56 is that a sequence of UNITY assignments corresponding to multiple atomic actions within one statement, under the standard execution model, can also be mapped into an equivalent CSP process.

Corollary IV.59 *Given two UNITY assignments $S1$ and $S2$, both of the form given in Equation 4.19 and executing under the standard execution model, then the UNITY process*

$$S1 \| S2$$

is equivalent to the CSP process

$$P1 \| P2$$

where $P1$ is equivalent to $S1$ and $P2$ is equivalent to $S2$.

Proof: This follows from Lemma IV.53 and

$$traces(P1) = \{\langle\rangle, \langle s\rangle\}$$
$$traces(P2) = \{\langle\rangle, \langle t\rangle\}$$

where s denotes the assignment of $S1$, and t denotes the assignment of $S2$, since the definition of $\|$ implies that

$$traces(P1 \| P2) = \{\langle\rangle, \langle s\rangle, \langle s,t\rangle, \langle t\rangle, \langle t,s\rangle\}$$

■

Under the UNITY execution model a UNITY program containing the statement

x := x-1 || x := x+1

does not have a well defined execution, since both assignments must occur simultaneously.

However, this statement is defined under the standard execution model, and is equivalent to the process that can execute either

$$x := x\text{-}1 \; ; \; x := x+1$$

or

$$x := x+1 \; ; \; x := x\text{-}1$$

where the ';' denotes sequential composition. The corresponding result regarding the UNITY execution model relates simultaneous execution of assignment components to a single assignment on the state variable.

Corollary IV.60 *Given two UNITY assignments S1 and S2, both of the form given in Equation 4.19 and executing under the UNITY execution model, then the UNITY process*

$$S1 \| S2$$

is equivalent to the CSP process

$$P$$

where the only nontrivial trace of P represents a single assignment to the variable denoting the state for the UNITY process.

Proof: Follows directly from the definition of the UNITY execution model (see Chapters 2 and 3 of Chandy and Misra (64)). Since all of the assignment components within a single statement must execute simultaneously, any individual component of the state vector is only assigned a new value at most once. The proof of Theorem IV.61 contains the details supporting this proof. ■

This next theorem is the first major consequent of the previous two lemmas and Corollary IV.59 and IV.60, and supplies the foundation that leads to the primary result of this section. Theorem IV.64.

Theorem IV.61 *There exists a total function f whose domain is the set of all UNITY assignment statements under the standard execution model, and there exists a total function g whose domain is the set of all UNITY assignment statements under the UNITY execution*

model, such that given the assignment statement S, the evaluation f(S), under the standard model, or the evaluation g(S), under the UNITY model, is a CSP process that is equivalent to S.

Proof: Enumerated assignments, quantified assignments, and quantified expressions are treated separately.

Enumerated Assignment: Induction, along with Corollary IV.59 and IV.60 proves the case where (see Section 2.3.2 of Chandy and Misra (64))

$$variable - list := simple - expr - list$$

For the case of

$$variable - list := conditional - expr - list$$

where

$$conditional - expr - list \quad ::= \quad simple - expr - list \qquad \text{if } boolean - expr$$
$$\{\sim \; simple - expr - list \qquad \text{if } boolean - expr\}$$

that there exists such a total function f or g follows from this statement from Chandy and Misra:

> An assignment with a *conditional – expr – list* causes assignment of values from any constituent *simple – expr – list* whose associated boolean expression is *true*. If none of the boolean expressions is *true*, the corresponding variable values are left unchanged. If more than one boolean expression is *true*, then all of the corresponding *simple – expr – lists* must have the same value; hence any one of them can be used for assignment. This guarantees that every assignment statement is deterministic: A unique program state results from executing an assignment in a given state.

In addition, if the UNITY model is to be considered as a two level model, such as the two level lambda calculus (258), then these boolean expressions must evaluate to either *true* or *false* at 'run time', since the UNITY/CSP equivalences are implicity run time equivalences ('run' means execution).

Quantified Assignment: The existence of the total function f follows from the UNITY requirement that only a finite number of instantiations exist for the quantification (see Section 2.3.3 of Chandy and Misra), and that a total function g must exist whose input is the quantification and whose output is a composition of assignment statements with all scoped variables bound. Thus the function f, when its input is such a quantification, has as output the CSP process that is equivalent to the output of g. Since g evaluates to an empty statement if there does not exist an instantiation, then in this case the output of f would be the CSP process whose only nontrivial trace is

$$\langle\surd\rangle$$

This CSP process, denoted by SKIP, is equivalent to the empty statement since both, when considered as functions from the previous state to the next state, are the identity function.

Quantified Expression: The quantified expression, of the form

$$expr ::= \langle op\ quantification\ expr \rangle$$

can always be evaluated to a unique value at run time, since the quantification can only include individual variables (over a finite domain), and if there is no evaluations for the quantification, then this expression returns unique default values. ■

Although the 'variables' addressed in these results refer to the named variables in a UNITY program, note that an implemented version of a given UNITY program may require additional variables. For example, consider the single assignment statement that swaps the values of two variables named x and y:

x,y := y,x

If the program variables are those associated with an implementation on a sequential architecture, then certainly this statement will also require at least one (the classic proof of correctness would need two) additional temporary variable, so that the actual execution of this statement would be:

temp := x

x := y

y := temp

Theorem IV.61 along with Corollary IV.59 and IV.60 supply the proof that any UNITY statement can be mapped by a total function into an equivalent CSP process. But in a UNITY program containing multiple statements, both the standard model and the UNITY model generate arbitrary unbounded sequencing of statement executions, using the fair choice operator | on statements. For example, consider the UNITY program P1 (the declare and initially sections have been ignored):

Program P1

assign

S1 | S2

end

S1 and S2 denote arbitrary statements. One technique for mapping P1 into an equivalent CSP description would be to map statements S1 and S2 into equivalent CSP processes, and then decide which CSP operator the UNITY | should be mapped into. Before stating the result which answers this question, the following Lemma is given.

Lemma IV.62 *Given the guarded CSP processes (see Section 4.3) P1 and P2, such that P1 and P2 have only finite length traces, then there exists a unique process that satisfies*

$$X = (P1|P2); X$$

that is, there exists a unique process

$$\mu X.(P1|P2); X$$

Proof: Let F denote the function whose domain and codomain are CSP processes such that

$$F(X) = (P1|P2); X$$

That the codomain of F is the class of CSP processes follows from X being a process, $(P1|P2)$ being a process, and the composition of two processes being a process (1165).

That $(P1|P2)$ is a process results from both $P1$ and $P2$ being guarded, that is there exists events x and y, and processes $P1(x)$ and $P2(y)$ such that

$$P1 = (x \rightarrow P1(x))$$
$$P2 = (y \rightarrow P2(y))$$

Consider two distinct processes X and Y, Corollary IV.38 implies that

$$X \neq Y \Longrightarrow \exists M[M \in \mathrm{N} \wedge d_p(X,Y) = 1/2^M]$$

where d_p is the metric used to define the metric space of CSP processes from Section 4.3 (see Corollary IV.38 for the definition of d_p). Next consider the instantiation of the process $(P1|P2)$ as (chosen arbitrarily) $P1$. Since $P1$ is guarded, the set

$$traces(P1)$$

contains a shortest (nontrivial) trace whose length is greater than or equal to one, since this shortest trace must contain x as its first event. Thus if the natural number N equals the length of this shortest trace,

$$N > 0 \Longrightarrow d_p(P1;X,P1;Y) \leq 1/2^{M+N} < 1/2^M$$

Thus, since this still holds if $P2$ had been chosen instead,

$$d_p(F(X),F(Y)) \leq (1/2)d_p(X,Y)$$

which means that F is a contraction mapping on the class of CSP processes. The Contraction Mapping theorem (see Section 4.3) completes the proof. ■

Note that the requirement for the two processes to be guarded ensures that the mapping F is a contraction (the guard supplies the $N > 0$ assertion), while the requirement that

the two processes have finite traces ensures that the evaluation

$$d_p(P1;X, P1;Y)$$

can be justified (and is not zero by definition). This raises an important issue, that Theorem IV.64 is not valid for UNITY statements that do not generate finite events. Lemma IV.62, needed for Theorem IV.64, requires finite traces for the constituent processes, which can be traced back to the requirement in Corollary IV.59 and IV.60 that to map a set of UNITY assignment components into an equivalent CSP process, the number of possible traces of the process must be finite.

This next lemma is the result regarding the mapping of the UNITY program given by P1 into an equivalent CSP process.

Lemma IV.63 *Given the UNITY statements S1 and S2, and the total function f whose domain is the class of UNITY statements, such that the CSP process f(S1) is equivalent to the UNITY statement S1, and the CSP process f(S2) is equivalent to the UNITY statement S2, then the CSP process*

$$\mu X.(f(S1)|f(S2));X \tag{4.20}$$

is equivalent to the UNITY program (fragment)

$$S1|S2$$

Proof: The use of Corollary IV.59 and IV.60 in the proof of Theorem IV.61 ensures that the processes $f(S1)$ and $f(S2)$ are guarded and have only finite length traces. Thus Lemma IV.62 ensures that the process given by Equation 4.20 exists and is unique. What remains to be proven is that the fixed point of the contraction mapping (see Lemma IV.62) F, given by Equation 4.20, where F is defined by

$$F(X) = (f(S1)|f(S2));X$$

represents the same operator on the sequences of states of $f(S1)$ and $f(S2)$, as the UNITY weak fair choice operator | on the sequences of states of $S1$ and $S2$.

Since F is a contraction, then for any choice of an initial process X_0, (from the Contraction Mapping theorem of Section 4.3)

$$\lim_{n\to\infty} X_n = \mu X.F(X)$$

where

$$X_n = F(X_{n-1}) \qquad n \in \{1, 2, ...\}$$

Since X_0 can be arbitrary, consider the process that does nothing except terminate, that is

$$X_0 = \text{SKIP}$$

Starting with this choice for X_0, each iteration of X_i sequentially composes either $f(S1)$ or $f(S2)$ onto the current process. Thus if $f(S1)$ is chosen for the first iteration, and $f(S2)$ for the second,

$$X_1 = F(\text{SKIP}) = \text{SKIP}; f(S1) = f(S1)$$
$$X_2 = F(X_1) = F(f(S1)) = f(S1); f(S2)$$

Denote by A the predicate that is *true* if $f(S1)$ is eligible to be chosen at a given iteration, and a the predicate that is *true* if $f(S1)$ is actually chosen at a given iteration. In a similar manner the predicates B and b denote the corresponding predicates for $f(S2)$. If the iteration index n for X_n is considered as elements from the set N, which along with the partial order $\leq$ form the frame for a linear temporal logic (see Appendix A), then the CSP (deterministic) requirement that | does not allow either of its arguments to refuse being chosen is stated as

$$\Box A \Longrightarrow \Box\Diamond a$$

and

$$\Box B \Longrightarrow \Box\Diamond b$$

These two wffs implie that the construction of the sequence

$$X_0, X_1, ...X_n, ...$$

satisfies the weak fair choice requirement. ■

All of these lemmas and corollaries provide the needed pieces so that the primary result of this section, Theorem IV.64, can be proved. Lemma IV.53 states that the basic UNITY statement, the assignment of an expression evaluation to a variable, can be mapped into an equivalent CSP process. Corollary IV.59 and IV.60 extend this result to the arbitrary sequential composition of assignment components under the standard execution model, and to the general UNITY statement under the UNITY model. Theorem IV.61 completes this tact by showing that such a mapping exists for any type of UNITY assignment statement under either model. Lemmas IV.62 and IV.63 show that the 'process' that results from applying the UNITY weak fair choice operator on statements is identical to a CSP process defined recursively in terms of the deterministic choice operator. Since (neglecting the always section) any UNITY program can be constructed with these pieces, the culmination of these previous results is stated in this theorem.

Theorem IV.64 *There exists total functions f and g, whose domains are the class of UNITY programs such that given the UNITY program P executing under the standard model, the CSP process*

$$f(P)$$

is equivalent to P, and such that given the UNITY program P executing under the UNITY model, the CSP process

$$g(P)$$

is equivalent to P.

Proof: Theorem IV.61 proves the existence of f and g for any UNITY assignment statement. Lemma IV.63 finishes the proof of existence for the only UNITY operator on statements, the weak fair choice |. The uniqueness of the process given by Equation

4.20 ensures that the total functions f and g from Theorem IV.61 can be extended to include the UNITY | operator. ■

Theorem IV.64 states that any UNITY program under either execution model can be mapped into an equivalent CSP process, such that any sequence of states of the UNITY program is also a sequence of states for the CSP process. This means that the topological analysis of Section 4.3 can be applied to UNITY programs by simply mapping those programs into the complete metric space of CSP processes. Thus the following result. (Note that any UNITY program with an always section can be converted into an equivalent program without one (64))

Corollary IV.65 *Given the set U of all UNITY programs, and the total functions f and g of Theorem IV.64, then*

$$(U, d_u)$$

denotes two metric spaces, where the total function d_u, whose domain is the cross product of U with itself, is defined by

$$d_u(P, Q) = d_p(h(P), h(Q))$$

such that if the UNITY programs P and Q execute under the standard model then h is the function f, and if P and Q execute under the UNITY model then h is the function g, and d_p is defined by Corollary IV.38.

Proof: Since d_p is a metric then d_u is also a metric, and Theorem IV.64 completes the proof. ■

This corollary shows the existence of two metric spaces, one for those UNITY programs that execute under the standard model, and another for those that execute under UNITY. But once the execution model is fixed, then there is only one applicable metric space. Thus in the following sections reference is only explicitly made to one metric space of all UNITY programs, with the implicit implication that the metric space is actually one of two.

Corollary IV.65 does not explicitly state that either metric space is complete, but any metric space can be uniquely completed (305). So there exists a complete metric space that is the completion of the metric space of all UNITY programs along with this metric. Does this complete metric space contain only UNITY programs, or did the completion require entities that do not correspond to UNITY programs, just as the completion of the metric space (Section 4.2) of all finite automata required objects that were not finite automata? This thesis does not require an answer to this question, but if the answer is no, then the following theorem gives an intriguing consequent.

Theorem IV.66 *If the completion of the metric space*

$$(U, d_u)$$

from Corollary IV.65 contains only UNITY programs, then there exists a UNITY program for which there is no equivalent finite automaton, where the equivalence is defined in terms of the sequences of states from Theorem IV.64, and the sequences of words accepted by a finite automaton (as sequences of states).

Proof: The equivalence between a UNITY program and a finite automaton results from Corollary IV.65 and IV.38. That is, for a given UNITY program P, the CSP process

$$f(P)$$

is equivalent to P, with f given by Theorem IV.64. Then assume that there exists a finite automaton A such that the set of accepted words for A is equal to the set of all traces of $f(P)$, with each trace considered as a word from Σ^*, where Σ is the alphabet of the events for the process. If this assumption is not valid, then the theorem is proved. Thus A is equivalent (as a process) to the program P. Now, if the completion of the UNITY metric space contains only UNITY programs, then these programs are equivalent to some finite automaton, but Corollary IV.38, along with Equation 4.11 and 4.4 imply that any accumulation point of the one space would be equivalent to the accumulation point of the other. Since the completion of the metric

space containing A requires objects that are not equivalent to any finite automaton (as a process), then these objects are equivalent to the UNITY programs that are the accumulation points of the UNITY metric space, a contradiction. ■

This theorem implies that if the completion of the metric space of all UNITY programs contains only UNITY programs, then there exists at least one UNITY program whose execution sequence cannot be modeled using finite automata. That is, UNITY considered as a formal language would be more 'powerful' (i.e. higher up on the Chomsky hierarchy) than the regular languages. UNITY may still be more powerful as a language than the regular languages even if the completion of the metric space

$$(U, d_u)$$

doesn't contain only UNITY programs.

Thus any UNITY program can be mapped into an equivalent CSP process. This mapping is based on the idea that the UNITY weak fair choice operator on statements $|$, behaves just as the fixpoint of the CSP choice operator on guarded processes $|$, as given in Lemma IV.63. This fixpoint can be found by iterating the CSP operator (in the manner of a Picard iteration (333)) an unbounded number of times, which corresponds to the UNITY concept that the weak fair choice operator is based on an unbounded number of iterations of a choice between two statements. However, since one of the primary goals of the UNITY design is to enable the same UNITY program to be mapped into different computer architectures, including sequential and parallel architectures, then an important question is what is the relationship between the UNITY choice operator $|$ and the CSP *concurrency* operator $\|$.

The first observation regarding the mapping of a UNITY program onto an arbitrary architecture is that not all properties of the UNITY program are retained after the mapping, and conversely, not all properties of the program after the mapping were properties of the original UNITY program. For example, consider the UNITY program E1:

Program E1

initially

x,y = M,N M,N are integers

assign

y := x |

x := x-1 if x=y || x := x+1 if x≠y

end

Under the UNITY execution model, each execution of the second statement results in only one of the two boolean expressions being satisfied, and thus only one assignment is made (actually the other defaults to an empty assignment). Thus the program has no fixed point. If initially M is larger than N the value of x can increase, but eventually the first statement executes, and after that the value of x can only remain the same or decrease. The UNITY execution model implies that the value of x can decrease with no lower bound. But the program's properties change markedly when mapped onto a sequential architecture. In this case if the ordering of the assignments is:

y := x

x := x-1 if x=y

x := x+1 if x≠y

Each pass through this program leaves the value of x unchanged, and in fact this program can be considered to terminate with y=x and both equal to the initial value M. But if the mapping to the sequential architecture had reversed the order of the last two assignments, then the program would have the property that repeated execution would cause the value of x to monotonically decrease without bound.

Analysis of the original program E1 in the standard execution model, however, shows that all of these possibilities can occur, since each one corresponds to a possible property of the program over any finite execution sequence. For example, one possible execution over a finite sequence yields the same behavior for x and y as does the UNITY execution model, considering that the two assignments of the second statement can be ordered so as to behave just as they would if they executed simultaneously.

The reason for the difference in the properties of the different mappings of the program E1 is that there are named variables in one statement that are modified in another statement. Obviously, one technique to circumvent this problem is to ensure that the

UNITY program contains no global variables, so that one statement cannot modify the value of a variable used by another statement. This concept of a global or *nonlocal* variable is formalized in the following definition.

Definition IV.67 *A UNITY variable is* nonlocal *if it is named in more than one statement.*

Definition IV.67 differs from the UNITY definition of a local variable as one that is named on only one processor, and a nonlocal variable as named on more than one processor, on either the right hand side or left hand side of an assignment (64). Since the basic unit of analysis within this thesis is the statement (process), Definition IV.67 is used here.

Before progressing to Theorem IV.71 which states the relationship between UNITY programs with no nonlocal variables and CSP processes that contain the concurrency operator, the following definition and two preliminary results are needed.

Definition IV.68 *Given the programs (processes) S and T, then T* computationally simulates *S, iff for any finite sequence of states*

$$\hat{S} = (s_0, ..., s_n)$$

such that S generates $\hat{S}$ as given in Definition II.6, then there exists a finite sequence of states

$$\hat{T} = (t_0, ..., t_m)$$

where T generates $\hat{T}$ as given in Definition II.6, such that

$$s_0 = t_0$$

and

$$s_n = t_m$$

Further, S and T are computationally equivalent *iff S computationally simulates T and T computationally simulates S.*

The concept of T computationally simulating S is based on (execution) sequences of states that are finite in length, that is computational simulation can be related to actual implementations. This concept focuses on the 'computation' that a process performs, i.e. what final output (state) can be produced from a given input (initial state), and not on the details of how the computation is performed. Thus if S and T are started in the same initial state, any state that can be 'reached' by S (in finite time) can also be reached by T, although T may have reachable states that S does not. Note that if $[S]$ denotes a function 'computed' by S, such that

$$s_n \in [S](s_0)$$

then computational equivalence corresponds to Mills' concept of functional equivalence (240). This concept of computational simulation also parallels Milner's definition of simulation between programs (243).

An example of computational simulation is given in the following lemma, which is used in the proof of Theorem IV.71.

Lemma IV.69 *Given the functions f and g of Theorem IV.64, and the UNITY statements S1 and S2, then the CSP process*

$$\mu X.(h(S1)|h(S2)); X \tag{4.21}$$

computationally simulates the CSP process

$$\mu X.(h(S1); h(S2)); X \tag{4.22}$$

where h is either f, under the standard execution model, or is g, under the UNITY execution model.

Proof: The solution to Equation 4.22 is given by

$$\lim_{n \to \infty} X_n$$

where this limit is applied to the following sequence of processes

$$\begin{aligned} X_0 &= \text{SKIP} \\ X_1 &= \text{SKIP}; h(S1); h(S2) = h(S1); h(S2) \\ &\vdots \\ X_n &= X_{n-1}; h(S1); h(S2) \end{aligned}$$

Next consider the following sequence of processes that corresponds to one possible instantiation of Equation 4.21

$$\begin{aligned} Y_0 &= \text{SKIP} \\ Y_1 &= \text{SKIP}; h(S1) = h(S1) \\ Y_2 &= h(S1); h(S2) \\ &\vdots \\ Y_n &= \begin{cases} Y_{n-1}; h(S1) & \text{if } n \text{ is odd} \\ Y_{n-1}; h(S2) & \text{if } n \text{ is even} \end{cases} \end{aligned}$$

Any finite sequence of states generated by X_n for any n can also be generated by Y_m, where $m \leq 2n$, such that both sequences have exactly the same states, and thus the same initial and final states. ■

This next lemma provides both an example of computational equivalence plus the final preliminary result needed for Theorem IV.71.

Lemma IV.70 *Given the functions f and g of Theorem IV.64, and the UNITY statements S1 and S2 that contain no nonlocal variables, then the CSP process*

$$\mu X.(h(S1); h(S2)); X \tag{4.23}$$

is computationally equivalent to the CSP process

$$\mu X.(h(S1) \| h(S2)); X \tag{4.24}$$

where h is either f, under the standard execution model, or is g, under the UNITY execution model.

Proof: Consider that each state is represented by a vector from R^n such that each element of the vector represents the value of one program variable, with each such variable coded as a number. The processes X_n that correspond to Equation 4.23 (from the proof of Lemma IV.69) each (for $n > 0$) contain one or more complete and/or partial sequences of states that are generated by

$$h(S1); h(S2)$$

Since there are no nonlocal variables, then for any such partial or complete sequence of states

$$(s_0, ..., s_k)$$

there exists a subsequence of states

$$(s_{10}, ..., s_{1l}$$

where

$$s_{1i} = s_j$$

for any $i \in \{0, ..., l\}$ with $j \in \{0, ..., k\}$, with the linear order of the original sequence preserved in the subsequence, such that this subsequence of states contain state vectors that have changes in element values that correspond only to the named variables from $S1$. Within this subsequence the value of each vector only depends upon the value of the immediately preceeding vector (see Chandy and Misra, Section 3.3 (64)), and not on any of the vectors outside of this subsequence. In a similar manner there exists a subsequence of states whose state vectors have changes in element values that correspond only to the named variables from $S2$. Thus these subsequences can be arranged into any interleaving, i.e. any sequence that would result from the process of Equation 4.24, provided that the linear order of states

within each subsequence is preserved, while still preserving the same initial and final state as the original sequence. ■

With these two lemmas, the first major result relating the UNITY weak fair choice operator on statements to the CSP concurrency operator on processes can be stated.

Theorem IV.71 *Given the functions f and g of Theorem IV.64, and the UNITY statements S1 and S2 that contain no nonlocal variables, then the CSP process*

$$\mu X.(h(S1)|h(S2));X$$

which is equivalent to the UNITY program (fragment)

$$S1|S2$$

computationally simulates the CSP process

$$\mu X.(h(S1)\|h(S2));X$$

where h is either f, under the standard execution model, or is g, under the UNITY execution model.

Proof: Computational simulation is transitive, since if R computationally simulates S, then for any sequence

$$(s_0,\ldots,s_k) \tag{4.25}$$

of S, there exists a sequence

$$(r_0,\ldots,r_j) \tag{4.26}$$

of R, such that

$$s_0 = r_0 \quad \text{and } s_k = r_j$$

If S computationally simulates T, then for any sequence

$$(t_0,\ldots,t_l) \tag{4.27}$$

of T, there exists a sequence of S, say the one from Equation 4.25, such that

$$s_0 = t_0 \quad \text{and } s_k = t_l$$

This implies that for any sequence of T, say the one from Equation 4.27, there exists a sequence from R, here given by Equation 4.26, that satisfies the computational simulation requirement, and R computationally simulates T. Since Lemma IV.70 implies that

$$\mu X.(h(S1); h(S2); X$$

computationally simulates

$$\mu X.(h(S1)|||h(S2)); X$$

then Lemma IV.69 along with the transitivity of computational simulation completes the proof. ■

Thus the mapping of the fair choice combination of two UNITY statements without non local variables into a pair of concurrent processes, may result in the loss of sequences of states, but will not introduce any new ones. This means that for a UNITY program containing the construct

$$S1|S2$$

with no nonlocal variables, the mapping to the CSP process

$$\mu X.(h(S1)|||h(S2)); X \tag{4.28}$$

will yield equivalent sequences of states if those sequences that result from

$$\mu X.(h(S1)|h(S2)); X$$

which are not possible for the process of Equation 4.28, are neglected.

4.5 Summary

A *complete metric space* is a topological space that satisfies certain constraints relating to the metric, which is a function that measures the 'distance' between pairs of elements of the space, and to the concept of completeness, which means that certain elements are included in the space. This chapter demonstrates that different types of computational models can be analyzed as complete metric spaces: finite automata, CSP, and UNITY.

Not only can these three models be cast as metric spaces, but the metric used in all three is based on a metric developed by other researchers for the metric space whose elements are strings of finite length formed from a finite alphabet of symbols. (If the alphabet is denoted by Σ, then these symbol strings are the elements of Σ^*) If the symbols are the decimal digits and the decimal point, then the completion of this metric space corresponds to the nonnegative real numbers. Although the basic concept is not new, the term *Turing computable complete metric space* is defined here, as the completion of this metric space using only the computational capabilities of Turing machines. This results in a space whose elements correspond to that subset of the nonnegative real numbers called the *Turing computable numbers* (see Appendix B). A new result is proved which shows that this Turing computable complete metric space also satisfies a major formulation (stated for the complete metric space in the paper) from the original paper by Scott on the mathematical basis of computation (311). Another new result shows that this Turing complete metric space satisfies the definition of a *theory* (based on categories) given in the previous chapter.

Based on the metric space of symbol strings, another metric space (which is itself not new) is presented whose elements are finite automata. Unifying several different ideas from the literature, the *binary automaton tree* is defined. The binary automaton tree permits the metric from the space of symbol strings to be used to define the metric on finite automata, and also permits the definition of *canonical* forms of finite automata, so that if any two of these canonical forms represent equivalent automata, then they are identical. The major new results are that for any (Turing) recursive set there is a 'machine' that is an element of the completion of this metric space of finite automata that *accepts* the set, and that these machines are no more powerful (computationally) than Turing machines.

The metric space of symbol strings is also used to define a metric space of *deterministic* CSP processes, using the binary automaton tree concept and the analogy between finite strings of symbols and finite length traces. Since to date all of the analysis of CSP processes is based on the mathematics in the text by Hoare (165), all of the topological analysis here is new (the analysis is new, some of the results are not). The first major result is that the CSP concept of a *continuous* function from processes into processes is equivalent to the metric space concept of a continuous function. Additional results are that the metric space based *contraction mapping* is equivalent to the CSP *constructive* function, and that the metric space of deterministic CSP processes is complete. These lead to the conclusion that what Hoare termed the fundamental theorem of deterministic processes is equivalent to the standard contraction mapping theorem for complete metric spaces. An interesting consequent of this equivalence between the CSP mathematical analysis and the topological analysis, is that new results for CSP can be generated by simply rewording existing theorems from the topological theory of complete metric spaces.

The chapter concludes with another new application of the topological theory of complete metric spaces, to the analysis of UNITY programs (specifications). The metric for UNITY programs is defined by showing that any UNITY program can be mapped into an equivalent CSP process, where the equivalence is based on the two exhibiting identical behavior. This approach is valid because all UNITY programs are deterministic (the CSP meaning of deterministic), so that the mapping only need be into deterministic CSP processes. Thus the metric on UNITY programs is defined using the metric on deterministic CSP processes. Since UNITY requires the simultaneous execution of assignment components within a single assignment statement, whereas CSP forces the sequential execution of atomic actions, this mapping equates UNITY assignment statements (which are executed sequentially) with CSP atomic actions (the symbols of the traces). The fact that UNITY programs generate unbounded execution sequences, versus the finite length traces of CSP processes, is treated by showing that the UNITY programs map into processes that are the fixed points of recursive definitions. New results are presented that link the computational hierarchy of machines based on finite automata with the 'computational power' of UNITY programs, and that relate the properties of UNITY programs without 'global variables' to

the equivalent CSP processes.

V. *Computational Temporal Semantics*

The last chapter demonstrated that computational models can be viewed topologically as complete metric spaces, a development motivated by the analysis of assignments and type transformations as continuous mappings. But this analysis treated the programs from a purely syntactical viewpoint, that is the analysis was performed on strings of symbols without any concern for the meaning of the symbols. Consequently, this chapter continues the analysis of the computational models, but with respect to the semantics, or intended meaning, of the symbol strings comprising the programs. This chapter addresses this semantic analysis using the tools of the temporal logic of Appendix A. Although this type of approach is not new (38, 75, 252), applying it to the proof of correctness of CSP programs in the manner of Section 5.3 is new.

Section 5.1 presents a brief overview of the three major classifications of formal semantics, the *operational*, *denotational*, and *axiomatic*, along with an introduction to how the temporal logic can be used in semantic analysis. Section 5.2 presents an introduction to the temporal analysis of finite automata based on the temporal logic of Appendix A. Since the finite automaton is the most basic model of computation, this temporal logic is introduced first. Although temporal reasoning about finite automata is not new (114, 113), this presentation of a formal temporal logic for finite automata is. Section 5.3 then extends the finite automaton based logic into a temporal logic for CSP processes, a computational model more powerful than finite automata (165). Section 5.3 demonstrates that this temporal logic for CSP supplies the framework within which to reason about proofs of correctness of CSP programs, an alternative to the approach used in Hoare's book (165). Although the previous chapter presented the topological analysis of finite automata, CSP, and UNITY, this chapter only covers finite automata and CSP, since the book by Chandy and Misra contains the basic temporal analysis for UNITY programs (64). However, even without the Chandy and Misra analysis, UNITY programs could be addressed by first mapping them into their CSP equivalents using the results from Section 4.4, and then applying the concepts from this chapter to the equivalent CSP programs.

5.1 *Program Semantics*

For a given program, if P denotes the formal syntactic representation of the program (such as the source code listing), then from Definition II.10 the function

$$P \rightarrow \mathcal{M}(P)$$

can be considered as the semantic analysis of the program, where $\mathcal{M}(P)$ denotes a formal representation of the meaning of the program, often as some type of mathematical model (201). Based on the structure of this mathematical model $\mathcal{M}(P)$, semantic analysis has been classified into three major groups, known as operational, denotational, and axiomatic (324):

Operational: Elements of $\mathcal{M}(P)$ are program states (see Definition II.6), where each program is represented by one or more sequences of these states called *execution sequences*. Each state is considered a mapping from program variables and other symbol strings into their values. Thus a program is mapped into a sequence that completely specifies the values of all program variables during execution of the program. A major thrust of the analysis is specifying mathematically the relationships between successive program states, with the temporal characteristics modeled by the linear order of the sequences of states (339, 233, 171). Also called *behavioral* semantics (201).

Denotational: An approach that is credited to the work of Scott (311), Strachey (314), and McCarthy (231), the elements of $\mathcal{M}(P)$ consist of functions and constants that can generate the sequences of program states given the initial state (the first state in any sequence representing a program). Equivalently, programs are represented as relations whose domain is sets of initial program states, and whose codomain is sets of final states, in what is called the input/output semantics (93, 209, 48, 323, 337, 140, 329). Includes the *action* semantics of Lamport (201). The term 'denotational' is sometimes used in conjuction with the functional representation only, whereas *predicative* semantics refers to the relational representation (50). See the article

by Bakker and Zucker (94) for a relationship between metric spaces, denotational semantics, and concurrent programs.

Axiomatic: Here the elements of $\mathcal{M}(P)$ include axioms about the statements of the original program, along with rules of inference to permit reasoning about the program. Because of this ability to reason about the program, early work with program verification (both sequential and concurrent programs) was based on axiomatic semantics, with the temporal characteristics of the program modeled with flowcharts (126, 161, 217, 86, 211, 344, 144, 200). One of the first approaches applied to parallel programming, probably because of the widespread acceptance of the 'Hoare' semantics of sequential programs (266).

As an example of these three types of semantics, consider the following program expressed in UNITY notation, with | denoting the fair selection operator.

Program P

assign coin := H | coin := T

end $\{P\}$

Thus program P represents an unbounded number of fair coin tosses, with the outcome of each toss represented by the assignment of H or T to the variable coin. The operational semantics of P can be given by (based on the CSP formalism from Section 4.3)

$$traces(P) = \{t | t \in \{H, T\}^* - \{\Lambda\}\}$$

Here the operational semantics are represented by the set of all possible sequences of program states, where each program state is denoted by the value assigned to the variable coin. These sequences are represented using trace notation, so that the set of all possible sequences of program states is denoted by the set of all traces, that is the set $traces(P)$.

One choice for the denotational semantics will be some kind of model that can generate this set of all possible program states, that is a denotation that represents the ability to generate this set of all traces. One such possibility for the denotational semantics is given by

$$P = (H \rightarrow P | T \rightarrow P)$$

where the formalism is based on the CSP from Section 4.3.

The last aspect of this example is a representation for the axiomatic semantics. Since the tradtional goal of axiomatic semantics has been to facilitate proofs of correctness, consider the proof that program P does generate all possible finite strings of H's and T's. One approach would be to represent how each program statement changes any existing string that represents the history of assignments to coin, from some starting time until the present. The variable str denotes this history, temp denotes a dummy variable, and cons denotes the function that concatenates a single symbol onto the end of a string.

{temp = str} coin := H {str = cons(H,temp)}

{temp = str} coin := T {str = cons(T,temp)}

This syntax is typical of the axiomatic approach, and has been credited to Hoare (161). It's based on assertions of the form

$$\{P\}S\{Q\} \tag{5.1}$$

where P is called the *precondition* of the program statement S, and Q is called the *postcondition*. This assertion is interpreted as 'if P is true before S executes, then after S terminates Q will become true (if S terminates). Note that this concept is based on the earlier techniques of Floyd that used flowcharts, with each node representing a program statement and the arcs representing flow of control (126). Thus Hoare's precondition and postconditions corresponded to labeling the arcs with these predicates, which was the basic approach to proving program properties until Hoare formalized the technique using axiomatic semantics (217).

Hoare's approach depended on a set of inference rules that allowed the derivation of assertions about the program. These rules gave assertions of the form of 5.1 for certain program statements such as assignment and iteration, and for the composition of statements. Unfortunately, Hoare's analysis was based on the idea of 'if the statement S terminates', which meant that total correctness did not follow implicitly. This was corrected by Manna and Pnueli (218) with the introduction of the nomenclature

$$(P(x)|S|Q(x,y)).$$

The interpretation is that for every x, where x is the initial value of the program variables, if $P(x)$ holds before S executes, then S will terminate with $Q(x, y)$ true, where y is the resulting value of the program variables. Just as Hoare did, Manna and Pnueli give the rules of inference for deriving assertions based on a set of program statements.

This thesis utilizes mainly the operational and denotational semantics, which, since the emphasis here is mostly with design and implementation, seems consistent with the idea expressed by Stoy that (Stoy emphasized that this statement is an oversimplification, but "not so far from the truth") "the operational method is likely to be of most value to the implementer, the axiomatic to the programmer, and the denotational to the language designer" (324).

Another broad categorization of semantics, and one that can include examples from either of the above two groups, is the *compositional* semantics. A compositional semantics is one such that given the program statements s and t, and their composed statement s, t, then the evaluation of

$$\mathcal{M}(s;t)$$

is completely determined by the two evaluations of $\mathcal{M}(s)$ and $\mathcal{M}(t)$ (201).

In addition to modeling the program itself, temporal logic can be used to prove certain statements (assertions) about the behavior (operational) of the program, or about the relation (denotational) that represents the program. These assertions can be either static, which means their evaluation to *true* or *false* is not dependent upon when the evaluation is performed (time independent), or else such evaluation does depend upon time, in which case they are called temporal assertions (see Appendix A). For this thesis, all assertions about programs are considered as temporal, so that those assertions that would normally be considered static are simply *true* or *false* for all relevant time. Temporal assertions about programs are also called *program properties*, since they make assertions about certain properties of the temporal structure of the program. For example, *invariant* properties (286, 64) have assertions of the form

$$\Box P(s)$$

where s is an element of the set of program *states*. Since program states can be considered as a mapping from the program variables into their values, the assertion $P(s)$ can be considered a function of the program variables, their values, and the last executable statement. The assertion $P(s)$ does not explicitly include a time variable, instead the implicit dependence on time is stated by the temporal operator $\Box$, which is the *always* or *henceforth* operator.

An example of an invariant property is partial correctness, which can be stated as, it is always true from now on (the $\Box$) that if a certain assertion holds at the start of the program, then another assertion (the termination assertion) always holds. Another terminology for these type of properties is *safety* properties, since they are based on the concept that 'no bad things will happen' (147). Another example of an invariant is *clean behavior*, which states that executing an instruction does not produce a fault or undefined results (219). Mutual exclusion and freedom from deadlock (which is addressed in Chapter VI) are also considered as invariant or safety properties (219). One of the major advancements in proving program properties was the development of an axiomatic theory that can be used to prove safety properties of concurrent programs (268, 265). The theory is based on the concept of shared variables and monitor-like (monitors are credited to Hansen (149) and Hoare (163)) control of these variables.

The second class of properties are called *liveness* properties, which can have the following (and others) forms:

$$\Diamond P(s)$$

$$\Box\Diamond P(s)$$

$$\Diamond\Box P(s)$$

The idea behind liveness properties is that if the current state of the computation does not satisfy the property, then eventually the computation will reach a state that does satisfy the property. This is the reasoning behind the statement by Pnueli that liveness properties are satisfied by infinitely many finite prefixes of an infinite computation (287). Liveness

properties include a class of properties that satisfy the *leads to* operator $\sim$ (200), that is these properties have the form

$$\Box(P(s) \sim Q(s))$$

Whereas partial correctness is classified with invariant properties, total correctness can be considered one such 'leads to' property. This results from the interpretation of total correctness as, if some assertion holds at the start of the program, then eventually another assertion will hold when the program terminates, since the program always terminates. Liveness properties embody the concept that 'good things will happen' (147). Another liveness property is called *intermittent assertions*, based on the intermittent-assertion statements of Manna and Waldinger (223), which state than whenever a certain assertion is satisfied during the execution of a program, then eventually another specific assertion will become *true*. *Accessibility* is a liveness property that ensures certain program statements will execute an unbounded number of times in unbounded execution sequences, while *responsiveness* states that in unbounded execution sequences that are modeled after the consumer-producer paradigm, no consumer can be left unsatisfied indefinitely.

A third class of program properties are the *precedence* properties, which are based on the until operator (see Appendix A), that is assertions of the form

$$P(s) \mathcal{U} Q(s)$$

which states that the predicate Q will eventually become *true*, and until that time the predicate P will remain *true*. Manna and Pnueli clasify a type of precedence properties as *safe liveness*, in which the predicate P represents a safety property that is to remain *true* until the liveness property Q is satisfied. They call this "nothing bad happens until something good happens", and "strongly suggest that a full specification of a program should always be expressed as an until expression" (219).

One important class is the *stable* properties, which have the formulation

$$P(s) \Longrightarrow \Box P(s)$$

These properties state that if P holds at any point during the execution of the program, then P will hold continuously from that point on. An important subclass of the stable properties are the *quiescent* properties, which include termination and deadlock (63). The quiescent properties embody the concept that a program (or process) is *idling*, that is, not achieving useful progress (65).

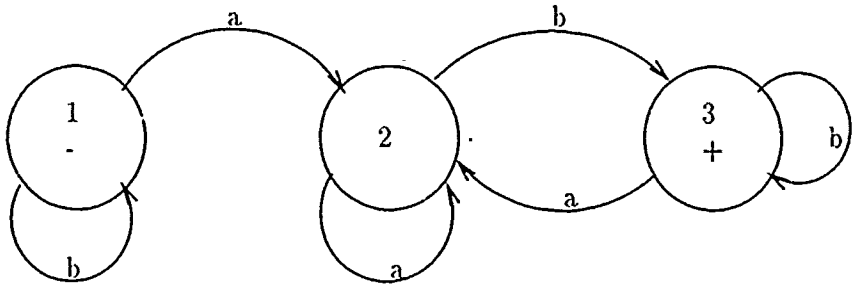

Figure 5.1. Finite Automaton M

5.2 Temporal Logic of Finite Automata

This section introduces the analysis of computational models using the concepts of temporal logic from Appendix A. Since the finite automaton is the simplest computational model addressed in this thesis, this temporal analysis begins with the finite automaton before progressing into the CSP model in Section 5.3. (Section 4.2 defines and describes finite automata)

Consider the finite automaton M of Figure 5.1 (repeated from Section 4.2). This automaton accepts all words from the set

$$accept(M) = \{ab, aab, bab, ...\}$$

where $accept(M)$ can be well ordered by length (since it's a recursive set). Next consider a subset of M, denoted by N,

$$N = \{bab, babab, bababab, ...\} = \{b(ab)^n | n \geq 1\} \tag{5.2}$$

For any two elements (words) of N, one of them will always be the prefix of the other, that is,

$$(s \in N \wedge t \in N) \Longrightarrow \exists u[u \in \Sigma^* \wedge (su = t \vee tu = s)] \tag{5.3}$$

where Σ denotes the alphabet of symbols, and $u = \Lambda$ (the empty word), if $s = t$.

Consider two arbitrary elements from the set N, say

$$s, t \in N$$

and the binary relation

$$R \subset N \times N$$

such that

$$sRt \iff \exists u[u \in \Sigma^* \wedge su = t]$$

This relation R is reflexive, since

$$s\Lambda = s$$

and thus sRs holds for any element s of N. R is transitive, since

$$(rRs \wedge sRt) \Longrightarrow \exists u \exists v[ru = s \wedge sv = t]$$

implies that

$$ruv = t \Longrightarrow rRt$$

since uv is just another word from Σ^*. A slight variation of this argument shows that R is antisymmetric, since

$$(sRt \wedge tRs) \Longrightarrow \exists u \exists v[su = t \wedge tv = s]$$

which implies that

$$tvu = t$$

or that

$$v = u = \Lambda$$

and thus

$$s = t$$

So R is a partial order, and further, is a linear order, since Equation 5.3 implies that

$$(s \in N \wedge t \in N) \Longrightarrow (sRt \vee tRs)$$

For any word from N, the state transitions that occur for each symbol can be considered as occurring at equally spaced intervals of time, with the first symbol (leftmost) representing a transition at a time instant called the 'present'. Thus any such word is a representation of a temporal history of state transitions, and for any two words from N, one history will include the other. This suggests that the elements of N are the interpretations associated with the frame (N, R) of a temporal logic (since R is a partial order), and indeed a linear temporal logic because R is a linear order on N (see Corollary A.9). Thus reasoning about the states of the finite automaton M can be carried out, and expressed in terms of, the temporal logic of Appendix A.

Although this analysis so far has only addressed sets of the form of N, the concept can be extended to the complete set $accept(M)$. This follows from the following lemma, which is a summary of certain paragraphs from Section 4.2.

Lemma V.1 *Given any finite automaton M, the set $accept(M)$ is the countable union of sets S_i, that is*

$$accept(M) = \bigcup_{i \in N} S_i$$

such that for each i there exists a linear order R_i of the elements of S_i, given by

$$(s \in S_i \wedge t \in T_i) \Longrightarrow (sR_it \Longleftrightarrow \exists u[u \in \Sigma^* \wedge su = t])$$

Proof: Follows directly from construction of binary automaton trees (see Section 4.2), in that every finite automaton corresponds to an equivalent finite binary automaton tree. The sets S_i correspond to the different 'paths' down through the tree starting from the root node, and ending at an accepting node. As each path is traversed, each traversal retracing the nodes of the previous traversal and then continuing on to an accepting node, the sequence of words formed (each one a prefix of the next) by these traversals contains the elements of the corresponding set S_i. The countability of

these different paths follows from the finiteness of the tree. (As opposed to 'infinite' trees, which could yield an uncountable number of paths) ■

Thus each pair (S_i, R_i) forms a linear temporal logic, which leads to the following theorem.

Theorem V.2 *For any finite automaton M, there exists a partial ordering R of the elements of*

$$accept(M)$$

such that

$$(accept(M), R)$$

forms the frame for a branching temporal logic.

Proof: The partial order R is formed by combining the linear orders R_i of Lemma V.1. Thus the branching temporal logic follows from

$$accept(M) = \bigcup_{i \in \mathrm{N}} S_i$$

along with the corresponding R_i, and the functions

$$\Phi_i : S_i \rightarrow \mathrm{N} \times \mathrm{N}$$

defined by

$$(s \in S_i \wedge t \in S_i) \Longrightarrow (sR_it \Longleftrightarrow \Phi_i(s) \leq \Phi_i(t))$$

where the inequality is one induced on ordered pairs of natural numbers, that is

$$(a, b) \leq (c, d) \Longleftrightarrow (a = c \wedge b \leq d)$$

The collection of the Φ_i functions defines the function

$$\Phi : accept(M) \rightarrow 2^{\mathrm{N} \times \mathrm{N}}$$

such that

$$s \in accept(M) \Longrightarrow ((n,m) \in \Phi(s) \Longleftrightarrow (n,m) = \Phi_n(s))$$

as required in Definition A.10. ■

Although these concepts can be extended to the set Σ^*, not just restricted to $accept(M)$, this effort deals only with the set of accepted words, based on the idea that a successful computation corresponds to an accepted word (165) (in reference to processes as languages).

The assertion

$$(n,m) = \Phi_n(s)$$

is based on the idea that for one path through the binary automaton tree, denoted as path (or branch) n, the string s is the prefix of any (different) string t, such that

$$(n,k) = \Phi_n(t)$$

with $k > m$. For a given s, $\Phi(s)$ evaluates to a set because of the possibility that a given string could be part of more than one branch (path). Consider the string bab, which is contained in the two branches represented by the set N of Equation 5.2 and the set

$$\{(bab)^n | n \geq 1\}$$

which is also a subset of the set $accept(M)$.

The branching temporal logic associated with any finite automaton permits temporal reasoning about sequences of states associated with strings of symbols that form the 'input' words to the automaton. Thus temporal reasoning about the states of the automaton is possible in a sort of indirect manner. Contrast this with a temporal logic based directly on the states. Such a temporal logic would require a partial order on the states themselves. Consider the attempt to define this partial order in an intuitive manner based on the arcs, that is

$$sRt \Longleftrightarrow \exists f_i[(s,t) \in f_i]$$

where s and t are arbitrary states of the automaton, and f_i is one of the functions that comprise the transitions of the automaton. This relation states that sRt, if and only if there exists an arc whose head is s, and whose tail is t. Unfortunately, it is easily shown that this R is not a partial order, since any (nontrivial) cycles cause R to violate the antisymmetry property (208). Thus an R defined in this intuitive manner cannot form the basis for a temporal logic.

With respect to Lemma V.1, the countability (versus finiteness) of the sets S_i can be shown using the finite automaton of Figure 5.1. A countable collection of subsets of $accept(M)$ consists of

$$\{b(ab)^n | n \geq 1\}$$
$$\{bb(ab)^n | n \geq 1\}$$
$$\vdots$$
$$\{(b)^k(ab)^n | n \geq 1\}$$
$$\vdots$$

where k is any natural number. And each one of these subsets can be linearly ordered using the appropriate R_i of Lemma V.1.

5.3 Temporal Logic of CSP

Given that CSP models of computation can be analyzed using the mathematical tools of complete metric spaces (see Section 4.3), a goal of this section is to show that the sat operator, used in proving assertions about programs, is another denotational tool for reasoning about programs using the modal logic of Appendix A. Thus statements about programs that invoke the sat operator can be equivalently reformulated using the symbology from the modal logic, given that there is a direct correspondence between the modal logic concept of an interpretation and the CSP concept of a trace. Note that there exist other approaches to include the temporal logic concept into CSP, such as the Real Time CSP (RT-CSP) (351).

Given a process P and a wff W, then

$$P \text{sat} W$$

read as 'P satisfies W', denotes that any possible behaviour of P, that is any possible trace of P, implies that the wff W evaluates to *true*. The formal definition states that if s is any trace of P, then

$$s \Longrightarrow W$$

where the wff W may or may not have s as a variable (165). Hoare develops the following initial set of *laws* (axioms) governing the sat operator: (Nomenclature is Hoare's)

L1

$$P \text{sat} true$$

L2

$$\text{If } P \text{sat} S$$
$$\text{and } P \text{sat} T$$
$$\text{then } P \text{sat}(S \wedge T)$$

L3

If $P \text{sat} S$

and $S \Longrightarrow T$

then $P \text{sat} T$

Since the concept of the sat operator is to show that a wff evaluates to *true* for any arbitrary trace of the process, this suggests that given any process P, a logic model $(traces(P), R, V)$ can be formed, where $traces(P)$ denotes the set of all of the possible traces of the process P. The set V contains those wffs which P satisfies by assumption (i.e. without requiring a formal proof), and the set R reflects the type of modal logic, either temporal or predicate (see Appendix A), used to prove that a process satisfies one or more wffs. One such 'assumed' wff that is an element of V for any process is *true*, which is given as the law L1. The other elements of V will depend upon the particular process chosen, such as for the process STOP (which does nothing), for which V contains the wff 'the trace of the process is empty' (Hoare's L4).

Thus the different traces of the process P form the elements of the set of interpretations within the modal logic $(traces(P), R, V)$, so that each trace is a different interpretation. A consequent of Definition A.2 then, is that

$$\models_s W$$

is *true*, if and only if, given some process, the assertion denoted by W evaluates to *true* given the trace s of this process. Given a one-to-one correspondence between a trace of a process and a *computation*, then this definition corresponds to the definition given with respect to denoting correctness within the temporal logic programming language *Tempura* (148). A consequence of this definition is that there may exist another trace t for this same process such that

$$\models_t W$$

does not hold. Since for any process P the set V in any model $(traces(P), R, V)$ used in

conjunction with this analysis of CSP processes will remain fixed (since it's the set of wffs that hold for the process by decree), then the nomenclature

$$\models W$$

can also denote that the assertion W is valid for a given process, which agrees with the nomenclature used for the Tempura programming language (see Definition A.4).

Since a process satisfies a wff if the wff evaluates to *true* for every trace, and a wff is true (see Appendix A) if it evaluates to *true* for every interpretation, then it follows from 'traces as interpretations' and Definition A.2 that

$$P \mathsf{sat} W \Longleftrightarrow \models W$$

The notation

$$\models_s W$$

where s is a trace of the proocess P will be denoted by

$$P \models_s W$$

whenever needed to prevent ambiguity.

As a result of this equivalence between traces and interpretations, the statement 'P satisfies W' is equivalent to saying that within the modal logic defined by the traces of P, W is true. Extending the concept from single processes to concurrent processes using the $\|$ operator does not change this 'traces as interpretations' idea. Consider the two processes P and Q that form the concurrent process $P\|Q$. Any trace of the concurrent process is an interpretation of the concurrent process, while such a trace restricted to either the alphabet of P or the alpabet of Q is an interpretation of either process P or Q respectively. Symbolically,

$$(P \models_s S \wedge Q \models_t T) \Longrightarrow P\|Q \models_q S \wedge T$$

if and only if the trace q restricted to the alphabet of P equals the trace s, and the trace q

restricted to the alphabet of Q equals the trace t. This is another formulation of Hoare's law regarding the specification of concurrent processes (see L1 in Section 2.7 of (165)).

Given that the concept of sat can be recast as a modal logic, each of the three laws L1 - L3 above can be derived from Definition A.2. The derivation of L1 is given by: Definition A.2 states that for any s,

$$\neg \models_s false$$

and that

$$\models_s \neg w \iff \neg \models_s w$$

which taken together imply that

$$\models_s true$$

Since this holds for any s, then

$$\models true$$

or

$$P\text{sat}true$$

which is L1. In a similar manner follows the derivation of L3: Definition A.2 states that

$$\models_s (v \Longrightarrow w) \iff (\models_s v \Longrightarrow \models_s w)$$

which implies that

$$(\models_s S \wedge S \Longrightarrow T) \Longrightarrow \models_s T$$

or

$$((P\text{sat}S) \wedge (S \Longrightarrow T)) \Longrightarrow P\text{sat}T$$

which is L3. This derivation uses the concept that Hoare's

$$S \Longrightarrow T$$

is equivalent to stating that

$$\models_s (S \Longrightarrow T)$$

for any trace s. The derivation of L2 uses the relationship between the $\wedge$ and the $\Longrightarrow$ operators from Appendix A, which is given by

$$\neg(a \Longrightarrow \neg b) \Longleftrightarrow (a \wedge b)$$

and leads to the derivation: Given the premise of L2

$$P\text{sat}S \wedge P\text{sat}T$$

which is equivalent to (for any s)

$$\models_s S \wedge \models_s T$$

then

$$\neg(\models_s S \Longrightarrow \models_s \neg T)$$

(since $\neg \models_s T$ if and only if $\models_s \neg T$) which is equivalent to

$$\neg \models_s (S \Longrightarrow \neg T)$$

which is equivalent to

$$\models_s \neg(S \Longrightarrow \neg T)$$

and to

$$\models_s S \wedge T$$

which is the consequent of L2, that is

$$P\text{sat}(S \wedge T)$$

An interesting extension to the CSP presentation would be to add a law that follows from the inclusion of the modal operator $\Box$ in Definition A.2, that is

$$\models_s \Box w \iff \forall t[(t \in S \wedge (s,t) \in R) \Longrightarrow \models_t w]$$

where S is the set of interpretations, which for CSP would be the set $traces(P)$ for the process P, and where

$$R \subset traces(P) \times traces(P)$$

A literal translation of this modal operator statement into the nomenclature of CSP would be

$$(s \in traces(P) \Longrightarrow \Box w) \iff ((s,tr) \in R \Longrightarrow w(tr))$$

where tr is any trace of the process P, and $w(tr)$ denotes that w evaluates to $true$ given the trace tr.

Returning to the frame that this analysis has been based on, $(traces(P), R)$, for some process P, what is the nature of R? Although the analysis required no specific R, the choice of an R that follows from the metric on traces given by Equation 4.11 yields an interesting result. Consider the requirement that the order satisfy the following two properties:

R1 $\pi(s,t) = 0 \iff (sRt \wedge tRs)$

R2 $\pi(s,u) \leq \pi(s,t) \iff (sRt \wedge tRu)$

where s, t, and u are arbitrary processes generated by a given process P, and π is the metric from Equation 4.11. The first property states that the distance between two traces should be zero if and only if the two traces would be equal if the relation were antisymmetric. The second property summarizes the idea that if the relation were transitive, then when compared to a fixed trace, traces that are 'deeper into' a given chain with respect to the relation would also be 'closer to' the fixed trace with respect to the metric, and vice versa. At least one such order on traces exists, and is given by the following definition for R.

$$sRt \iff \exists u[su = t] \tag{5.4}$$

where s, t, and u are traces associated with a given process P. Thus one trace is 'less than or equal to' another trace if and only if it's a prefix of the other trace, which is exactly the idea behind the partial order used in the temporal analysis of finite automata in Section 5.2. Note that with this order, the set of traces formed by following one path 'down' the binary trace tree is linearly ordered by R. That this relation satisfies the first property follows from the fact that R is antisymmetric (as shown in Section 5.2). R also satisfies the second property, since

$$(sRt \wedge tRu) \Longleftrightarrow \exists v \exists w[sv = t \wedge tw = u]$$

which also means that

$$sv = t$$

$$svw = u$$

or equivalently,

$$0 \leq \pi(s,u) \leq \pi(s,t) \leq 1/2^k$$

where k is the length (in symbols) of the trace s. Note that this argument also shows that R is transitive, since

$$\exists v \exists w[svw = u] \Longleftrightarrow sRu$$

The reasoning of Section 5.2 implies that R is reflexive, and thus R is a partial order. But R is not a linear order, since for the process P such that

$$traces(P) = \{\langle\rangle, \langle 0\rangle, \langle 1\rangle\}$$

it follows that

$$(\langle 0\rangle, \langle 1\rangle) \notin R \wedge (\langle 1\rangle, \langle 0\rangle) \notin R$$

Theorem V.3 *Given a process P, then the modal logic based on the frame $(traces(P), R)$, where R is any relation that satisfies the two properties* R1 *and* R2 *(such as the logic used*

in deriving the sat *equivalences), is a temporal logic (see Definition A.6). Further, such a logic is a branching temporal logic (see Definition A.10).*

Proof: As just shown, such an R is a partial order, thus $(traces(P), R)$ forms the frame for a temporal logic. That this temporal logic is also a branching temporal logic follows from the constructive function whose domain is the class of all processes and whose codomain is the class of all binary trace trees (see Section 4.3). For any two traces

$$s = \langle s_0, s_1, ..., s_n \rangle$$
$$t = \langle t_0, t_1, ..., t_m \rangle$$

where n does not necessarily equal m, the function Φ from Definition A.10 satisfies the following

$$\{(i,0),(j,0)\} \subset \Phi(\langle\rangle)$$
$$\{(i,k),(j,k)\} \subset \Phi(\langle s_0, ..., s_k \rangle) \Longleftrightarrow (l \leq k \Longrightarrow s_l = t_l)$$

and for any single trace

$$u = \langle u_0, u_1, ..., u_o \rangle$$

Φ satisfies

$$(h,p) \in \Phi(\langle u_0, ..., u_p \rangle)$$

where i, j, and h are the disjoint *branch* numbers associated with the three traces s, t, and u, and

$$0 \leq k \leq \min(\{n,m\}) \qquad 0 \leq p \leq o$$

Since all traces are finite in length, this completely defines the function Φ. ■

That these logics are branching temporal logics can be demonstrated with the following example. Consider the process P defined by the formula shown in figure 5.2. From

$$P = (0 \rightarrow (1 \rightarrow STOP | 0 \rightarrow STOP))$$

Figure 5.2. Example Process P Formulation

this formula it follows that

$$traces(P) = \{\langle\rangle, \langle 0\rangle, \langle 0,0\rangle, \langle 0,1\rangle\}$$

Applying the relation R defined by Equation 5.4 yields

$$\langle\rangle R\langle 0\rangle \qquad \langle 0\rangle R\langle 0,0\rangle \qquad \langle 0\rangle R\langle 0,1\rangle$$

These 'relationships' demonstrate that each path thru the binary trace tree generates traces that constitute one 'branch' of the branching temporal logic associated with the frame $(traces(P), R)$. One such branch is given by

$$\langle\rangle, \langle 0\rangle, \langle 0,0\rangle$$

while the other branch comprises

$$\langle\rangle, \langle 0\rangle, \langle 0,1\rangle$$

To relate these branches to the function Φ given in Definition A.10, consider that

$$\Phi(\langle\rangle) = \{(1,0),(2,0)\}$$
$$\Phi(\langle 0\rangle) = \{(1,1),(2,1)\}$$
$$\Phi(\langle 0,0\rangle) = \{(1,2)\}$$
$$\Phi(\langle 0,1\rangle) = \{(2.2)\}$$

which further demonstrates the existence of two 'branches' for the branching temporal logic associated with the process P.

5.4 Summary

The *syntax* of a program refers to the rules that govern how the symbols can be combined to form the symbol strings that comprise the program. The *semantics* of a program refers to some type of 'meaning' credited to the symbol strings, although this meaning is often defined using another syntax, such as the syntax of the temporal logic. The temporal logic can be used in the semantic analysis of programs by allowing a compact denotation for *program properties.* Program properties are simply assertions about programs that can be expressed using the syntax of the temporal logic, where the predicate symbols within the temporal formulas represent statements regarding the program *states.* A program state, as used extensively in the next chapter, is a representation of all of the values of the program variables at an instant of time.

This chapter demonstrates that the temporal logic (see Appendix A) can be used for reasoning about, and developing proofs for, program properties of 'programs' that are represented with the computational models used in this research. This application of the temporal logic is first shown for finite automata, and while the basic concept is not new, the specific presentation given here is. The proof that the program properties of finite automata can be represented using the *branching temporal logic* is based on the same concepts used in the previous chapter to develop the complete metric space of finite automata, further emphasizing the unifying ideas presented by this research.

Besides finite automata, the other two computational models addressed by this effort are CSP and UNITY. Since the last chapter demonstrated that any UNITY program can be mapped into an equivalent (equivalence of observed behavior) deterministic CSP process, this chapter only shows that temporal logic can be used for semantic analysis of CSP programs (processes). Although temporal logic has been used in conjunction with CSP, the approach presented here is new. This approach is to show that the CSP operator sat, used in proofs of program properties, can be equivalently expressed using the symbology of the branching temporal logic. The presentation utilizes the same concepts used in developing the branching temporal logic for finite automata, and demonstrates that the 'proof rules' given for CSP using the sat operator have equivalent parallels in the symbology of the branching temporal logic.

The previous chapter demonstrates that three diverse computational models, finite automata, CSP, and UNITY, can be cohesively analyzed with respect to their syntactic representations by considering the different instantiations of these models as elements of complete metric spaces. This chapter shows that these same three computational models can also be cohesively analyzed with respect to their semantic representations using the branching temporal logic. These two chapters complete the unified mathematical framework that comprised the first major objective of this research. The next chapter addresses the second major objective, a methodical technique for developing formal specifications.

VI. Extensional Based UNITY Program Transformations

This chapter presents the second major portion of this research effort. It includes the development of methodical techniques to generate UNITY formal specifications from either informal specifications or from multiple UNITY programs, and the methodical transformation of one UNITY program into another. When merging multiple programs into one, these techniques preserve desired program properties from the multiple programs, and when transforming one UNITY program into another, the presented methodology preserves the desired program properties of the program to be transformed.

Section 6.1 discusses the reasons for the selection of UNITY as the formal specification language for this chapter. This section also discusses the modeling power of UNITY, and demonstrates UNITY's ability to model asynchronous distributed architectures (such as the Intel Hypercube).

Section 6.2 develops a semantic model for UNITY program execution based on the concept of a *state space*. This semantic model, termed the *state space semantics*, differs from the approach used in the Chandy and Misra book (64). This state space semantics permits an intuitive insight into the execution of a UNITY program, and also leads to new results regarding the concept of program *union* and *superposition*. The basic concept of the state space semantics is that a UNITY program is a dynamical system, whose execution generates trajectories through a state space. This state space consists of vectors whose component values represent the instantiations of the (named) variables of the UNITY program.

Section 6.3 uses the foundations from the state space semantics to develop a methodical approach to the generation of UNITY programs as formal specifications, and to their transformation into other UNITY programs. For both the generation and the transformation, the resulting UNITY programs retain the desired program properties of either their constituent pieces, or immediate predecessor. This section presents a UNITY program, called search, which combines both the generation and the transformation processes into one program. This program randomly searches for the generated/transformed programs that retain the desired properties of either the separate programs that were combined to

form a new one, or else the original program before the transformation. Although the basic concepts behind the Search program are not new (71, 210), their application to UNITY is new.

6.1 UNITY

UNITY was chosen for this chapter for two primary reasons. The first is that UNITY has syntax which more closely resembles standard imperative programming languages, versus a model such as CSP, which does not (note that in his CSP book (165), Hoare uses LISP to illustrate how the trace theory aspects of CSP could be implemented). The second reason is that UNITY permits relatively simple representations of other models, along with formal specifications. This characteristic of 'simple representation' is one of Peter Naur's primary requirements for ranking the worth of any computing language (considering UNITY as a modeling and specification language), a concept Naur expressed in the following quote (which Naur attributes to Otto Jespersen) (255):

> That language ranks highest which goes farthest in the art of accomplishing much with little means, or, in other words, which is able to express the greatest amount of meaning with the simplest mechanism.

As a specification language, UNITY is equivalent in expressive power to the first order temporal logic over a finite domain, since all variables must be individual (to prevent dynamic statement modification), and must come from finite domains to ensure statement termination (see Section 4.4). The actual syntax of UNITY programs can be equivalently expressed using the first order predicate logic, but the inclusion of the UNITY execution model increases this expressive power to the temporal predicates. Note that Chandy and Misra present other reasons for choosing UNITY in the first chapter of their book (64), while Knapp states that (193)

> it abandons the notion of a process as a fundemental concept of parallel program design and that it facilitates program derivation by rigorously separating the concerns of program correctness from that of hardware and implementation. The method is completely formal in the sense that ... all inferences are done

within predicate calculus. Furthermore a program is viewed as a mathematical object enjoying certain properties (invariants, stability).

As examples of the modeling power of UNITY, consider Chandy and Misra's (64) examples of UNITY programs that simulate certain asynchronous and synchronous message passing systems, in particular systems based on Hoare's CSP (165) and the work of Milner (244). Additionally, Chandy and Misra give examples of transforming certain logic programs (based on the work of Kowalski (195)) into their UNITY equivalents (equivalent based on the ability to resolve the same facts). The modeling power of UNITY is further demonstrated by the following example based on one of the most unstructured models of a message passing system, what Bertsekas calls the *totally asynchronous* model (39).

Consider a function f

$$f : X \times Y \rightarrow X \times Y$$

whose domain and codomain are a set of 2-tuples (ordered pairs), such that

$$f(x, y) = (g(x, y), h(x, y))$$

where

$$g : X \times Y \rightarrow X$$

and

$$h : X \times Y \rightarrow Y$$

If this function is to be used iteratively to generate an unbounded sequence

$$((x_0, y_0), (x_1, y_1), ...)$$

where

$$(x_{i+1}, y_{i+1}) = f(x_i, y_i) \qquad i \in \{0, 1, 2, ...\}$$

then the totally asynchronous model would have two processors, each with its own local memory, such that one processor iterates a sequence of 2-tuples using only the function g, and the other processor iterates another sequence of 2-tuples using only the function h.

Thus the first processor repeatedly evaluates

$$(x, y) := (g(x, y), y)$$

while the second processor repeatedly evaluates

$$(x, y) := (x, h(x, y))$$

and at arbitrary times each processor receives a message updating its local copy of (x, y) (note that it doesn't matter if the update is only for one of the variables x or y, or both of them). At unspecified times, and with arbitrary transit delays, a processor will send its local copy of the 2-tuple to the other processor in an update message. The totally asynchronous requirement is that each processor receives such update messages an unbounded number of times, and for any given processor and any given (real) time t, there exists another time, say $\hat{t}$, such that

$$\hat{t} > t$$

and the processor receives an update message at time $\hat{t}$.

This totally asynchronous system is modeled by the following UNITY program (design), in that both generate exactly the same sequences of states.

```
Program Totally Asynchronous
    assign
        (r,s) := (g(r,s),s) |
        (u,v) := (u,h(u,v)) |
        (r,s) := (u,v) |
        (u,v) := (r,s)
    end
```

The first two statements replicate the sequences of states generated by the two processors, while the third statement is equivalent to an update message from the second processor (the one iterating (u, v)) to the first (the one iterating (r, s)), while the fourth statement is equivalent to an update message from the first processor to the second. This UNITY

program assumes that the update messages send a copy of the complete 2-tuple. If instead, only that portion of the 2-tuple that is actually changed by a processor is sent out in the message (i. e. the first processor only changes the first component, while the second processor only changes the second), then the third and fourth statements in the UNITY program would be

$$(r,s) := (r,v) \,|$$
$$(u,v) := (r,v)$$

The execution model for the UNITY weak fair choice operator | ensures that the third and fourth statements (the update messages) are executed an unbounded number of times. The execution model also ensures that with respect to any given instant of time considered as the present, that eventually the third statement will execute at some future instant of time, and the fourth statement will eventually execute at some future instant of time (see Section 4.4). Thus the set of all possible sequences of vectors whose components are the values of the variables of the UNITY program, is equal to the set of all possible sequences of vectors whose components are the values of the corresponding variables from the totally asynchronous system.

6.2 State Space Semantics

This section introduces the state space semantics that, along with the material from Section 4.4, form the basis for the the material in Section 6.3. The purpose of this state space semantics is to consider a UNITY program as a dynamical system, where the system states are the vectors formed by the instantiations of the named variables from the program, so that a sequence of states corresponding to an execution can be analyzed as would a trajectory within a state space. But whereas the trajectories for dynamical systems based on difference or differential equations are fixed (352), the inclusion of the weak fair choice operator (see Section 4.4) into a program yields multiple possible trajectories for a given program. So, while a difference equation yields one trajectory for a given initial state, a UNITY program may generate a collection of trajectories, implying that a UNITY program may correspond to a collection of difference equations. This state space semantics is basically an abstraction of what Arbib, Kfoury, and Moll call the *operational semantics* of While programs (190).

Is this analogy of a UNITY program as a dynamic system consistent with the standard meanings of 'dynamic system' and 'state'? Consider the following definition of a dynamical system from Cook (81):

> A dynamical system is characterized by a set of related variables, which can change with time in a manner which is, at least in principle, predictable provided that the external influences acting on the system are known.

This would certainly seem to apply to UNITY programs, given that the 'external influences' are the different possible execution sequences resulting from the fair choice operator on statements | (note that Chandy and Misra designate this operator as [] (64)). Cook also states that a 'state'

> should contain all information about the past history of the system which is relevant to its future behavior. That is to say, if the state at a given instant is known, then its subsequent evolution can be predicted without any other knowledge of what has previously happened to the system.

This UNITY state based on the values of the named variables also satisfies this definition, given that the execution sequence is fixed. Thus for a UNITY program with the fair choice operator, each possible execution sequence (there is in general a countable number of such sequences) must be considered separately so that the UNITY program can be treated as a dynamical system with its named variables defining the states of the system. Note that just the named variables is not quite enough information to satisfy the state requirement, since there is no way of knowing which assignment to execute next. Thus the UNITY states also contain an implicit variable which contains some type of pointer to the next executable assignment. This idea is also treated in Section 3.3 of Chandy and Misra (64).

This section presents the viewpoint that a UNITY program can be considered a dynamical system whose dynamics are represented by trajectories through a state space of vectors formed from the named variable instantiations. This implies that analysis of dynamical systems based on difference equations of the form

$$x(n+1) = T(n, x(n)) \tag{6.1}$$

can be applied to the analysis of UNITY programs. In Equation 6.1 the vector x denotes the state, that is the vector whose components are the values of the named variables, the index n is a counter that corresponds to the assignment statement being executed at the n^{th} step, and the operator T maps states into states, and is itself a function of which assignment statement is executing (thus a function of n). This operator T is also dependent upon which execution sequence (of the possible ones resulting from the fair choice operator $[]$) is chosen, but once a sequence is fixed, then T only depends upon n. An execution sequence is a sequence of events (see Definition II.1).

Lemma VI.1 *Given the set S of possible execution sequences for a given UNITY program, then each execution sequence*

$$s_\alpha \in S$$

generates a sequence of states equal to the sequence of states given by

$$x(n+1) = T_\alpha(n, x(n)) \qquad n \in \mathrm{N} \tag{6.2}$$

where x denotes the state, and T_α is a Turing computable function of s_α.

Proof: A given execution sequence fixes the sequence of states once the initial state x_0 is instantiated, so that T_α is determined by this fixed sequence of execution of the UNITY assignments. Equation 6.2 represents this sequence of states in the form of a standard initial value problem (303). ■

In Equation 6.2 the index is α, suggesting that the number of possible execution sequences for a given UNITY program might be uncountable. Certainly the number of executable sequences that would be possible without the weak fairness requirement on the UNITY statement operator | is uncountable, but is this uncountable set 'reduced' to a countable set with the inclusion of weak fairness. Unfortunately, the answer is no.

Theorem VI.2 *The number of different execution sequences for a UNITY program with at least two assignment statements is uncountable.*

Proof: Denote the execution sequences by unbounded strings over the alphabet $\{0, 1\}$, where a 0 denotes the choice of one statement, and a 1 denotes the choice of the other. The set of all such strings is uncountable, and can be put into a one-to-one relationship with the real numbers between 0 and 1 (with an implicit leading deicmal point) Now remove those strings that violate the weak fair choice operator | requirement. This means removing all strings that end in an unbounded number of 0's or 1's, which is equivalent to removing a subset of the rational numbers in the interval. This removed subset of rationals is countable, which means that there are still an uncountable number of strings left. ■

Another problem with the weak fairness requirement is that it is not easily testable in finite time. An interesting conjecture is that perhaps a different type of statement operator requirement would yield both a countable set of execution sequences would be easier to test in finite time.

Although Equation 6.2 is in the form of the classical nonlinear dynamical system, the standard analysis of dynamical systems described by Equation 6.1 cannot be directly

applied to UNITY programs. This results from the requirements imposed on the operator T from Equation 6.1 by the many different classifications of the initial value problem. These classifications can be grouped into the following three types. The first requires the operator T have as domain and codomain either a compact set or a bounded subset of a complete set (a precompact set), such that T is analytic (everywhere differentiable) over this set (307). The second classification forces the vectors x to be elements of a Banach space (complete normed linear space (256)), plus T must be continuous over this domain (204, 303). The third classification requires that T must be continuous over its domain, which, along with its codomain, must be a Frechet space (42). A Frechet space is defined so as to be metrisable (a metric can be defined for the space), such that a complete metric space results (294). Thus at a minimum, the domain and codomain of T must be complete (with respect to some topology or metric), and T must be continuous.

Although the completeness requirement can be satisfied, say by using some subset of the natural numbers as a domain and codomain for T, the continuity of T remains a problem. Even if an appropriate definition of continuity could be found so that the continuity of any possible UNITY assignment could be assessed, there remains yet another hurdle. The primary purpose of using the classical analysis is to address the stability of the UNITY program at the fixed points. Since UNITY programs in general would be nonlinear systems, then the stability analysis would require taking the derivatives of all possible UNITY assignments (81, 187). What would be the derivative of, say a conditional assignment such as

$$z := x + y \text{ if } x > y$$

Consequently, this section presents the basic theory of difference equations of the type of Equation 6.2 that correspond to UNITY programs. These equations can be characterized as coming from a family of similar equations, where each member of the family has a different set of $T_\alpha(n, o)$ depending upon the actual execution sequence chosen. The primary concern here is one of *stability* for these equations, since this stability relates directly to the concept of fixed points for the corresponding UNITY programs.

The term *fixed point* as defined by Chandy and Misra "is a program state such that execution of any statement in that state leaves the state unchanged." and "reaching a

fixed point is equivalent to termination in standard sequential-programming terminology." (64)

Definition VI.3 *Given a UNITY program P, a* fixed point *is any state s, such that for any execution sequence*

$$\{e_0, e_1, ...,\}$$

of P,

$$\exists k[n \geq k \Longrightarrow e_n(s) = s] \qquad k, n \in \mathrm{N}$$

One assumption in this definition is that $e_n(s)$ is defined for any natural number n, an assumption substantiated by the UNITY philosophy that every assignment statement terminates. This definition permits a fixed point to be 'unreachable', in that although s may be a state that satisfies the fixed point definition, some (or all) execution sequences of a given program may not actually generate the state s.

Consider the UNITY program E1 from Section 4.4, which is repeated here:

Program E1

initially

x,y = M,N M,N are integers

assign

y := x |

x := x-1 if x=y || x := x+1 if x≠y

end

Since E1 has only two named variables, x and y, a sequence of states for E1 is a sequence of 2-tuples, such that the first component is the current value of x, while the second is the current value of y. So if x is initially equal to 3, and y is initially 2, then one sequence of states for E1 is

$$(3,2),(4,2),(5,2),(5,5),(4,5),(5.5),(4,5),(5,5)$$

This corresponds to two executions of the second statement, followed by the first statement, then two more executions of the second statement, and so on. As shown by continuing this

execution sequence indefinitely, program E1 does not have a fixed point over the domain (for x and y) of computable numbers (see Appendix B).

For any UNITY program whose states represent an n-tuple of the instantiations of the named variables, these n-tuples can be considered as vectors from a subset of C^n, where C denotes the computable numbers. Although finite computers actually execute programs with finite domains, these unbounded domains suffice for the mathematical analysis. This means that certain UNITY programs, such as E3 below, have theoretical properties that cannot be realized on any finite machine.

Program E3

initially

x = c {c ≠ 0}

assign

x := x/2

end

Program E3 has the property that if x is initially any nonzero computable number, then this program has no reachable fixed point (zero is the only fixed point within the computable numbers). But if x is initially any nonzero number from the domain of finite precision floating point numbers, then E3 does have a fixed point, which is x=0. The formal reason why E3 has a fixed point on any finite machine is that the domain for any individual variable is actually a finite set of integers,

$$\{0, ..., 2^{N-1}\}$$

where N is the maximum number of bits used for variable storage, and the division represents integer division.

Denote by D a set that represents a suitable domain for all of the individual variables of any UNITY program (what D represents can vary, depending upon the analysis). Then the state space for the program is D^n, where n is the number of named variables in the program. Any execution sequence generates a sequence of discrete state changes, each discrete change resulting from a single assignment statement (all of whose components ex-

ecute simultaneously). This means that any sequence of states generated by the execution of the program is represented by a trajectory of points within D^n. The convention is that for any sequence of states

$$\{s_0, s_1, ..., s_n, ...\}$$

then for any two states

$$s_i, s_{i+1} \qquad i \in \{0, ..., n-1\}$$

the trajectory is represented by a directed arc from the state s_i to the state s_{i+1}. Thus the trajectory for any (nonsingleton) sequence of states is a directed path (in the sense of a directed graph).

Definition VI.4 *Given the UNITY program F, then the* domain *for F is the set D^n, where each element of this domain is a state vector,*

$$(v_0, v_1, ..., v_n)$$

such that each v_i denotes the instantiation of a named program variable from F.

Definition VI.5 *Given the UNITY program F, with domain D^n, then a* trajectory *of F is a sequence of state vectors contained within the domain.*

The trajectories of program E3 above illustrate an important concept regarding the proof of program termination using well founded sets. A well founded set is a set along with a strict partial order, such that there are only finite decreasing sequences (with respect to the order). This means that if a trajectory represents a decreasing sequence within a domain that is a well founded set, then that trajectory reaches a fixed point, i.e. it terminates in a single state vector. Thus the fixed point concept is analogous to that of a stable equilibrium point in a dynamical system (187). Consider a trajectory for program E3 within the domain C (the computable numbers) that does not start at zero. Such a trajectory approaches arbitrarilty close (Cauchy sequence with respect to the $|o, o|$ metric over the domain) to the zero vector, but never reaches it in finite time. This implies that the domain is not a well founded set, and program termination is not ensured. However, if

the domain is changed to the set of floating point numbers with fixed 'word length', then any trajectory will reach a fixed point, the domain is a well founded set, and the program 'terminates'.

Definition VI.3 implies that each assignment statement in an execution sequence cannot alter the values of the variables once the fixed point has been reached, any trajectory for a program with a reachable (for that trajectory) fixed point terminates in a single point. Thus any trajectories terminating in a single point correspond to the standard notion of program termination (217). If every trajectory for a UNITY program ends in a single point in the state space, then that program can be said to terminate. This leads to the idea that certain UNITY programs can be considered as *subprograms* that can be called from other UNITY programs with the guarantee that these subprograms will terminate within a certain fixed number of statement executions.

Definition VI.6 *Given the UNITY program P, then P is a* subprogram *if and only if, there exists a Turing computable function f, such that for any execution sequence of P,*

$$\{e_0, e_1, \ldots,\}$$

and any (possible) initial state s_0*, then any sequence of states*

$$\begin{aligned} s_1 &= e_0(s_0) \\ s_2 &= e_1(s_1) \\ &\vdots \\ s_n &= e_{n-1}(s_{n-1}) \\ &\vdots \end{aligned}$$

satisfies

$$k \geq M \Longrightarrow e_k(s_M) = s_M \qquad \in \mathrm{N}$$

where

$$f(P) = M \qquad M \in \mathrm{N}$$

Any subprogram will reach a fixed point within at most M statement executions, where M is computed by a Turing computable function dependent only on the subprogram, and not on either the initial state nor the actual execution sequence chosen. Thus the subprogram satisfies the intent of the 'functions' used by Chandy and Misra in constructing UNITY programs. Any algorithm that can be written as a UNITY program with a single asssign ment statement and that terminates for any possible input can in general be converted into a subprogram (i.e. any total recursive function that can be encoded as a single statement UNITY program). If a UNITY program contains two assignment statements, and they both must execute at least once before the fixed point is reached, then that UNITY program is not a subprogram, since the execution model permits either statement to execute M successive times, for any finite M. For algorithms that compute convergent sequences, if the rate of convergence can be computed (and is fixed), then in general the algorithm can be converted into a subprogram, given that the fixed point is defined by one or more terms from the convergent sequence that satisfy a termination condition (such as two successive terms within some interval).

This definition of a subprogram implies that any subprogram can be modified into another UNITY program that reaches the same fixed point for the named variables of the subprogram, and also sets a boolean variable to *true* only after the fixed point on these variables has been reached.

Theorem VI.7 *For any subprogram P, there exists another UNITY program G, such that any property of P is a property of G, and all of the named variables of P are also named variables of G. Additionally, the named variables of G that are named in P, reach a fixed point for any execution of G, that is equal to the fixed point reached for any execution of P, given that these variables have the same initial values within P and G, plus G contains a named variable that is assigned a true value only after these common variables have reached the fixed point.*

Proof: The proof is by construction of the program G from P. Add to P two additional variables that are not named in P, these are k, which is declared integer and intialized to 0, and flag, which is declared boolean and initialized to *false*. Each assignment

statement s of P is changed into

$$s \| k := k + 1$$

which does not affect the assignments made to the variables of P, nor the attainment of a fixed point for the variables of P in M statement executions. One additional assignment statement is added,

$$\text{flag} := \mathit{true} \text{ if } k \geq \mathrm{M}$$

which also does not affect the assignments made to the variables of P, nor their attainment of a fixed point in M statements excluding this additional one. Thus whenever flag = *true*, the variables from P have reached the fixed point (although G itself is not claimed to be a subprogram, since this additional statement can execute an arbitrary finite successive number of times). ■

This state space concept leads to a development of the program properties *unless*, *ensures*, and *leads-to* which is parallel to the UNITY based development in Chandy and Misra. This parallel development utilizes the concept that if a predicate p is *true* at some point in the execution, this can be stated as saying that the corresponding state vector is an element of the set $\hat{p}$ of states for which p holds. Conversely, if p is *false*, then the corresponding state vector is not contained within the set $\hat{p}$.

Definition VI.8 *Given the UNITY program P over the state space domain D^n, and any program property (predicate) p which is a function of the state vector, then the statement that the program property p is true at that instant of program execution that corresponds to state s, is denoted by*

$$s \in \hat{p}$$

where

$$s \in \hat{p} \Longleftrightarrow p(s)$$

and

$$s \in D^n \iff true$$
$$s \in \emptyset \iff false$$

While a program is executing, whenever the state trajectory lies within the set $\hat{p}$, then p is *true*, and if the state trajectory lies outside the set $\hat{p}$, then p is *false*. Since the entire state space is denoted by D^n, then $D^n - \hat{p}$ denotes the set that is outside the set $\hat{p}$. With these preliminaries, the program property definitions can be restated in terms of this state space semantics.

Definition VI.9 *Given the UNITY program P over the state space domain D^n, any execution sequence*

$$\{e_0, e_1, ...,\}$$

any (possible) initial state s_0, any sequence of states

$$\begin{aligned} s_1 &= e_0(s_0) \\ s_2 &= e_1(s_1) \\ &\vdots \\ s_n &= e_{n-1}(s_{n-1}) \\ &\vdots \end{aligned}$$

and the nomenclature

$$\neg\hat{p} = D^n - \hat{p}$$

then:

p unless q *iff*

$$s_i \in (\hat{p} \cap \neg\hat{q}) \Longrightarrow s_{i+1} \in (\hat{p} \cup \hat{q})$$

p stable *iff*

$$p \ unless \ false$$

p invariant *iff*

$$s_0 \in \hat{p} \wedge p \; stable$$

p ensures q *iff*

$$p \; unless \; q \wedge \exists k[s_k \in (\hat{p} \cap \neg\hat{q}) \Longrightarrow s_{k+1} \in \hat{q}]$$

p leads-to q *iff*

$$s_i \in \hat{p} \Longrightarrow \exists k[k \geq i \wedge s_k \in \hat{q}]$$

From the definitions of *unless*, *stable*, and Definition VI.8, it follows that

$$p \; stable \Longleftrightarrow s_i \in (\hat{p} \cap D^n) \Longrightarrow s_{i+1} \in (\hat{p} \cup \emptyset)$$

which reduces to

$$p \; stable \Longleftrightarrow s_i \in \hat{p} \Longrightarrow s_{i+1} \in \hat{p}$$

which states if p *stable* holds for a program, then once any trajectory enters the set $\hat{p}$ it does not leave the set. The definition of *invariant* implies that if p *invariant* is *true* for some program, then no trajectory of the program ever leaves the set $\hat{p}$. The *leads-to* concept is one easily verified from the state space trajectories. To show that p *leads − to* q, it suffices to have every trajectory that passes through $\hat{p}$ pass through $\hat{q}$ either at the same or a later state.

A consequence of Lemma VI.1 is that once an execution sequence is fixed, then the sequence of assignments for a UNITY program can be used to define a *solution operator*

$$U(n, m)$$

such that

$$U(n, m)x(m) = x(n) \qquad 0 \leq m \leq n \qquad n, m \in \mathrm{N}$$

where

$$x(0) = s_0$$

denotes the inital value (established by the initially section of the UNITY program) for the state (represented as the function x). With respect to the nomenclature of a sequence of states as

$$\{s_0, s_1, ..., s_n, ...,\}$$

then

$$x(n) = s_n$$

denotes the value of the state for a given execution sequence of the program. This formulation of the solution operator is basically identical to that for an initial value differential equation problem (273), except that the domain for the evaluation of the state values is the natural numbers. This leads to the following definition, which is also based on the analysis of the differential equation problem.

Definition VI.10 *Given the* solution operator *U, whose domain and codomain are the class of all possible states of a given UNITY program P, that satisfies*

$$0 \leq m \leq n \Longrightarrow U(n,m)x(m) = x(n) \qquad n, m \in \mathrm{N}$$

and defines the solution to the difference equation of Equation 6.2 for a given execution sequence of P, then U is a discrete evolution system *if and only if,*

$$U(m,m) = I$$

and

$$U(i,j)U(j,k) = U(i,k) \qquad 0 \leq k \leq j \leq i$$

where I denotes the identity operator on states.

This discrete evolution system is identical with that for a differential intial value problem except that in the discrete case there is no strong contir equirement for the mapping f, whose domain is the class of ordered pairs of natu mbers such that the first number

is not less than the second, and (273)

$$f(n,m) = U(n,m)$$

A direct consequence of Definition VI.10, and the UNITY requirement that each assignment yields a unique state given a unique state, is that every UNITY program has an associated set of discrete evolution systems.

Theorem VI.11 *Given the UNITY program P, there exists a unique set*

$$\bigcup_{\alpha} U_{\alpha}$$

whose elements are discrete evolution systems, such that for any given execution sequence of P there is a unique U_{α}.

Proof: This follows directly from Lemma VI.1 and Definition VI.10. Since each execution sequence is unique (see Section 3.3 of (64) for the standard UNITY execution model), then given an execution sequence the set T (from Lemma VI.1)

$$T = \{T_0, ..., T_m, ...\}$$

is unique, and since each U_i is uniquely determined by the set T that corresponds to each execution sequence, each U_i is unique. ■

As an example, consider the UNITY program E1, such that the execution sequence (under the UNITY model) consists of an alternating sequence of the two statements, with the statement y := x executing first. This means that the discrete evolution system can be defined as

$$U(m+1,m) = \begin{cases} \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} & \text{if } m = 0 \bmod 2 \\ f & \text{if } m = 1 \bmod 2 \end{cases}$$

where the states are represented as column vectors whose first component is the value of x, and whose second component is the value of y. The function f maps states to states,

such that

$$f\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x-1 \\ y \end{bmatrix}$$

Although this is a simple example, it does demonstrate that the formulation of the discrete evolution system can take many forms, with two possibilities shown here. Whenever the next assignment to execute is y := x, then the solution operator $U(m+1, m)$ takes the form given by the matrix above, whereas if the other assignment executes, this solution operator takes the form given by f. Thus the evaluation

$$U(m+1, m)x(m) = x(m+1)$$

where x denotes the vector representation of the state, can be performed for any natural number m. The value of $x(i)$ for any $i \in \mathrm{N}$ can be computed from the initial value $x(0)$ by induction.

A result of having discrete evolution systems for UNITY programs is that a trajectory cannot intersect itself unless the intersection is the entrance to a cycle, i.e. a loop.

Theorem VI.12 *Given a UNITY program P, then for any trajectory of P,*

$$\{s_0, s_1, ...\}$$

there can be no noncyclic intersections, that is

$$(s_{j-1} \neq s_{k-1} \wedge s_j = s_k) \Longrightarrow \forall n[s_{j+n} = s_{k+n}] \qquad n \in \mathrm{N}$$

Proof: (Based on a proof from Pazy (273)) At the intersection state, the unique

$$U(m, j)$$

and the unique

$$U(m, k)$$

imply that for $j = k$, the two must be equal. ■

This implies that any trajectory can only contain one cycle, since there can be no exit from a cycle. Another consequent is that if a cycle is contained entirely within a set $\hat{p}$, then eventually p is *stable*, which can be expressed as

$$\Diamond\, p \textit{ stable}$$

A very useful result of Theorem VI.12 is that the counterexample to

$$\frac{p \text{ leads-to } q \textit{ in } F,\ p \text{ leads-to } q \textit{ in } G}{p \text{ leads-to } q \textit{ in} F|G}$$

with respect to the *union* (see Section 7.2 of Chandy and Misra (64)) of two UNITY programs F and G, can be demonstrated using state space trajectories in an abstract manner, instead of having to actually write the UNITY programs F and G. The claim that p *leads-to* q for the program F is equivalent to stating that all trajectories of F that have a state $s_i \in \hat{p}$ also have a state $s_j \in \hat{q}$ where $j \geq i$. Theorem VI.12 implies that if $j > i$, then there can be no cycles in the trajectory between s_i and s_j. The same reasoning holds for G, any trajectories of G that have a state $s_k \in \hat{p}$ also have a state $s_l \in \hat{q}$ where $l \geq k$. Theorem VI.12 implies that if $l > k$, then there can be no cycles in the trajectory between s_k and s_l. But when the two programs are unioned together, i.e. $F|G$, then it is possible that a trajectory from F and another one from G could together form a cycle between the state s_i that lies in $\hat{p}$ and the state s_j that lies in $\hat{q}$. Figures 6.1, 6.2, and 6.3 demonstrate graphically how this could occur. The solid trajectory in Figure 6.1 is one from F that satisfies the p *leads-to* q property, while the dashed trajectory in Figure 6.2 is one from G that satisfies this property. But when the programs are unioned together, then the cycle that forms as shown in Figure 6.3, is a possible trajectory of $F|G$, and so the unioned program does not satisfy the p *leads-to* q property.

This state space semantics leads to an alternate development of what Chandy and Misra call the *superposition* structure for use in "layered program development" (64). The idea of superposition is that an *underlying program*, with its associated named variables called the *underlying variables*, can be modified by either adding another statement (the idea behind union), or by adding another assignment component to an existing statement.

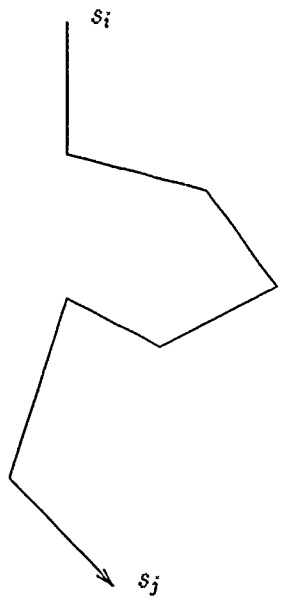

Figure 6.1. Example Trajectory From F

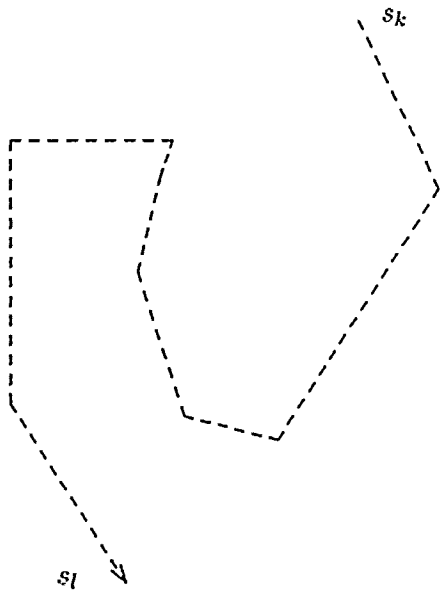

Figure 6.2. Example Trajectory From G

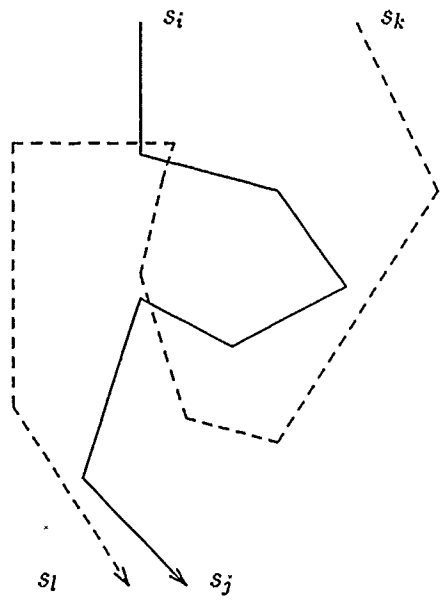

Figure 6.3. Example Trajectory From $F|G$

The allowable types of these modifications are summarized in the following two rules from Chandy and Misra: (64)

Rules of Superposition (Original)

1. Augmentation rule. A statement s in the underlying program may be transformed into a statement $s\|r$, where r does not assign to the underlying variables.
2. Restricted union rule. A statement r may be added to the underlying program provided that r does not assign to the underlying variables.

Chandy and Misra then prove the superposition theorem, which states that since no modification to an underlying program that satisfies the rules of superposition can modify any of the preexisting underlying variables, then every program property of the underlying program is preserved in the modified program. This concept can be formalized with respect to the state space semantics through the use of *restrictions.*

Definition VI.13 *Given the UNITY program F with domain D^n, and the program G with domain D^m, $m \geq n$, then a trajectory t of G* restricted to the domain of *F is equal to*

the trajectory u of F (or equivalently, the trajectory u of F is equal to the trajectory t of G restricted to the domain of F), if and only if, for any state vector of t,

$$(s_0, s_1, ..., s_n, s_{n+1}, ..., s_m)$$

there exists a state vector of u,

$$(s_0, s_1, ..., s_n)$$

and for any state vector of u,

$$(r_0, r_1, ..., r_n)$$

there exists a state vector of t

$$(r_0, r_1, ..., r_n, r_{n+1}, ..., r_m)$$

This leads to the modified version of the rules of superposition which expresses a more inclusive form of the original rules in the language of the state space semantics.

Definition VI.14 (Rule of Superposition (Modified)) *Given the UNITY program F, then G is a* superposition transformation *of F, if and only if, any trajectory of G restricted to the domain of F is equal to a trajectory of F, and any trajectory of F is equal to a trajectory of G restricted to F.*

This generates a more inclusive form of the superposition theorem from Chandy and Misra.

Lemma VI.15 (Intermediate Form of Superposition Theorem) *If the UNITY program G is a superposition transformation of the program F, then every program property of F is a property of G.*

Proof: Since the program properties of F are dependent only upon the variables represented by the state vectors in the domain of F, then every trajectory of G satisfies the program properties of F; every trajectory of G is equivalent to a trajectory of F with respect to the domain of F. ■

This proof implies that the theorem still holds even if there are trajectories of F that do not correspond to trajectories of G restricted to the domain of F. This leads to the following more general result.

Theorem VI.16 (General Form of Superposition Theorem) *If every trajectory of the UNITY program G restricted to the domain of the program F is equal to a trajectory of F, then every program property of F is a property of G.*

Proof: Since the program properties of F hold for every possible trajectory, any trajectory of F which does not have a correponding trajectory in G does not affect the conclusion of the theorem. ■

By extending the state space semantics to include the concepts of a *projection* and a *sum*, a result which combines the concept of both superposition and union is developed.

Definition VI.17 *A function whose domain and codomain are a given UNITY domain D^m is a* projection *on D^m, if and only if,*

$$f(f(D^m)) = f(D^m).$$

This projection is similar to the standard projection from vector space analysis, except here there is no claim that either the projection is a linear transformation, or that the domain is a linear space (256).

Definition VI.18 *The domain D^m is the* sum *of the two domains D^k and D^{m-k}, if and only if, there exists two projections f and g on D^m, such that for any state vector in D^m,*

$$f(s_0, s_1, ..., s_m) = (s_0, s_1, ..., s_k, 0, ..., 0)$$
$$g(s_0, s_1, ..., s_m) = (0, ..., 0, s_{k+1}, ..., s_m)$$

and

$$(s_0, s_1, ..., s_k) \in D^k$$
$$(s_{k+1}, ..., s_m) \in D^{m-k}$$

If a domain A is the sum of two other domains B and C, then each of B and C has state vectors that represent a subset of the set of variables represented by the state vectors of A. Additionally, the union of the two sets of variables corresponding to B and C equals the set of variables corresponding to A, such that the intersections of these two sets is empty. This sum is also similar to the concept of a linear space being the sum of two subspaces (256).

The concept of the domain of one program being the sum of the domains of other programs leads to the following theorem, which combines concepts from both the union theorem and the superposition theorem of Chandy and Misra.

Theorem VI.19 (Superposition/Union Theorem) *If the UNITY program H is formed by the union of the program F with the program G (adding the declarations, initializations, and assignment statements of the program G to the program F), and if the domain of the UNITY program H is the sum of the domain of the program F and the domain of the program G, then any program property of F is a property of H, and any property of G is a property of H.*

Proof: Any trajectory of H restricted to the domain of F (G) is a trajectory of the program F (G), so that Theorem VI.16 implies that all properties of F (G) are also properties of H. ■

If the domains of F and G differ, and G is a superposition transformation of F, then in a sense G is an extension of F, since any trajectory of F corresponds to a trajectory of G that performs the same assignments on the variables of F as does F, but can also perform other assignments to variables of G that are not variables of F. If F and G have identical domains though, and G is a superposition transformation of F, then both programs generate the exact same trajectories. This implies that for a given domain, the superposition transformation defines equivalence classes of UNITY programs, where the equivalence is based on identical trajectories. Thus for a given equivalence class, any one program is a superposition transformation of any other program, even though no two may be identical syntactically, and there is one fixed behavior (as defined by the trajectories) for all programs in the entire class.

Definition VI.20 *Within the class of UNITY programs with the same domain, two UNITY programs are* equivalent, *if and only if, one is a superposition transformation of the other.*

This definition follows from the observation that within a set of UNITY programs all having the same domain, superposition transformation represents a reflexive, symmetric, and transitive relationship between UNITY programs, that is, it induces an equivalence relation.

Since superposition transformations yields programs with identical trajectories within a single domain, what type of transformation leads to the 'extension' concept within domains? That is, what type of transformation applied to a program F with domain A, yields another program with domain A whose behavior (in terms of trajectories) is an extension of the behavior of F. Such a transformation is the *simulation transformation.*

Definition VI.21 *Given the UNITY program F with domain A, then the program G with domain A is a* simulation transformation *of F, if and only if, G simulates F (see Definition II.8).*

If G simulates F, and both programs have the same domains, then any trajectory of F is a trajectory of G. But unlike the superposition transformation, G may have trajectories that F does not. A consequence of G being an extension of F is that program properties of G are also properties of F, but the converse is not true.

Theorem VI.22 *If G is a simultation transformation of F, then any program property of G is also a program property of F.*

Proof: Since G simulates F, then any sequence of states (trajectory) of F is a sequence of states (trajectory) for G. Since any program property of G must hold for all trajectories, then these properties hold for ali trajectories of F. ■

Obviously for any UNITY program there exists the trivial simulation transformation that leaves the program unchanged, i.e. an identity transformation. But what of the existence of other nontrivial simulation transformations? Consider this modification of the

program E3:

Program {Simulation Transformation of E3}

initially

$x = c \ \{c \neq 0\}$

assign

$x := x/2 \mid x := 0$ if $x = c$

end

If the first statement executed is the

$$x := x/2$$

statement, then this transformed program has exactly the same trajectories as does E3. But if the other statement is executed first, then this program has a trajectory which E3 does not have, assuming that c is not initially zero. Thus the set of all trajectories of this transformed program is a superset of the set of all trajectories of E3, and this program is a simulation transformation of E3. Note that this program has the property that for any given execution, the probability of reaching a fixed point is 1/2, whereas the original program E3 never attains a fixed point (over the computable numbers).

6.3 *Development of UNITY Specifications*

This section presents a heuristic type search technique for developing UNITY programs as formal specifications and transforming these UNITY programs into other UNITY programs such that specified program properties are preserved. This section also presents an algorithm for transforming UNITY programs to achieve greater efficiency with respect to mappings of the programs to multiprocessor architectures. The techniques of this section are based on those developed for systolic arrays by Huang and Lengauer (173, 172). A systolic array is a collection of processors with local memory, and some type of interconnecting communications network. This communications network is usually designed so that the processors are considered to be either a two or three dimensional array. This means each processor can communicate only with its nearest neighbors, where the nearest neighbors are found by visualizing the processors at the nodes of either a two or three dimensional grid. Since each processor in the array only communicates with a fixed number of other processors (not counting the boundaries of the array), a systolic array is considered a *constant valence* cube. The term 'constant valence' means that the number of communications links connected to a single processor is constant, versus a *hypercuboid* (such as the Intel Hypercube) architecture, where the number of communication links is $lg(n)$ where n is the number of processors (321). Typically systolic arrays are designed so that the processors perform very few (often only one) different operations individually, and the communications links pass specific kinds of data in a repetitive manner. Such an array is actually a hardware implementation of a specific algorithm, a *systolic program* (207). This hardware implementation of the systolic program is called a *systolic solution*, and such systolic solutions exist for many matrix and graph based algorithms (174, 170, 52).

The basic idea of this approach to developing UNITY programs is to start with a description of the algorithm to be specified, and decompose it into individual *operations*. An operation, which is not precisely defined, combines the properties of both an atomic action and a function application. With respect to informal specifications, an operation corresponds to one event, or a sequence of events that can be considered together as one unit. In the structural decomposition of a program, an operation corresponds to any section of code that can be considered as a single-entry, single-exit unit such as a subroutine. This

section presents a collection of rules for the generation and transformation of UNITY programs. With respect to these rules, if a UNITY program is considered as the source of the operations in the application of a rule, then an operation corresponds to either an assignment component, an assignment statement, or a complete program.

Regardless of the actual type of operations considered, they are grouped according to the following classifications (called *semantic relations* in the systolic array literature (173)):

Definition VI.23 *Given the two operations s and t:*

Idempotent *The operation s is idempotent, iff two successive applications of s is equal to one application of s; denoted by the predicate*

$$idem(s)$$

Commutative *The two operations s and t are commutative, iff the sequential application of s then t is equal to the sequential application of t then s; denoted by the predicate*

$$comm(s,t)$$

Independent *The two operations s and t are independent, iff the application of s and t simultaneously is equal to either the sequential application of s then t, or the sequential application of t then s; denoted by the predicate*

$$ind(s,t)$$

An immediate consequent of this definition is that the independence of two operations implies they are also commutative.

Lemma VI.24 *Given the two operations s and t,*

$$ind(s,t) \Longrightarrow comm(s,t)$$

For examples of idempotent and commutative operations, consider the class of total functions with domain and codomain equal to the natural numbers as the operations. Then for the identity function id,

$$idem(id)$$

holds, because for any natural number a,

$$id(a) = id(id(a)) = a$$

If f and g denote the two functions

$$\begin{aligned} f(a) &= a + a \\ g(a) &= a + a + a \end{aligned}$$

then

$$comm(f, g)$$

since

$$f(g(a)) = g(f(a)) = 6a$$

For an example of independent operations, consider the class of all UNITY assignments over the domain whose state vectors are denoted by

$$(x, y)$$

that is, x and y are the only two named variables. Let f denote the function that maps UNITY assignments into equivalent CSP processes (see Section 4.4). Then it follows that

$$ind(\text{x := x+x,y := y+y})$$

because the arbitrary sequential application of the two operations, which corresponds to

$$\text{f(x := x+x) } \| \text{ f(y := y+y)}$$

(this $\|$ is the CSP concurrency operator) yields the same result as the simultaneous appli-

cation of the two operations, which corresponds to

$$x := x+x \parallel y := y+y$$

(this $\parallel$ is the UNITY assignment component separator). An example of nonindependent operations is given by the *true* value for

$$\neg ind(x := y, y := x)$$

since the application denoted by

$$x := y \parallel y := x$$

which corresponds to the simultaneous application of the two asignments, is not equal to the application denoted by

$$f(x := y) \parallel f(y := x)$$

which corresponds to the two possible sequential applications of the two assignments. In the first case, the execution of the two assignment components yields the swapping of the values for x and y, whereas in the second case, the sequential execution of the two assignments yields either both x and y equal to the original value of x, or equal to the original value for y. Note that the UNITY program containing only the single statement

$$x := y \parallel y := x$$

does not reach a fixed point, but instead repeatedly swaps the values of x and y back and forth. However, the UNITY program containing only the two statements

$$x := y \mid y := x$$

does reach a fixed point, with the values of x and y equal.

These examples imply that there exists a direct relationship between these types of assignment operations and their formulation as UNITY programs and program fragments.

Theorem VI.25 *Given the UNITY assignment statement r, and the two UNITY assignment components s and t, then for any UNITY program containing the assignment statement r,*

$$idem(r)$$

holds, which implies that the UNITY program containing only the assignment statement r is equivalent (see Definition VI.20) to the UNITY program containing only the assignment

statements

$$r \mid r$$

If

$$ind(s,t) \wedge idem(s) \wedge idem(t)$$

then the UNITY program containing only the assignment statement

$$s \parallel t$$

is equivalent to the UNITY program containing only the two assignment statements

$$s \mid t$$

Proof: The first part of the theorem is direct consequent of the definition of idempotence and the UNITY execution model, since

$$r \mid r$$

just represents the repeated application of r.

The second part of the theorem follows from considering that $ind(s,t)$ implies that the simultaneous execution of s and t is equal to the sequential execution of s and t in either order, say s first. But $idem(s)$ states that s can execute an arbitrary number of times before t executes. Then $idem(t)$ states that t can also execute an arbitrary number of times. Thus any execution sequence of

$$s \mid t$$

where s executes first is equal to the exeuction sequence of $s, t, s, t, \ldots$, and if t executes first then any execution sequence of these two statements is equal to the execution sequence of $t, s, t, s, \ldots$. Since independence implies commutativity, then the execution sequence with s first equals the one with t first, which both equal the repeated simultaneous execution of s and t resulting from

$$s \parallel t$$

■

Theorem VI.25 supplies the initial result needed to address the mapping from an informal specification into a UNITY program. This mapping is based on the relationship between

idempotence and sequential operations, the relationship between independence and parallel (concurrent) operations, and the relationship between commutativity and both sequential and parallel operations. The following rules supply the basis for this mapping, based on UNITY programs that consist of a declare, initially, and assign sections. Whenever new assignments are added to an assign section, the corresponding declarations and initializations must be added to the declare and initially sections, with the assumption that no new declaration or initialization conflicts with an existing one.

Consider a specification that requires the explicit sequencing of a finite number of steps that constitute the operation *s*. One technique for mapping this sequencing to a UNITY program is to imbed *s* in the always section of the UNITY program (see Section 5.3.2 of Chandy and Misra (64)). Unfortunately, this process can become unwieldy whenever this UNITY program contains other equations in the always section, or contains other assignments in the assign section, because of the lack of theorems regarding the superposition of assignments and equations. A primary reason for a lack of such results is probably the lack of a direct correspondence between the equations of an always section and the trajectories generated by the assign section. Another problem is that while syntactically correct assignments can be added without interfering with the program execution (although the program properties may change), the addition of equations into the always section can cause the equations to contradict the assignment executions. Because of this, this research does not recommend that the always section be used to implement sequentially executing algorithms. Instead, the recommendation here is to use assignments, such as the type used in Chandy and Misra to implement specified sequential execution of nested loops (Section 5.3.3 of Chandy and Misra). The technique used by Chandy and Misra is to map a loop into the assignment statement

$$s \parallel t$$

where s denotes an assignment component equivalent to the assignments inside the loop, while t denotes an assignment component that increases the loop (for a nested loop t increases all appropriate loop indices) index or counter. Since the mapping to the UNITY program yielded a form (i.e. the s and t) for the loop that is designed to execute sequen-

tially any number of times, then

$$idem(s \parallel t)$$

This implies that the UNITY program consisting of just one such statement can be rewritten with multiple copies of this statement, without affecting any of the program trajectories. This leads to the first rule of specification composition:

Rule 1 *For the UNITY program F containing the assignment statement s, add idem(s) to the fact base for F.*

The *fact base* is a set of assertions regarding the idempotence, commutativity, and independence of portions of the UNITY program.

If the specification consists of multiple sequential algorithms, then these can be added into an existing UNITY program if certain constraints are satisfied. The concept is that each added piece must not alter an already existing trajectory, so not to affect the program properties that have already been proven, and the additional trajectories must also satisfy any already proven program properties. These constraints can be worded in terms of the superposition transformation, the General Form of Superposition Theorem, and the Superposition/Union Theorem of Section 6.2. The technique is to convert each sequential algorithm into a separate UNITY program, and then to either union the separate programs, or else form a superposition of the separate programs.

Rule 2a *Given two specification operations s and t, create two UNITY programs F and G that correspond to s and t respectively, such that F and G share no named variables. The UNITY program H that consists of the union of the* declare, initially, *and* assign *sections of F and G preserves all of the program properties of F and G.*

The partial justification that *H* preserves all of the program properties of *F* and *G* is given in this next lemma.

Lemma VI.26 *If the UNITY programs F and G share no named variables, then the domain of the UNITY program formed by the union of the two programs F and G is the sum of the domain of F and the domain of G.*

Proof: If program F contains the variables

$$x_1, x_2, ..., x_n$$

and program G contains the variables

$$y_1, y_2, ..., y_m$$

then the domains of F, G, and H are, respectively:

$$(x_1, x_2, ..., x_n)$$

$$(y_1, y_2, ..., y_m)$$

$$(x_1, ..., x_n, y_1, ..., y_m)$$

■

The remainder of the justification for Rule 2a is the Superposition/Union Theorem. Since Rule 5 can be applied multiple times simultaneously (see the UNITY implementation of Rule 5 in " UNITY program Search in Figure 6.4), then Rule 5 and 6 can be used in conjuction with Rule 2a to add assertions of independence into the fact base of the program H of the form

$$ind(s, t)$$

where s is any statement of F, and t is any statement of G. With these independence assertions, Rule 7 and 8 can then be used to modify the program H without changing any of the program properties.

Part b of this rule gives the additions to the fact base for the resulting program H. As a result of applying part b, Rule 7 and 8 can be used to modify the assignment statements of the program H, just as they can be used in conjuction with part a and Rule 5 and 6 to modify the assignment statements of the program H.

Rule 2b *For every pair of assignment statements s, t, such that s is in program F, and t*

is in program G, add $ind(\hat{s}, \hat{t})$ *to the fact base for H.*

With respect to Rule 2, if the operations s and t are required to pass information back and forth, this can be done by adding a third operation, say u. The idea behind u is that it uses a set of variables that includes the (set) union of a subset of those used by s and a subset of those used by t. Thus s and t communicate by *shared variables*, where the shared variables are among the variables used by u. The philosophy embodied by u is given in the statement by Chandy and Misra that "interfaces among modules be narrow, limited to the sharing of a few variables or based on a small number of assumptions about the shared variables." (64) This philosophy motivates the next design rule.

Rule 3a *Given three specification operations s, t, and u, such that the UNITY programs F and G, that correspond to s and t respectively, contain no shared named variables, the UNITY program H that corresponds to u and shares named variables with both F and G, and the UNITY program* $\bar{H}$ *that results from the replacement of any assignment in H to a shared variable with a nondeterministic assignment to that variable; create the UNITY program* $\hat{F}$ *by the union of the* declare, initially, *and* assign *sections of F and* $\bar{H}$, *and create the UNITY program* $\hat{G}$ *by the union of the* declare, initially, *and* assign *sections of G and* $\bar{H}$. *The UNITY program* $\hat{H}$ *that consists of the union of the* declare, initially, *and* assign *sections of F and G and H, preserves all of the program properties of* $\hat{F}$ *and* $\hat{G}$.

This rule states that for two separate programs that share a common assignment statement (the communication statement), but otherwise have no common named variables, then the union of these two programs preserves certain properties of the separate programs. These preserved properties are those resulting from the replacement of the assignments to the communication variables in the common assignment statement, with nondeterministic assignments, subject to the constraints of the variable declarations. This implies that the declarations of these shared variables should impose what Chandy and Misra call the "small number of assumptions" about their permissible values. The following lemma supplies some of the justification for this statement about this preservation of properties.

Lemma VI.27 *If the UNITY program $\hat{F}$ contains only the two assignment statements*

$$\hat{s}|\hat{u}$$

and the UNITY program $\hat{G}$ contains only the two assignment statements

$$\hat{t}|\hat{u}$$

where the statements $\hat{s}$ and $\hat{t}$ share no named variables, and all of the assignments within the statement $\hat{u}$ to the named variables common to $\hat{u}$ and $\hat{s}$, or common to $\hat{u}$ and $\hat{t}$, are nondeterministic, and the UNITY program $\hat{H}$ is equal to the union of the programs $\hat{F}$ and $\hat{G}$ such that every nondeterministic assignment is replaced with a deterministic one (it doesn't matter what the replacement assignments are), then every trajectory of $\hat{H}$ restricted to the domain of $\hat{F}$ is equal to a trajectory of $\hat{F}$, and every trajectory of $\hat{H}$ restricted to the domain of $\hat{G}$ is equal to a trajectory of $\hat{G}$.

Proof: The program $\hat{H}$ contains the assignment statements

$$\hat{s}|\hat{t}|u$$

where u denotes the statement resulting from replacing all of the nondeterministic assignments in $\hat{u}$ with deterministic ones, and the idempotence of u eliminates the need for two copies. Within any execution sequence of $\hat{H}$, if $\hat{s}$ is the next statement to execute, then whatever the current values are for the shared variables could have resulted from an arbitrary assignment during the execution of $\hat{F}$. Similarily, if $\hat{t}$ is the next statement to execute, then whatever the current values are for the shared variables could have resulted from an arbitrary assignment during the execution of $\hat{G}$. If u is the next executable statement, then whatever the current values are for the shared variables could have occurred during an execution of either $\hat{F}$ or $\hat{G}$. Thus any execution sequence for $\hat{H}$ with all of the executions of $\hat{t}$ removed generates a trajectory that is also a trajectory of $\hat{F}$, and if all of the executions of $\hat{s}$ are removed the resulting trajectory is also a trajectory of $\hat{G}$. Since $\hat{s}$ and $\hat{t}$ share no common

named variables, any trajectory of $\hat{\Pi}$ restricted to the domain of $\hat{F}$ can be generated by $\hat{F}$ (since the effect on the shared variables by $\hat{\imath}$ can be duplicated by arbitrary assignments), and any trajectory of $\hat{\Pi}$ restricted to the domain of $\hat{G}$ can be generated by $\hat{G}$. ■

Although this lemma only addresses programs $\hat{F}$ and $\hat{G}$ that contain just one other assignment statement besides the one denoted by $\hat{u}$, the reasoning of the proof applies to any finite number of assignment statements in either program. The General Form of Superposition Theorem completes the justification for the statement of Rule 3a. As with Rule 2a, the application of Rule 5 and 6 permits the inclusion of new independence assertions into the fact base for the program $\hat{\Pi}$, such that the application of Rule 7 and 8 yields permissible (without affecting the program properties) modifications to the program $\hat{\Pi}$.

The remainder of this rule parallels Rule 2b. Just as for Rule 2, the application of Rule 3b implies that Rule 7 and 8 can be used to modify the statements of the program $\hat{\Pi}$.

Rule 3b *For every pair of assignment statements $\hat{s}, \hat{\imath}$, such that $\hat{s}$ is in program F, and $\hat{\imath}$ is in program G, add $ind(\hat{s}, \hat{\imath})$ to the fact base for $\hat{\Pi}$.*

The argument used to prove Lemma VI.27 can be used to prove the following result.

Corollary VI.28 *If the UNITY program $\hat{F}$ contains only the assignment statement*

$$\hat{s} \| \hat{u}$$

and the UNITY program $\hat{G}$ contains only the assignment statement

$$\hat{\imath} \| \hat{u}$$

where the components $\hat{s}$ and $\hat{\imath}$ share no named variables, and all of the assignments within the component $\hat{u}$ to the named variables common to $\hat{u}$ and $\hat{s}$, or common to $\hat{u}$ and $\hat{\imath}$, are nondeterministic, and the UNITY program $\hat{\Pi}$ is equal to the union of the programs $\hat{F}$ and $\hat{G}$ such that every nondeterministic assignment is replaced with a deterministic one (it

doesn't matter what the replacement assignments are), then every trajectory of $\hat{H}$ restricted to the domain of $\hat{F}$ is equal to a trajectory of $\hat{F}$, and every trajectory of $\hat{H}$ restricted to the domain of $\hat{G}$ is equal to a trajectory of $\hat{G}$.

Proof: Follows from Lemma VI.27, since there is no differentiation between the sequential or the simultaneous execution of $\hat{s}$ and $\hat{u}$, or $\hat{t}$ and $\hat{u}$, in the argument of the proof. The resulting program $\hat{H}$ contains the assignment statements

$$\hat{s}\|u \quad | \quad \hat{t}\|u$$

■

This result and the General Form of Superposition Theorem form the basis for the next rule, which is similar to Rule 3, but lacks the parallel structure shared by Rule 2 and Rule 3. Whereas Rule 3 addresses a specification whose separate modules communicate in a manner represented in UNITY as a separate assignment statement, Rule 4 addresses communication represented in UNITY as a separate assignment component within one or more of the assignment statements corresponding to the modules.

Rule 4a *Given two specification operations s and t, such that the UNITY programs F and G, that correspond to s and t respectively, contain no shared named variables except for statements of the form, for a specific û,*

$$\hat{s}\|\hat{u}$$

in the program F, and

$$\hat{t}\|\hat{u}$$

in the program G, where ŝ and û share named variables. t̂ and û share named variables, but ŝ and t̂ do not share named variables; create the UNITY program F̂ by replacing any assignment in û to a shared variable with a nondeterministic assignment to that variable, and create the UNITY program Ĝ by replacing any assignment in û to a shared variable with a nondeterministic assignment to that variable. The UNITY program Ĥ that consists

of the union of the declare, initially, *and* assign *sections of F and G, preserves all of the program properties of $\hat{F}$ and $\hat{G}$.*

Rule 4b *For every pair of assignment statements, $\hat{s}$ from the programs F, and $\hat{t}$ from the program G, that satisfy Rule 4a, add $ind(\hat{s}, \hat{t})$ to the fact base for H.*

Rule 4c *For any statement u of F, and any statement v of G, such that u and v share no named variables, add $ind(u, v)$ to the fact base for H.*

Because of the inability to state that

$$ind(\hat{s}\|\hat{u}, \hat{t}\|\hat{u})$$

with only the given information, Rule 4 cannot make as strong a statement about the inclusion of assertions into the fact base for the program H as does part b of Rules 2 and 3; thus the two 'weaker' statements of parts b and c of Rule 4. As with Rules 2 and 3, the application of Rule 5, 6, and 4a yields additional independence assertions for the program H, which can then be used in the application of Rule 7 and 8 to modify H without changing its program properties. Applying Rules 4b and 4c in conjunction with Rule 7 and 8 also generates permissible modifications to the program H without altering its properties.

With respect to Rule 4c, since a program property of F does not depend upon the values of the variables of the statement v, and a program property of G does not depend upon the values of the variables of the statement u, the independence of u and v is true. This is because if u and v share no named variables, then any property of the individual execution of u is also a property of the simultaneous execution of u and v, any property of the individual execution of v is also a property of the simultaneous execution of u and v.

Attempting to make Rule 4c more general, by applying it in an arbitrary program to any two statements s and t that do not share variables , does not work! For example, if the execution of s forms the trajectory from state a to state b, and the execution of t forms the trajectory from state b to state c, then their simultaneous execution forms the trajectory from state a to state c. Although the variables of s do not change from b to c, and the variables of t do not change from a to b, it is possible that a program property of H could rely on the transitional state b, which is lost if the two statements are combined

into one statement using the || operator under the assumption of independence.

This next rule supplies the needed criteria for adding an assignment statement to the existing ones within a given program, while still preserving the program properties. Additionally, along with Rule 7 and 8, this rule can be used to add an assignment component to an assignment statement within a given program without changing that program's properties. This rule follows from the reasoning used to justify the previous rules.

Rule 5a *If the statement s shares no named variables with any statement within the UNITY program F, then the program G, formed by the union of the statement s with the program F, preserves all of the program properties of F.*

Rule 5b *For the statement s of Rule 5a, add idem(s) to the fact base of the program G; for every statement u of the program F, add ind(s,u) to the fact base of the program G.*

Rule 1 added assertions to the fact base of a given program, while Rules 2, 3, 4, and 5 generated new UNITY programs and added assertions to the fact base for the new programs. Rule 6 is the last rule in this first group of rules, and it also adds assertions to the fact base of a given program. Although Rule 6 can be applied at any time in the development of a UNITY program, this rule is primarily designed so that: Rule 6a is applied after all of the other rules, Rule 6b is applied after Rule 7, and Rule 6c is applied after Rule 8. Rule 6 simply states direct results of Lemma VI.24, the UNITY execution model, and the definition of independence.

Rule 6a *For any two assignment statements s and t of the UNITY program F, add comm(s,t) to the fact base for F.*

Rule 6b *For any two assignment statements r and s, such that ind(r,s) is an element of the fact base for F, add the two assertions ind(r,r||s) and ind(s,r||s) to the fact base for F.*

Rule 6c *For any three assignment statements r, s, and t, such that ind(r,s), ind(s,t), and ind(r,t) are all elements of the fact base for F, add the four assertions ind(r,s||t), ind(s,r||t), ind(t,r||s), and ind(r||s,s||t) to the fact base for F.*

Rule 7 and 8 are different than the first six rules, in they do not alter the fact base for a program, nor combine other programs into one. Instead, both of these rules use assertions from the fact base for a given program, to modify the assignment statements of the program, while still preserving the program properties. Rule 7 uses the independence

property of statements within a given program to arbitrarily rewrite certain statements, and is a direct consequent of Theorem VI.25.

Rule 7a *Given a UNITY program F, then for any pair of statements s and t of F such that ind(s,t) is an element of the fact base for F, removing the statements s and t plus adding the statement s||t, does not alter either the program properties or the fact base of F.*

Rule 7b *Given a UNITY program F, then for any statement s||t of F such that ind(s,t) is an element of the fact base for F, removing the statement s||t plus adding the two statements s and t, does not alter either the program properties or the fact base of F.*

Rule 8 *Given a UNITY program F, then given the assertions ind(r,s), ind(s,t), and idem(s) are elements of the fact base for F, and that the statements r and s||t are in F, then the deletion of the statement r plus the addition of the statement r||s, does not alter either the program properties or the fact base of F.*

This next example, while straightforward, demonstrates a fairly complex application of a combination of Rule 3 with Theorem VI.7 regarding subprograms from Section 6.2. Consider a UNITY program F which has been written to implement a specification operation, which primarily consists of multiple assignment statements on the variable denoted by x. One of the assignment statements of F is

$$x := \hat{P}(x)$$

such that $\hat{P}$ is a total recursive function. Although this initial UNITY specification assumes that $\hat{P}$ can be implemented, it is desired to encode $\hat{P}$ into UNITY and include this into the program F. If a self contained UNITY program P is written for the function $\hat{P}$, then how is P to be added into F? The first step is based on Definition VI.6. Thus the UNITY program P is written as a subprogram, such that P and F share no named variables, with the variable denoted by z representing the input to, and final value of, $P(o)$. Thus $P(x)$ is the fixed point value of z in the subprogram P with z initialized to x.

Thus the two UNITY programs F and P represent the specification operations s and t from Rule 3a. This implies that the link between the two programs, corresponding to the

specification operation u, must be designed. In this case the u is based on the assignment statement

$$x := \hat{P}(x)$$

which is in the program F. Since at the conclusion of Rule 3a the UNITY program H corresponding to this u is unioned with the program F, at this stage the program F is modified by removing this statement. This is the F used to complete the example.

What is needed is some way to pass the value of x into the subprogram P by assigning the value of x to z, and after P reaches the fixed point, assign the value of z to x. Additionally, once the assignment of x to z is made, the assignment of x to z or z to x should not occur again until P has reached the fixed point. By not assigning x to z again until the fixed point is reached, ensures that P reaches the fixed point in the predicted number of statement executions, and not assigning z to x again until the fixed point is reached minimizes the analysis of how to implement the 'nondeterministic' assignment to the shared variable x. These constraints can be satisfied in the following manner. First modify P into the program G as given in the proof to Theorem VI.7. This means that G contains the new boolean variable flag initialized to *false*, the new integer variable k initialized to 0, and an additional statement (compared to P)

flag := *true* if k=M

along with the modifications to the original assignment statements of P as given in the proof of the theorem. The value of M is the maximum number of statement executions of P needed to reach the fixed point.

The next step is to encode the UNITY program H (from Rule 3a) corresponding to the u operation of x := P(x). This program is given by:

Program H

declare

suspend : boolean

initially

suspend = *false*

assign

x,flag := z,*false* if flag || z,suspend := x,*true* if ¬ suspend

end

One additional change is needed, which is to replace the statement

flag := *true* if k=M

in the program G, with the statement

flag := *true* if k=M || suspend := *false* if k=M

which is justified by the assertion

ind(flag := *true* if k=M,suspend := *false* if k=M)

and the fact that suspend is not a named variable of the program G (Rule 5). This modified G is the G used in the remainder of the example.

Since the result (on the variable x) of any execution of the assignment statement in H, is also a result of some execution of the nondeterministic statement

x := x if random ~ z if ¬ random

where random is a nondeterministic boolean valued variable not named in either F, G, or H, then H (from Rule 3a) is formed by replacing the assignment statement of H with this nondeterministic one. This is because nondeterministic assignments to z, flag, and suspend are not needed, since the only property of G that is needed is that it reaches a fixed point for any possible legal input value z, which is not changed by a nondeterministic assignment to z; and neither do the values of flag and suspend affect this property of G.

As stated in Rule 3a, the programs $\hat{F}$ and $\hat{G}$ are formed from the union of the programs F and H, and from the union of the programs G and H, respectively. Consequently, the program $\hat{H}$, which is formed by the union of the programs F, G, and H, preserves all of the program properties of F and G. The only program property of G that is required is that it reach the same fixed point for any initial value of z as does the original G, which

is true because the value of z is not changed until the fixed point is reached. Since F was formed by removing the assignment statement x := P(x) from the original version of F, the union of F with $\bar{H}$ to form $\hat{F}$ is equivalent to replacing this assignment in the original F with the nondeterministic assignment statement from $\bar{H}$. What this means is that if the properties of the program resulting from replacing the statement

$$x := P(x)$$

in the original version of F with the nondeterministic statement

$$x := x \text{ if random} \sim P(x) \text{ if } \neg \text{ random}$$

do not conflict with the desired properties of the original Γ, then the program $\hat{H}$ preserves all of the desired properties of the individual specification operations.

6.4 UNITY Program Search

The generation and transformation of UNITY programs as formal specifications using these rules follows the flow of a heuristic search. Starting with the root node (of the search tree) which corresponds to some combination of informal specification and/or UNITY program(s), some subset of these six rules are applied generating one or more UNITY programs, which represents a new node in the search tree. Which of the rules to apply requires some decision based on the nature of the operations represented in the informal specification and/or UNITY program(s). This process continues, such that for any node in the search tree, an application of a subset of the rules yields another new node. The decisions as to which rules to apply at a given stage of the search can be guided by heuristics, i.e. guidelines that indicate one choice is better than another.

If such a search process can be designed, then automating it (to some degree) would logically follow, since this automation would represent a major contribution to the design of (parallel) computer programs. A first step in the attempt to automate such a search process is the formal specification of the search. This implies that a UNITY program should be written to specify how such a heuristic search can be conducted on UNITY programs, that is, a UNITY program that can manipulate other UNITY programs. Unfortunately, UNITY does not have the expressive power of the second order predicate logic, but is constrained to the first order predicate logic (see Appendix C). Thus, some method must be found

for UNITY programs to manipulate other UNITY programs in an indirect manner. This section takes the first step towards this automation by presenting such a UNITY program: This program both represents a formal specification of the initial version of such a search (without the heuristics), and also incorporates a novel technique for manipulating other UNITY programs (as part of the search process).

If the heuristic based decision capability to choose which rules to apply is removed from this search process, and instead each new node is generated from a given node by the random application of one of the rules (assuming the rule can be applied), then the resultant search tree is a *random search tree*. The collection of all of the nodes from all possible random search trees represents all of the possible UNITY programs that could be generated from the original root node through the application of these rules. The UNITY program Search in Figure 6.4 generates all such possible random search trees. Note that the variable names used in the assignments are chosen to closely match those used in the wording of the rules.

The UNITY program Search manipulates other UNITY programs. Since UNITY was not designed as a second order language (i.e. not designed to allow UNITY programs as named variables), some type of data structure must be used to represent these UNITY programs. The choice made here is to use the *set* structure, with assignment statements denoted as the elements of the sets. Certain properties of UNITY programs are naturally condusive to the set structure. For example, assignment statements can be written in any order, corresponding to the same property for the elements of a set. Additionally, adding multiple copies (or deleting multiple copies) of any assignment statement to (from) a UNITY program does not change any of the program properties, which corresponds to the concept of unique elements within sets. There are two other properties of UNITY programs that are not inherent properties of the set structure, however, and these must be handled in some other manner. The technique used in the program Search is to include these two properties in the always section, as the two statements

$$s||t = t||s$$

$$s \| s = s$$

Whereas Rule 5 addresses one statement at a time, the UNITY implementation of Rule 5 in program Search applies this rule to multiple statements simultaneously, given that each one individually satisfies the constraint of Rule 5. This UNITY version of Rule 5 resulted from applying the Search program to the original version of the rule, which was in the form

$$\hat{H} := F \cup \{s\} \text{ if } R5a \ldots$$

However, for multiple statements that satisfy R5a, this single assignment can actually be written as multiple statements of this form, each one with unique named variables. This results from applying Rule 5 to the version of the Search program containing this original version of Rule 5! Rule 3 can then be applied to the Search program with these multiple copies of the Rule 5 statements, which yields the version of Rule 5 shown in Figure 6.4. Thus the program Search was actually applied to itself, generating an improved version! Although the current form of Rule 5 in the program Search could be written

$$\hat{H} := F \cup G \text{ if } R5a \ldots$$

the use of the

$$\langle \cup s : R5a :: \{s\} \rangle \ldots$$

instead of the set G, is to give some insight as to how the set should be constructed.

6.5 Summary

Chandy and Misra's UNITY is used for the formal specification language in this research effort because of its similarity to standard imperative programming languages, and also because the UNITY execution model has an inherent temporal basis. This temporal aspect permits both reasoning about the temporal properties of UNITY specifications (programs), and the design of UNITY specifications that exhibit certain temporal properties. In addition to its use as a specification language, UNITY can be used as a computation

model and has the flexibility to model many other computation models, such as shared variable, message passing, synchronous, and asynchronous models.

The basic concept behind UNITY is that of a shared variable computation model, with all variables either global or locally scoped within quantified statements. Since a major classification of parallel computers includes those with separate processors each having their own local memory (such as the Intel Hypercube), this chapter introduces the *shared variable execution model* for UNITY programs written in an extended syntax. This new execution model includes the concept of a mapping of the statements of a UNITY program onto separate processors that each contain their own memory and execute asynchronously. An additional constraint imposed on this execution model is that no two assignments (processors) can execute simultaneously. The additional syntax includes two additional operators, one delineates which assignments map to a given processor, while the other indicates the sequencing of assignments that are mapped to a given processor. An interesting consequent of this execution model is that any UNITY program (written in the Chandy and Misra syntax) can be rewritten in the syntax of the shared variable model, such that the UNITY program under the UNITY execution model and the shared variable program under the shared variable model both generate the exact same set of possible sequences of states; that is the two programs exhibit identical behavior. Another consequent of this execution model is that the shared variable model mapping of any UNITY program preserves the program properties of the UNITY program.

This chapter also applies concepts from dynamical systems to the semantic analysis of UNITY programs. This new approach (for UNITY programs), termed *state space semantics*, is based on the analogy between a UNITY program state (as defined by Chandy and Misra) and the 'state' of a dynamical system, along with the analogy between a UNITY program and a nonlinear discrete operator (on states). As shown in this chapter, any UNITY program can be represented as such a nonlinear operator, an operator that is Turing computable. Considering a UNITY program as a dynamical operator, the sequence of states resulting from the program's execution corresponds to the concept of a *trajectory* for a dynmaical system. One result of this approach is that the UNITY concept of a fixed point is analogous to the stable equilibrium points of dynamical systems.

An important new concept presented in this chapter, resulting from the UNITY program as dynamical system paradigm, is that of a UNITY *subprogram*. A subprogram is a UNITY program that reaches a fixed point within at most a fixed number of statement executions, where the fixed number is independent of the initial state and the particular execution sequence; it's dependent only upon the program. The importance of the subprogram is that since the fixed point will always be reached in a certain number of statement executions, a subprogram can be imbedded into another UNITY program just as a subroutine can be imbedded into a standard imperative program. Additionally, the program properties of the subprogram can be preserved after being imbedded into another program.

Chandy and Misra introduce the concept of *augmentation* and *union*, which are the modification of and addition to the statements of a UNITY program, respectively. Their *superposition theorem* states that if augmentation and union follow the *rules of superposition*, then the augmented/unioned program preserves all of the program properties of the original program. Based on the concept of UNITY programs as dynamical systems, this chapter presents a modified version of the rule of superposition that leads to a more general statement of the superposition theorem called the Intermediate Form of Super position Theorem. This leads to an even more general result called the General Form of Superposition Theorem, which gives the requirement for a modified version of a UNITY program to preserve the program properties of the original in terms of the corresponding program trajectories. These results also lead to the Superposition/Union Theorem, which combines concepts from both the superposition theorem and the *union theorem* of Chandy and Misra.

This chapter presents a new approach to developing UNITY programs, representing formal specifications, based on techniques developed for systolic array design (although Chandy and Misra address the specification of systolic arrays using UNITY, the material presented here concerns the development of UNITY programs, not systolic arrays). The techniques presented here are applicable to both the generation of a UNITY program, or the transformation of a UNITY program into another UNITY program while preserving the program properties of the original. The key idea is to consider the assignment state

ments of a UNITY program as the *operations* of the systolic array literature, and then to develop rules for transforming or generating UNITY programs based on certain relationships between the assignment statements known as *semantic relations*, in a similar manner to how these semantic relations have been used for synthesizing systolic arrays.

Several new results are proved, which along with the Intermediate and General Form of Superposition Theorems, and the Superposition/Union Theorem form the basis for a collection of eight rules about UNITY programs. These rules address how multiple UNITY programs can be combined so that the desired program properties of the constituent programs are preserved, and also how UNITY programs can be modified while still retaining the desired program properties. These rules are designed to facilitate some type of automation, such as an expert system, because of the use of the if-then structure of the rules. Also, the rules use a set based data structure for the UNITY programs, which relates directly to some type of computer based automation. As a start towards this automation, the UNITY program Search is given. This program serves as a formal specification for the random application of these rules to the development of UNITY programs.

As stated in Section 6.4, the UNITY program Search represents the formal specification of a first version of a search methodology designed to transform UNITY programs into other UNITY programs such that certain *program properties* are preserved by the transformations. These program properties represent the desired temporal behavior of the UNITY programs, that is, the program properties embody the temporal characteristics defined by the specifications. Consequently, the program Search represents the beginning of additional research effort that could lead to automated tools to help in the development of formal specifications for both serial and parallel algorithms for multiple hardware architectures. This additional research should include the analysis of the program properties of the Search program (which are not addressed here), and the introduction of heuristic techniques into the program.

Program Search

declare

R1,R2a,R3a,R4a,R4b,R4c,R5a : boolean

{These are *true* if and only if, the antecedent of the corresponding rule is true}

s,t : assignment statements

{Whenever assignment statements are used in a quantification, they are from the set of all statements that could satisfy the quantification}

F,G,H,$\hat{H}$: UNITY programs

{$\hat{H}$ is the UNITY program that is being developed, i.e. each new $\hat{H}$ represents another node in the search tree}

Ho,Hi : UNITY programs

{UNITY programs are represented as sets of assignment statements}

Facts : array [1..4] of fact base

{A fact base is a set of facts, the indices are integer values for the UNITY programs}

random( ) : integer

{random(n) returns an arbitrary number from 0,....n}

always

s||t = t||s

s||s = s

{The two primary axioms of the UNITY execution model needed for this program}

initially

{Initializations to F, G, and H dependent upon root node of search tree}

$\hat{H}$ = {}

Ho = {}

Hi = {}

{Initializations of R2a, R3a, R4a, R4b, R4c, R5a dependent upon initial values of the programs F, G, and H}

{The variables s and t are only used in quantified expressions, and are not initialized}

{Initializations of Fact[F], Fact[G], and Fact[H] dependent upon initial values of the programs F, G, and H}

Fact[$\hat{H}$] = {}

assign
{Rule 1}
Fact[$\hat{H}$] := Fact[$\hat{H}$] ∪ ⟨∪ s : s ∈ $\hat{H}$ :: {idem(s)}⟩ |

{Rule 2}
$\hat{H}$:= F ∪ G if R2a ‖ Fact[$\hat{H}$] := Fact[$\hat{H}$] ∪ ⟨∪ s,t : s ∈ F ∧ t ∈ G :: {ind(s,t)}⟩ |

{Rule 3}
$\hat{H}$:= F ∪ G ∪ H if R3a ‖ Fact[$\hat{H}$] := Fact[$\hat{H}$] ∪ ⟨∪ s,t : s ∈ F ∧ t ∈ G :: {ind(s,t)}⟩ |

{Rule 4}
$\hat{H}$:= F ∪ G if R4a ‖ Fact[$\hat{H}$] := Fact[$\hat{H}$] ∪ ⟨∪ s,t : s ∈ F ∧ t ∈ G ∧ R4b :: {ind(s,t)}⟩ ∪
⟨∪ s,t : s ∈ F ∧ t ∈ G ∧ R4c :: {ind(s,t)}⟩ |

{Rule 5}
$\hat{H}$:= F ∪ ⟨∪ s : R5a :: {s}⟩ ‖ Fact[$\hat{H}$] := Fact[F] ∪ ⟨∪ s : R5a :: {idem(s)} ∪
⟨∪ t : t ∈ F :: {ind(s,t)}⟩⟩ |

{Rule 6}
Fact[$\hat{H}$] := Fact[$\hat{H}$] ∪ ⟨∪ s,t : s ∈ $\hat{H}$ ∧ t ∈ $\hat{H}$ :: {comm(s,t)}⟩ ∪
⟨∪ r,s : ind(r,s) ∈ Fact[$\hat{H}$] :: {ind(r,r‖s),ind(s,r‖s)}⟩ ∪
⟨∪ r,s,t : ind(r,s) ∈ Fact[$\hat{H}$] ∧ ind(s,t) ∈ Fact[$\hat{H}$] ∧ ind(r,t) ∈ Fact[$\hat{H}$] ∧ ::
{ind(r,s‖t),ind(s,r‖t),ind(t,r‖s),ind(r‖s,s‖t)}⟩ |

{Rule 7}
{This statement constructs the set of statements to be removed from $\hat{H}$, denoted by Ho, and the set of statements to be put into $\hat{H}$, denoted by Hi. For every s|t or s‖t that is removed (i.e. put into Ho), a random selection of either s|t or s‖t is put back in its place (i.e. put into Hi)
Ho := ⟨∪ s,t : ind(s,t) ∈ Fact[$\hat{H}$] ∧ s ∈ $\hat{H}$ ∧ t ∈ $\hat{H}$ :: {s.t}⟩ ∪
⟨∪ s,t : ind(s,t) ∈ Fact[$\hat{H}$] ∧ s‖t ∈ $\hat{H}$ :: {s‖t}⟩ ‖
Hi := ⟨∪ s,t : ind(s,t) ∈ Fact[$\hat{H}$] ∧ s ∈ $\hat{H}$ ∧ t ∈ $\hat{H}$ :: {s,t} if random(1) = 0 ~
{s‖t} if random(1) = 1⟩ ∪
⟨∪ s,t : ind(s,t) ∈ Fact[$\hat{H}$] ∧ s‖t ∈ $\hat{H}$ :: {s.t} if random(1) = 0 ~
{s‖t} if random(1) = 1⟩ |
$\hat{H}$:= ($\hat{H}$ - Ho) ∪ Hi |

{Rule 8}
{This statement uses Ho and Hi as Rule 7 does. Given the facts ind(r.s),ind(s,t), and idem(s), the two statements r and s‖t can arbitrarily be converted into the statements r‖s and s‖t (the fact base is unchanged)}
Ho := ⟨∪ r,s,t : ind(r.s) ∈ Fact[$\hat{H}$] ∧ ind(s.t) ∈ Fact[$\hat{H}$] ∧ r ∈ $\hat{H}$ ∧ s‖t ∈ $\hat{H}$ ::
{r.s‖t}⟩ ‖
Hi := ⟨∪ r.s.t : ind(r.s) ∈ Fact[$\hat{H}$] ∧ ind(s.t) ∈ Fact[$\hat{H}$] ∧ r ∈ $\hat{H}$ ∧ s‖t ∈ $\hat{H}$ ::
{r.s‖t} if random(1) = 0 ~ {r‖s.s‖t} if random(1) = 1⟩ |
$\hat{H}$:= ($\hat{H}$ - Ho) ∪ Hi |

{New Search Node Assignments} {Once a new $\hat{H}$ has been generated. then all of the variables can be reinitialized accordingly}
F,G,H,R2a,R3a,R4a,R4b,R4c,R5a,Fact[F],Fact[G],Fact[H] := {new values} if $\hat{H} \neq \{\}$ ||
$\hat{H}$,Fact[$\hat{H}$],Ho,Hi := {},{},{},{} if $\hat{H} \neq \{\}$ |

end

Figure 6.4. UNITY Program Search

VII. Conclusions and Recommendations

The major goals of this research effort are twofold: first, to demonstrate that much of the formal, mathematical analysis in theoretical computer science can be recast in terms of the topology of complete metric spaces; and second, to develop a methodical, transformation based technique for developing UNITY programs that represent formal specifications.

7.1 Conclusions

With respect to the first major goal, the topology of complete metric spaces does provide a tool that can be used to both recreate important results about computational models, and to develop new results about these same computational models. The supporting mathematics used in this research represents the type of topics, and level of presentation, of an advanced undergraduate or graduate level course in real analysis, such as from the text by Apostol (8).

This research presents the *class* as the basic concept of grouping things together, instead of the set. However, the most important results presented only required classes that were also sets. This means that a mathematical background based on sets would suffice to understand and apply the major results. The two primary uses of the non set based mathematics in this research are to develop the formal definition of a *theory*, which is based on *categories*, and the inclusion of the computational models within the category of complete metric spaces.

The complete metric spaces used in this research have as their elements instantiations of computational models. Although the metrics used in the different spaces could have been developed completely independently, that approach would have violated the underlying theme of cohesiveness that this research attempts to preserve. Consequently, this effort sought what could be called a *baseline metric*, which all of the other metrics would be based upon. This search was successful, and resulted in the metric (the metric is not new) used to develop the metric space whose elements form the *free monoid* (138) of symbol

strings based on a finite alphabet of symbols. Since computational models are also based on symbol strings, this metric space is a logical choice for a baseline metric space.

It is shown that although the traditional concept of completeness applied to this metric space yields symbol strings that could not be generated by any computing device (Turing machine), all of the *computable* strings are elements of this complete metric space. Consequently the *Turing computable complete metric space* is defined, a new definition that equates the completion of a metric space to only those Cauchy sequences which are computable. An intriguing result is proved that demonstrates the connection between this Turing computable complete metric space and the original analytical work by Scott (311) regarding the set of all possible finite and infinite strings of symbols formed from a finite alphabet. This work by Scott eventually led to the type of mathematics used in computer science with respect to which this research presents an alternative. Additionally, the metric space of symbol strings is shown to satisfy the category based definition of a *theory*.

This investigation demonstrates that other metric spaces based on diverse computational models can be developed from this baseline metric space of a monoid of symbol strings. This is first shown for the *finite automaton*, a computational model that can represent the computational power of any finite (i.e. real) machine. The metric defined for this space of finite automata is a direct application of the one defined for the symbol strings. Further, this metric is shown to be consistent with an intuitive concept of 'closeness' between two finite automata.

Several results are proven that connect the concept of the completion of the metric space of finite automata with the standard hierarchy of computing machines. These results yield a hierarchy of sets, which is shown in Figure 7.1. In the figure, the term 'completed regular' refers to the sets accepted (or generated) by machines corresponding to the computational power of the Turing computable completion of the metric space of finite automata. The inclusion of regular sets in recursive sets is strict (i.e. there are recursive sets which are not regular), but neither the strictness of the inclusion of recursive sets in completed regular sets, nor the strictness of the inclusion of completed regular sets in recursively enumerable sets is proven.

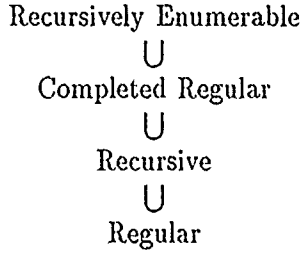

Figure 7.1. Topolgical Based Hierarchy of Sets

The *communicating sequential proccssses* (CSP) (165) computational model is also analyzed using this complete metric space viewpoint. A metric space of deterministic CSP processes is developed using basically the same metric used in the metric space of finite automata. Certain major definitions from Hoare's book on CSP are shown to be equivalent to standard definitions from metric space analysis. For example, the definition of a continuous function that maps CSP processes into CSP processes is shown equivalent to the metric space based definition of continuous functions. And the CSP concept of a *constructive* function mapping processes into processes is shown equivalent to the definition of *contraction* mappings over a metric space. Based on this equivalence between constructive and contractive mappings, this research demonstrates that what Hoare termed the "fundamental theorem" of deterministic processes is equivalent to the contraction mapping theorem, a major result from metric space analysis. Because of this equivalence of the metric space analysis of CSP processes and the analysis from Hoare's book, this effort gives an example of one of many theorems from metric space analysis that could be added to the literature on CSP processes.

UNITY, which is classified as a computational model because of its execution model, is also analyzed using the concept of complete metric spaces. Compared to the finite automata and CSP models, the approach differed with UNITY; instead of developing a metric for UNITY programs, several theorems showed that any UNITY program can be mapped into an equivalent CSP process. The definition of equivalent requires that the UNITY program and the CSP process have the same set of atomic actions, and that a sequence of atomic actions is possible for an execution of the UNITY program, if and only

if, the same sequence is possible for some execution of the CSP process. This mapping is used to define a metric space for UNITY programs such that the metric distance between programs is evaluated by mapping the programs into equivalent processes in the CSP metric space, and then evaluating the CSP metric distance between the processes. Thus the mapping is an *isometric imbedding*, since it preserves metric distances.

Part of the analysis of computational models is the development of a *semantics* for or *reasoning* about these models. The formal system used in this effort for the semantic analysis is the temporal logic. This research shows that the *branching temporal logic* supplies the necessary tools to reason about finite automata, and also to duplicate the proof of process correctness technique used in Hoare's book on CSP (165).

With respect to the development of an executable program, the approach of proving the program correct after it has been written has not been widely accepted by the software development community. This research supports an alternate approach, which is based on proving the correctness of an informal or formal specification, and then transforming this (in)formal specification so that the original proof of correctness is preserved. The second major portion of this research effort addresses this alternate approach through the use of the formal specification language UNITY.

To facilitate reasoning about the execution model for UNITY, another semantic model called the *state space semantics* is introduced, that is based on the concept of UNITY programs as nonlinear dynamical systems. This state space semantics is shown to lead to intuitive proofs of statements from Chandy and Misra's book (64), and to also permit short proofs of theorems that represent more inclusive versions of theorems from Chandy and Misra. The state space semantics is used to define the new concept of a UNITY *subprogram*. The subprogram is used much as a subroutine would be in a sequential language.

State space semantics permits analysis of UNITY programs that is more *black box* oriented, i.e. the internal structure of the UNITY program is not so important, versus the *white box* approach of Chandy and Misra, i.e. more concerned with the actual structure of the assignments within a UNITY program (264).

State space semantics is combined with the concept of *independent*, *commutative*, and *idempotent* properties of assignment components and statements from UNITY programs, to develop an algorithmic approach to the generation and transformation of UNITY programs. The generation is from either informal versions (specifications) of the programs, or from multiple programs that need merging into one program. This algorithm is designed such that both the merging of multiple programs into one, and the transformation of a program into another program, are accomplished while preserving the desired program properties of the ancestor program(s). This algorithm is presented as a UNITY program, called Search, which was itself transformed into its final form using the original version!

7.2 Recommendations

With respect to the first major goal of this effort, the topological analysis of computational models demonstrated that major results pertaining to these computational models, can be duplicated and augmented using the basic concepts of complete metric spaces. However, the following theoretical questions remain unanswered by this research effort:

1. The topological analysis of finite automata yielded partial results connecting the classical hierarchy of computational machines, and a hierarchy based on complete metric spaces. One missing piece of this connection is whether the computable completed metric space of finite automata is equal in computational power to the space of Turing machines, instead of the inclusion relation shown.
2. The topological analysis of CSP generated results that had been proven using other techniques, plus new results. But this analysis is restricted to the deterministic processes only, and should be extended to nondeterministic processes.
3. The metric space of UNITY programs is based on an isometric imbedding (into) from UNITY programs into the metric space of CSP processes. It remains to be shown whether or not this mapping is also surjective (onto). Two other unanswered questions are: does there exist a homeomorphism between the two metric spaces, and does the completion of the metric space of UNITY programs include elements that are not UNITY programs.

The second major goal of this research effort develops theory and techniques for developing UNITY programs as formal specifications. In addition to the many accomplishments, there still remain many areas for further research:

1. The *shared variable model* for UNITY programs presents the groundwork for a more comprehensive approach to the mapping of UNITY programs to actual machine architectures, such that the syntax of the UNITY program guides the mapping. Other syntax-execution model combinations could be developed for various types of architectures.
2. The analogy of UNITY programs as dynamical systems can also be developed further. Can concepts such as *stable* and *unstable equilibrium points* be applied to the fixed points of UNITY programs? What constraints must a UNITY program satisfy with respect to the state space semantics so that certain properties of the program, such as a reachable fixed point, are valid? What are the state space equivalents to the theorems from parallel programming, such as the Asynchronous Convergence Theorem (39)?
3. The replacement of nondeterministic assignment components for standard deterministic UNITY assignment components can be further developed, and should lead to additional rules which can be added to the program Search.
4. The additional evolution of the program Search should occur, especially since this program can evolve by applying its rules to itself. What are also needed are the necessary and sufficient constraints that force Search to reach a fixed point.

7.3 Final Comments

The topological analysis of computational models presented in this thesis represents one solution to a current problem at schools such as the Air Force Institute of Technology (AFIT). This problem is either the lack of mathematics courses to support high level theoretical computer science, and/or the lack of room in the students schedules for such courses. These math course would have to cover such topics as *lattices*, *domains*, *signature algebras* and *category theory*, so as to correspond to the current trends in computer science

research. The topological analysis presented here has duplicated many of the major results from the theoretical analysis of computational models, plus has supplied new results. Thus theoretical computer science research can be accomplished without introducing any new courses, but instead by having the student take a real analysis sequence.

The development of the program Search represents a milestone in the evolution of UNITY into a useful framework for the design and implementation of parallel algorithms. In its present form Search can provide the software engineer with the techniques needed to develop UNITY programs as formal specifications. However, Search limits the range of informal specifications, and the range of possible collections of UNITY programs to be merged, that a single UNITY formal specification can be developed from. Since the Air Force will rely even more on parallel algorithms as future airborne and ground based computer systems evolve into more complex distributed systems, the continued improvement of the Search program and the ideas it represents, will only enhance the Air Force's ability to design and manage software for these systems.

Because, finally, he said:
"This is really great stuff!
"And I guess the old alphabet
"ISN'T enough!"

Appendix A. *Relation Based Logic: Modal, Temporal, and Predicate*

The first section from this appendix presents a description of modal logic, which is used for both the specification and verfication of programs (2, 30, 257). This introduction includes the primitive modal unary operator $\Box$ (henceforth), along with the derived unary operator $\Diamond$ (eventually) and the derived binary operator $\rightsquigarrow$ (leadsto). The basic concept behind any interpretation of the meanings of the modal operators, the *frame*, is defined and illustrated with examples using different types of binary relations and underlying sets. Definitions are given for the terms *syntax, interprctation*, and *semantics*, and the syntax, semantics, and possible interpretations are presented for the modal logic. This presentation stresses the generalized nature of this definition of a modal logic, which leads to the definition in the second section of a temporal logic as a modal logic with a specified type of underlying set and binary relation constituting the logic's frame. This section also defines the syntax and semantics of the temporal logic used within this thesis, along with possible interpretations. Additionally, the standard temporal logic primitive operators $\bigcirc$ (next) and $\mho$ (until), along with the derived operator $\mathcal{U}$ (unless) are defined. Finally, the third section presents a definition of the predicate logic as another special case of the modal logic based on certain types of frames which lead to logics without the modal operators, along with the possible interpretations, semantics and syntax of the predicate assertions used in this thesis.

Goldblatt's book (139) gives a definition of modal logic similar to the one presented here, while the book by Cresswell and Hughes (176) contains more detail about modal logic. The book by Rescher and Urquhart (300) supplies additional information about temporal logic, whereas a shorter reference for temporal logic would be Chapter 4 from the published version of Hailpern's dissertation (147). The first three chapters from Manna and Waldinger's (Volume 1) book (225) gives much more detail regarding both propositional and predicate logic, whereas Chapter 2 from Manna's book (217) presents the material in a more concise (read 'denser') manner.

In this research the symbols used to represent functions and predicates are uniquely specified without regard to arity, so that

$$P(*,*)$$

$$P(x,y)$$

$$P$$

all represent the same function or predicate of arity two. The $P(*,*)$ denotes that P has two arguments, but is not evaluated, while $P(x,y)$ represents the evaluation of P at (x,y), where the evaluation is either symbolic or instantiated. The symbol P simply represents the function or predicate, in effect giving it a name. Note that literature based on symbolic logic programming (such as PROLOG) often differentiates between P and $P(x,y)$ as two different entities (76).

A.1 Modal Logic

This section presents the definitions of the terms syntax, interpretation, and semantics, and then gives these for the modal predicate logic, which is called the modal logic within this thesis. The modal logic definition is designed so that the standard temporal and predicate logics can both be defined as subsets of the modal logic.

Given a finite set of symbols from which formulas, or strings of symbols from this set, can be formed, a *syntax* is a collection of rules defining the formation of certain formulas. If the syntax is that of a grammar, then the formulas formed from the syntax can be grouped into a set called the language of that grammar (327). With respect to the modal logic, any formula formed from the syntax is called a *well formed formula*, or wff. Thus the modal logic can be informally defined as the study of the well formed formulas. In addition to the standard syntax of the predicate logic, the modal logic adds (as a minimum) the two modal operators denoted by $\Box$ and $\Diamond$. Although these operators extend the semantics of the predicate logic, many of the proof techniques, including *resolution* (100), can be carried over from the predicate to the modal logic.

Once a syntax has been defined, the next step is the idea of an *interpretation*. Given a finite set and corresponding syntax, the interpretation is a function that maps certain symbol strings from the well formed formulas into a codomain whose elements are called the *values* of the interpretation (225). The purpose of these values is to give some meaning to the symbol strings that are not the operators within the wffs. Each different interpretation can have a different mapping, a different codomain, or both. For the modal logic within this thesis, the codomain of the interpretations includes *constants*, *variables*, and *functions*. Note that relations were not included, since any relation required by a wff will be recast as a function whose codomain is the set of atomic symbols *true* and *false*. For example, consider the relation $<$ over the natural numbers. Instead of statements of the form

$$(2,3) \in <$$

(see Section 3.2 for a description of relations), the relation is represented with a predicate function, say p

$$p : \mathrm{N} \times \mathrm{N} \rightarrow \{true, false\}$$

such that

$$p(2,3) = true.$$

The final concept is *semantics*, which is a function that maps the remaining symbols of a wff that are not in the domain of the interpretation into a fixed codomain whose elements are called the *meanings* of the symbol strings. As opposed to the interpretation, the mapping and the codomain of the semantics remains fixed for a given modal logic. Thus the semantics presented in this appendix holds for every wff within this thesis. Taken together, the interpretation and the semantics assign to every symbol string within the wff some form of meaning, so that the entire wff can be given some meaning. Within modal logic, the only possible meanings given to a wff are *true*, *false*, or undefined. So the combination of the interpretation and the semantics is a partial function whose codomain is the set $\{true, false\}$. The term interpretation is used to signify this combination of the semantics and interpretation into one function, so that the interpretation of a wff is a predicate function whose domain is the set of all wffs.

The following definition of the syntax of the modal logic differs from the standard presentation (139) in that the syntax of the predicate logic is included, instead of first defining the predicate syntax separately and then adding the syntax for the modal operators. Additionally, this appendix presents the modal predicate logic, and not just the modal propositional logic (functions, including predicate functions, can be represented as variables in the formulas).

The set of symbols and syntax is given in two steps, the first being those *primitive* symbols that are used in the definition of the initial syntax, and then the additional symbols that can be derived from the primitive symobls along with the syntax governing these additional symbols. The first set of primitive symbols consists of:

1. The upper and lower case letters of the English alphabet.

2. The arabic numerals 0 thru 9.

3. The symbol *false*.

4. The unary logic operator $\neg$.

5. The binary logic operator $\Longrightarrow$.

6. The unary modal operator $\Box$.

7. The logic binary relation $=$.

8. The arithmetic binary relation $=$.

9. The quantifiers $\exists$ and $\exists!$.

10. The punctuation symbols '(', ')', '[', ']', and ','.

The corresponding syntax for this set of primitive symbols is given by:

$$< wff > \ ::= \ < term >=< term > | false | < pred > | < wff > \Longrightarrow < wff > | \neg < wff >$$
$$| < quant >< var >< wff > | \Box < wff > | [< wff >] | (< wff >)$$

subject to the following constraints:

Right associativity: $FbGcH$ if and only if $Fb(GcH)$.

Binding strength: $uFbG$ if and only if $(uF)bG$.

where

$< wff >$ is a well formed formula.

$< term >$ is a term.

$< pred >$ is an evaluated predicate. (See Section 3.2 for the syntactic form of an evaluated predicate)

$< quant >$ is a quantifier.

$< var >$ is a variable.

and F, G, and H are wffs, b and c are binary operators (either logic or modal), and u is a unary operator (either logic or modal). This format for presenting the syntax of a wff is called *Backus-Naur form* (BNF). The ::= can be read as 'is', while the vertical bar | is read as 'or'. Thus if $< formula >$ represents any formula, and $< not - wff >$ any formula that is not a wff, then

$$< formula > ::= < wff > \mid < not - wff >$$

can be read as 'any formula is a wff or is not a wff', which is a true statement.

To conform with the standard practices for the predicate logic (217), the following lists those interpretations that will always hold true for certain symbol strings within the wffs used in this thesis. Given any wff, the following hold:

Any symbol(s) representing a constant is a term.

Any symbol(s) representing a variable is a term.

Any symbol(s) repreenting a function is a term.

Any symbol string representing an evaluated function is a term.

Although our list of terms does not include relations (see Section 3.2), we can include relations by considering them as predicates. Thus the relation $<$ on the natural numbers defines a predicate, say p, such that $p(x, y)$ evaluates to *true* if $x < y$, else it evaluates to *false*.

Additionally, certain other interpretations are used without explicitly listing them, if they represent standard mathematical or logical terms or predicates. For example, consider the wff

$$\exists x[x \in X]$$

which contains the symbol string $x \in X$, whose interpretation as an evaluated predicate implies that the entire string is a wff.

The book by Schoning (310) presents a similar definition of well formed formulas, while the book by Manna (217) presents a more comprehensive definition of wffs that

includes the 'IF THEN ELSE' and the 'if then else' operators, neither of which is needed for this thesis.

The second set of symbols contains the derived symbols and operators, in that they can be derived from the primitive symbols and operators presented above. This set contains:

The atomic symbol *true*.

The binary logic operators $\vee$, $\wedge$, $\Longleftrightarrow$, and $\longrightarrow$.

The unary modal operator $\Diamond$.

The binary modal operator $\rightsquigarrow$.

The quantifier $\forall$.

The derivations are given by:

true if and only if $\neg false$.

$F \vee G$ if and only if $\neg F \Longrightarrow G$.

$F \wedge G$ if and only if $\neg(F \Longrightarrow \neg G)$.

$F \Longleftrightarrow G$ if and only if $(F \Longrightarrow G) \wedge (G \Longrightarrow F)$.

$F \longrightarrow G$ if and only if $(F \Longrightarrow G) \wedge F$.

$\Diamond F$ if and only if $\neg\Box\neg F$.

$F \rightsquigarrow G$ if and only if $\Box(F \Longrightarrow \Diamond G)$.

$\forall x F$ if and only if $\neg\exists x \neg F$.

where F and G are wffs.

The following two examples present both a wff and a formula that is not a wff. In showing that the first is actually a wff, the combination of the syntactic analysis with a partial interpretation demonstrates that whether a formula is a wff of not may depend upon the interpretation, since syntax alone does not always resolve what a particular symbol string is. Consider that

$$F(x, y)$$

could be either an evaluated predic e, which makes this formula a wff, or an evaluated function whose domain is the natur numbers, making the formula a term. The following paragraphs present more detail regarding interpretations.

The first formula is

$$\Box\exists x[f(x) = 0 \wedge \forall y[p(y) \Longrightarrow q(x,y)]] \tag{A.1}$$

To show this formula is a wff, break it up into appropriately chosen pieces and then combine the pieces following the rules of the syntax. If $p(y)$ represents an evaluated predicate, then it is a wff, so that if $q(x,y)$ is another evaluated predicate, then

$$p(y) \Longrightarrow q(x,y)$$

is also a wff. This implies that

$$\forall y[p(y) \Longrightarrow q(x,y)]$$

is also a wff. If $f(x)$ is a term, based on letting $f(x)$ be an evaluated function, then

$$f(x) = 0$$

is a wff since the constant 0 is a term. This implies that

$$f(x) = 0 \wedge \forall y[p(y) \Longrightarrow q(x,y)]$$

is a wff, which leads to the conclusion that the entire expression A.1 is a wff. The second example is the formula

$$\exists f[f : X \rightarrow Y]$$

which is not a wff, although for any two sets X and Y it can be shown that this is a true statement with the unambiguous meaning that 'given any two sets there exists a function

whose domain is the first and whose codomain is the second'. This is not a wff because

$$f : X \rightarrow Y$$

is not a well formed formula, although it could be considered an evaluated predicate (but is not considered such for this example).

The definition of the alphabet of symbols used with the modal logic differentiated between the equality used with predicates and the equality used with arithmetic constructions even though both are the '=' symbol. This is to emphasize that the formation of the wff $s = t$ from the two terms s and t requires the class definition of equality that goes with the class (see Section 3.1) that both s and t are elements of. This also means that $s = t$ is only defined when both s and t are elements of the same class. However, the syntax does not preclude forming wffs such as

$$(s = t) = (u = v)$$

which, if both equalities inside the parantheses are undefined, yields the true interpretation that 'undefined' equals 'undefined'. In this example, the term $s = t$ is considered a constant whose value is undefined. Additionally, this syntax substitutes the binary operator $\Longleftrightarrow$ for the equivalence operator $\equiv$ used by Manna (217, 225), since $F \Longleftrightarrow G$, where F and G are wffs, is defined exactly the same as Manna's $F \equiv G$.

Now that the syntax has been defined, the next step is to define the concept of an *interpretation*. This presentation is an informal definition of an interpretation, the book by Manna and Waldinger (225) gives a formal definition that parallels this one. Well formed formulas contain symbol strings that represent both predicates and other types of functions (many texts present these other types of functions as partial functions whose domains are the natural numbers), constants and variables. These are the symbols that are the domain of the interpretation, whereas the other symbols such as the quantifiers and the binary logic operators are the domain of the semantics (see following paragraphs). An interpretation is a mapping that assigns to each constant, variable, function, and predicate symbol(s) a *value*. Informally a value is something that has meaning to us, where these assigned

values determine the *true* or *false* evauation of the wff, based on the semantics given in the following paragraphs. Thus an interpretation is a mapping from certain symbol strings within the wff into a specified domain such that the interpretation plus the semantics permits the wff to be evaluated as either *true* or *false*. Note that a *partial interpretation* assigns values to some or all of the symbol strings within the wff, but does not premit the wff to be evaluated as either *true* or *false*.

As an example, consider the following wff

$$\forall f[\exists x \exists q[p(x) \Longrightarrow q(f(x))]] \tag{A.2}$$

One interpretation consists of the following assignments:

1. f is an element of that class of functions that have a defined evaluation for at least one element of their domains.
2. p is a predicate such that $p(x)$ evaluates to *true* if x is an element of the domain of the function f, else $p(x)$ evluates to *false*.
3. q is a predicate.

Observe that an interpretation does not require the same level of detail for like objects. The variable f is described in some detail, but the variable x is not addressed at all, while the variable q is described in less detail than f. Additionally, the predicate q can be used as a variable, in which case it's represented just as 'q', and can also be represented as an evaluated predicate, as '$q(f(x))$', a symbol string that is a wff by itself. As demonstrated in the following paragraphs, this interpretation contains enough information such that the value of the wff can be determined using the following rules of semantics.

$\neg F$ means 'not F', such that if F evaluates to *true*, then $\neg F$ evaluates to *false*.

$F \Longrightarrow G$ means 'F implies G', such that if F evaluates to *true* and G to *false*, then $F \Longrightarrow G$ evaluates to *false*, else to *true*.

$\exists$ means 'there exists', such that $\exists x[F]$ evaluates to *true* if there exists some x such that F evaluates to *true*.

$\exists!$ means 'there exists a unique', such that $\exists! x[F]$ evaluates to *true* if there exists a unique x such that F evaluates to *true*.

Note that the modal operators are not defined, since they are not considered to have an intrinsic meaning. Instead, depending upon the context within which the modal logic is used, the modal operators are assigned meanings consistent with the context. One example is the *temporal logic* defined in Section A.2. Another example is a logic based on nondeterministic program termination, such that

$$\Box F$$

would mean 'After every possible termination of the program the wff F evaluates to *true*', while

$$\Diamond F$$

would mean 'After at least one possible program termination the wff F evaluates to *true*.

Also note that the meanings of the derived symbols follow directly from the primitive symbols, so that

$$\forall x[F]$$

means 'There does not exist an x such that not F is *true*', or the more accepted 'For all x F is *true*. The symbol $\Longleftrightarrow$ means 'if and only if', and the symbol $\longrightarrow$ means 'leads to the derivation of', or 'leads to the truth of'. The symbols $\vee$ and $\wedge$ mean 'or' and 'and' respectively. The derived modal operators $\sim$ and $\Diamond$ have no inherent meaning, although Section A.2 gives meanings for these modal operators within the context of temporal logic.

With these semantic definitions, the previous sample wff (see A.2) can now be assigned a meaning that permits its valuation as either *true*, *false*, or undefined. This wff is

$$\forall f[\exists x \exists q[p(x) \Longrightarrow q(f(x))]]$$

and means 'There exists a function which has a defined evaluation for at least one element of its domain such that there exists an element of its domain such that there exists a

predicate which evaluates to *true* for the function evaluated at this element'. That this wff is indeed true can be seen by letting the predicate be a test for whether its argument is an element of the codomain of the function.

The following concepts regard the correspondence between the different types of modal logic and the forms of wffs that are always true (i.e. theorems) within these logics.

The first definition is an informal one for the word *schema*. A schema is a set of wffs that all have the same form, with respect to an additional syntactic rule. For example, within the temporal logic of concurrent programs (see Section A.2) those wffs that have the form

$$\Box < wff >$$

are called invariants. Thus we could call such wffs members of the invariant schema.

The next definitions involves those different possible interpretations for which a given wff is *true*. The combination of an interpretation and the semantics establishes the evaluation of a wff as either *true* or *false*. This leads to the concept of a *domain of interpretation* as the domain for the interpretation(s) of either one wff, a schema, or a class of wffs. For example, the wff

$$\forall x \exists y [x \neq y \wedge f(x) = f(y)]$$

could have as one domain of interpretation those partial functions whose domain and codomain are a finite subset of the natural numbers, and as another domain of interpretation the two functions *sin* and *cos*. The selection of any member from either domain of interpretation is sufficient to determine the value of the wff. If all domain of interpretations are restricted to be sets, then the following definitions can be made.

Definition A.1 *A* frame *is a two-tuple* (S, R), *such that* S *is a domain of interpretation (a set), and* R *is a binary relation* $R \subset S \times S$.

This leads to the following definition.

Definition A.2 *Given the frame (S, R), a set W of wffs whose domain of interpretation is S, a* **model** *is the three-tuple (S, R, V), such that V is a function*

$$V : W \rightarrow 2^S$$

Further, given the model (S, R, V), the interpretation s, $s \in S$, the wffs v, w, $v \in W$ and $w \in W$, then the assertion

$$\models_s w$$

can be evaluated recursively as follows:

$$\neg \models_s false$$

$$\models_s \neg w \Longleftrightarrow \neg \models_s w$$

$$\models_s (v \Longrightarrow w) \Longleftrightarrow (\models_s v \Longrightarrow \models_s w)$$

$$\models_s \Box w \Longleftrightarrow \forall t[(t \in S \wedge (s, t) \in R) \Longrightarrow \models_t w]$$

$$\models_s w \Longleftrightarrow w \in V(w)$$

An equivalent definition is that V is a set

$$V \subset W$$

such that all of the above statements regarding the evaluation of

$$\models_s w$$

hold, except that

$$\models_s w \Longleftrightarrow w \in V$$

instead of

$$\models_s w \Longleftrightarrow w \in V(w)$$

Note that $\models_s$ (and the $\models$ of the next definition) binds tighter than any binary operator, so that

$$(\models_s vBw) \Longleftrightarrow ((\models_s v)Bw)$$

where B is any binary operator and v and w are wffs. Often the binding precedence is not strictly followed if the intent is clear from the text, such as when

$$\models_s vBw$$

is intended to mean

$$\models_s (vBw)$$

The nomenclature $\models_s w$ is taken to mean that the wff w evaluates to *true* at the point or interpretation s. For certain wffs in W, the function V maps a wff w into a subset of S, $V(w) \subset S$, that contains those points in S for which the wff evaluates to *true*. The equivalent definition, where V is a subset of W instead of a function, yields the concept that V contains those wffs that evaluate to *true* regardless of the interpretation s. In other words, the set V is a set of axioms that are given for the logic and hold for all interpretations. One example would be the proof laws regarding the sat operator for CSP (see Section 4.3). Since V is only a partial function, or typically only a proper subset of the set of all wffs ($V = W$ yields an uninteresting logic where every wff is true), this definition supplies a recursive algorithm that permits the evaluation of $\models_s w$ for which $V(w)$ is undefined or w is not an element of V. For example, consider the wff

$$F \Longrightarrow \Box F \tag{A.3}$$

such that

$$s \in V(F)$$

where s is an element of S. If the binary relation R, $R \subset S \times S$, is the identity relation, that is

$$\forall s, t[(s,t) \in R \Longleftrightarrow s = t]$$

then assuming that

$$\models_s F$$

for $s \in S$, implies that

$$\models_s \Box F$$

since

$$\models_s F \Longrightarrow \forall t[(s,t) \in R \Longrightarrow \models_t F]$$

because t must be equal to s for this R. Thus the wff A.3 must be *true*. This result will form the basis for the *predicate logic* in Section A.3

Note that the statement above that

$$\models_s (v \Longrightarrow w) \Longleftrightarrow (\models_s v \ \Rightarrow \models_s w)$$

is equivalent to claiming that

$$\models_s (v \vee w) \Longleftrightarrow (\models_s v \vee \models_s w)$$

since

$$(v \Longrightarrow w) \Longleftrightarrow (\neg v \vee w)$$

This implies that this appendix presents a modal logic that also possesses the *modal disjunction property* (133). since the wffs v and w can represent wffs from the schema $\Box u$.

This example along with previous analysis has used the concept of a true or false wff, without formalizing the idea. The following presents the formal definition of a true wff, and the default dual concept is that any wff that is not true is either false or undefined.

Definition A.3 *Given the model* (S, R, V), *the wff* w *is* true, *denoted by*

$$\models w$$

if and only if

$$\forall s[s \in S \Longrightarrow \models_s w]$$

A schema is true if and only if every wff within the schema is true.

The next definition abstracts the concept of truth to that of validity, which is truth for all possible models.

Definition A.4 *Given the frame* (S, R)*, the wff* w *is* valid *if and only if*

$$\models w$$

for all models (S, R, V)*. A schema is valid if and only if every wff within the schema is valid.*

These definitions of 'model', 'true', and 'valid' which are based on the definitions used in the symbolic logic literature (139), differ from those used in the computer science literature. The definition of true from Definition A.3 corresponds with the computer science definition of 'valid' as a wff that evaluates to *true* for every interpretation, while the computer science literature defines a model as an interpretation that results in the *true* evaluation of a wff (40).

Based on these definitions, the following theorem presents a result from the modal logic that is used in the definition of predicate and temporal logic in the following sections. The proof given here augments the partial proof presented in Goldblatt's book (139).

Theorem A.5 *Given a set* S*, a binary relation* R*,* $R \subset S \times S$*, the following properties of* R *where* s*,* t*, and* u *are elements of* S*:*

1. Reflexive $\forall s[(s,s) \in R]$
2. Symmetric $\forall s,t[(s,t) \in R \Longrightarrow (t,s) \in R]$
3. Directed (serial) $\forall s \exists t[(s,t) \in R]$
4. Transitive $\forall s,t,u[((s,t) \in R \wedge (t,u) \in R) \Longrightarrow (s,u) \in R]$
5. Functional $\forall s \exists! t[(s,t) \in R]$
6. Weakly Dense $\forall s,t[(s,t) \in R \Longrightarrow \exists u[(s,u) \in R \wedge (u,t) \in R]]$

the following schemata where w is any wff whose domain of interpretation is S:

1. $\Box w \Longrightarrow w$
2. $w \Longrightarrow \Box\Diamond w$
3. $\Box w \Longrightarrow \Diamond w$
4. $\Box w \Longrightarrow \Box\Box w$
5. $\Diamond w \Longleftrightarrow \Box w$
6. $\Box\Box w \Longrightarrow \Box w$

such that (S, R) is a frame, then if R satisfies any of the properties 1-6 given above, the corresponding schema is valid. Conversely, if any of the schemata 1-6 given above are valid, then R satisfies the corresponding property.

Proof: The proof is given for the symmetric property, the others follow the same pattern. Consider first that R is symmetric, and (S, R, V) is any model based on the frame (S, R). Then for any $s \in S$ (assuming non empty S so that the proof is nontrivial), show that

$$\models_s w \Longrightarrow \models_s \Box\Diamond w$$

From Definition A.2 the right hand side of this implication can be written as

$$(s,t) \in R \Longrightarrow \models_t \Diamond w$$

or, since $\Diamond w$ is $\neg\Box\neg w$

$$(s,t) \in R \Longrightarrow \models_t \neg\Box\neg w$$

or

$$(s,t) \in R \Longrightarrow \neg \models_t \Box\neg w$$

which can be written as

$$(s,t) \in R \Longrightarrow \neg((t,u) \in R \Longrightarrow \models_u \neg w). \qquad (A.4)$$

Since R is symmetric, then

$$(s,t) \in R \Longrightarrow (t,s) \in R$$

and substituting s for u in A.4 yields

$$(s,t) \in R \Longrightarrow \neg((t,s) \in R \Longrightarrow \models_s \neg w)$$

which is true, since

$$(s,t) \in R \Longrightarrow ((t,s) \in R \Longrightarrow \models_s w)$$

The next step is to prove the converse statement. Thus given that the schema

$$w \Longrightarrow \Box\Diamond w$$

is valid, prove that R is symmetric. Given any model (S,R,V) based on (S,R), assume that $(s,t) \in R$ and that $\models_s w$. Then from A.4 the assertion

$$\neg((t,u) \in R \Longrightarrow \models_u \neg w)$$

is true, which means that

$$(t,u) \in R \Longrightarrow \models_u \neg w$$

is false. For this implication to be false, both

$$(t,u) \in R \tag{A.5}$$

and

$$\models_u w \tag{A.6}$$

must be true. Since this conclusion holds for any model, it must hold for that model that only yields w *true* at s, implying that

$$(t,s) \in R$$

must be true to satisfy both A.5 and A.6 with $u = s$. Thus the relation is symmetric, since the choice of s and t were arbitrary.

With respect to the corresponding property for the symmetric relation

$$w \Longrightarrow \Box\Diamond w$$

the proof of this theorem assumed that w represents $\models_s w$ for some $s \in S$. Since both concepts of w being true and w being valid are based on $\models_s w$, then this proof technique holds for w being interpreted as $\models_s w$, w true, or w valid. In the next section on temporal logic, the symbology w is taken to mean $w(0)$, where $w(t)$ is a predicate with a temporal (time) argument.

Note that any relation satisfying the functional property is also a function. Also note that the proof of this theorem uses the fact that

$$\models (p \Longrightarrow q) \Longleftrightarrow (\models p \Longrightarrow \models q)$$

A.2 Temporal Logic

Temporal logic is that branch of modal logic within which the modal operators have semantic interpretations with respect to a temporal domain, i.e. time (306). Thus temporal predicates or assertions have a parameter whose values come from a domain (set) that is considered time, although this domain is actually nothing more than a set along with a binary relation over the set. Within this thesis the set is always a set of numbers or tuples of numbers.

In the following formal definition temporal logic is differentiated from modal logic by the nature of the frames within which the wffs are evaluated, such that the underlying set of these frames is this set of numbers that is considered as 'time'. Although different types of sets have been used to represent the time variable, with corresponding differences in the interpretations and analyses of the resulting wffs (300, 139), this section presents the two sets used within this thesis, which result in what is called *linear* and *branching* time (115). Linear time stems from the concept of time as a single sequence of numbers, which is useful in reasoning about sequential processes, while branching time is based on multiple sequences of numbers, leading to the ability to reason about concurrent processes. Another term for branching time is *multiform time*, which includes additional (sequence) operators to those presented here (282). These two concepts are presented in Definitions A.8 and A.10. The presentation of temporal logic within this section follows closely that of Goldblatt (139), and Manna and Pnueli (219).

This thesis defines temporal logic as what is called *Stoic* temporal logic, named after the Stoics of ancient Greece. The Stoics based their logic on the concept of a relative *now*, so that all assertions about time refer to time starting with the present. This implies that the wffs can only assert truths or non truths about the present and the future, and not about the past (as opposed to the Megarians, who allowed assertions about past times) (300). This approach corresponds to the standard analysis of programs as sequences of events that are based on the first statement of the program (126, 217). Thus the time at which the first statement (or event) executes is considered the 'now' of the temporal logic used to reason about these programs. Temporal logic based on past and future times with respect to a single instant in time is also called temporal logic with *point semantics*

(263), as opposed to *interval semantics* (251). Point semantics temporal logic is used in the specification of programs, and also in the synthesis of programs in certain languages such as CSP (226).

Before presenting the formal definition of temporal logic, consider the possibilities for the set of numbers that comprises the time domain. One approach is to allow the set to be a continuum over some bounded or unbounded interval, where the concept of a continuum is a set that can be put into a one-to-one correspondence with some subset of nonzero measure of the real numbers, and satisfies the completeness axiom (305). (Disregarding any nonstandard analysis questions, such as the issue of infinitesimals (304)) Thus a continuum is a set that is uncountable. This choice implies that the probability of any two events occurring at exactly (to infinite precision) the same time is zero. This eases the analysis of concurrent computations, since it can be assumed that no two atomic actions occur simultaneously, an assumption that holds even for a countable number of processors. The drawback though, is that this time domain contains values that are noncomputable, and in some sense random (see Appendix B). Another alternative is for the time domain to be a countable set, such as the natural numbers N, yielding what is called linear time. Although all elements of the time domain are now computable, the assumption is no longer always valid that the probability of two atomic actions (such as two processes starting) occurring simultaneously is zero. A third alternative is also based on a countable set, but one that contains countable subsets for each possible process that could occur simultaneously, an approach called branching time. This thesis bases the underlying countable sets of both linear and branching time on the natural numbers. Thus a program starts execution at time 0, and the shortest possible time duration for any event is arbitrarily set to 1. This implies that the first possible event after the program starts may not occur at time 1, although in most cases this assumption is valid.

The following definition formalizes the concept of temporal logic.

Definition A.6 *A modal logic with frame* (S, R), $s \in S$, $t \in S$, $u \in S$, *is called a* temporal logic *if and only if the binary relation* R *is*

1. *Reflexive:* $\forall s[(s, s) \in R]$

2. *Transitive:* $\forall s,t,u[((s,t) \in R \wedge (t,u) \in R) \Longrightarrow (s,u) \in R]$

3. *Antisymmetric:* $\forall s,t[((s,t) \in R \wedge (t,s) \in R) \Longrightarrow s = t]$

Thus Theorem A.5 states that the following schemata are valid for any temporal logic:

1. $\Box w \Longrightarrow w$
2. $\Box w \Longrightarrow \Diamond w$
3. $\Box\Box w \Longleftrightarrow \Box w$

The second item from the list follows from the fact that R being reflexive implies that R is also directed (see Theorem A.5), while the third follows from R being transitive and weakly dense. That any reflexive relation is also weakly dense is shown by

$$\forall s,t[(s,t) \in R \Longrightarrow (s,s) \in R \wedge (s,t) \in R]$$

Definition A.6 leads to the following result.

Theorem A.7 *Within any temporal logic the following schema is valid.*

$$(\Box p \Longrightarrow \Box q) \Longrightarrow (p \Longrightarrow q)$$

Proof: This follows directly from the valid schema

$$\Box w \Longrightarrow w$$

In the previous section the assertion that w is true meant that w evaluated to *true* for every interpretation s, $s \in S$, and with respect to a schema such as

$$w \Longrightarrow \Diamond w \tag{A.7}$$

the assumption that w was *true* meant that there existed some interpretation s such that w evaluated to *true* at s. Within this thesis, the use of modal operators within a schema

signifies that the logic is actually temporal logic, with the convention that the schema given by A.7 means that 'if w is *true* at the present (time 0), then eventually w is *true*'. Thus the s that w is taken to be *true* at is that which represents the present, which is typically $s = 0$.

Whether the schema of A.7 is valid or not within temporal logic addresses a subtle issue regarding the $\Diamond$ operator. Although

$$\models_s \Box w$$

can be *true* even if there is no $t \in S$ such that $(s,t) \in R$,

$$\models_s \Diamond w$$

can only be *true* if there exists a $t \in S$ such that $(s,t) \in R$ and

$$\models_t w.$$

Since Definition A.6 states that R is reflexive for a temporal logic, then the schema

$$w \Longrightarrow \Diamond w$$

is valid, since if

$$\models_s w$$

is *true*, then

$$\models_s \Diamond w$$

must also be *true* because (s,s) is always an element of R. This follows from the semantics of the $\Diamond$ under reflexive time to mean 'at the present or any future time', whereas if time were defined as irreflexive (not an unintuitive definition), then the $\Diamond$ would mean 'at any future time'.

The previous section presented the primitive modal operator $\Box$, along with the derived modal operators $\Diamond$ and $\rightsquigarrow$. The sytactical use of these operators did not require any reference to the frame of any particular modal logic, but conversely the semantics associated with these operators, which was not given in the last section, does require reference to the underlying frame. Consider first the primitive $\Box$ operator. With respect to a given frame (S, R) that satisfies Definition A.6, $\Box w$, where w is a wff, is valid if and only if

$$\forall s[s \in S \Longrightarrow \forall t[(t \in S \wedge (s,t) \in R) \Longrightarrow \models_t w]]$$

for all possible models based upon (S, R). This statement says that a wff of the form $\Box w$ is valid if and only if it's true for all possible models, where $\Box w$ is considered true if and only if for every interpretation s that makes w *true*, then if (s, t) is an element of R, w must be *true* at t. Given the above definition of a linear time, such that for s and t elements of S

$$(s,t) \in R \Longleftrightarrow s \leq t$$

the following semantics for the $\Box$ operator are a natural consequent. The wff

$$\Box w$$

is true if and only if the wff w is *true* at the present time and for all future times. For this reason this operator is called *henceforth*, so that $\Box w$ reads 'henceforth w'. Consequently the derived operator $\Diamond$ is called *eventually*, so that $\Diamond w$ reads 'eventually w', and the derived operator $\rightsquigarrow$ is called *leadsto*, with $w \rightsquigarrow v$ read as 'w leadsto v'. The name 'eventually' follows from the linear time semantics of the truth of

$$\Diamond w$$

meaning that 'it is not true that for all present and future times w is and will be *false*', or equivalently 'for some present or future time w is or will be *true*. Likewise, 'leadsto'

arises from the semantic meaning that the wff

$$w \rightsquigarrow v$$

is true if and only if henceforth if w is *true* then eventually v is *true*, which means that 'if at any time w becomes *true*, then at that or some future time v will become *true*'. Note that for all other times except those specifically addressed by these semantics the values of the wffs are unknown.

Another presentation of the meanings of the modal operators within linear time based frames uses explicit symbolic representations of the time variable. Given a predicate P that has a time variable as a parameter, denoted by $P(t)$, then with respect to the value of 0 for t, which represents the present, the following statements relate the modal operators to quantified wffs from the predicate logic:

$$\Box P \Longleftrightarrow \forall t[P(t)]$$

$$\Diamond P \Longleftrightarrow \exists t[P(t)]$$

Within linear time temporal logics there are additional operators which appear in the literature, refered to as temporal operators and typically applied to logics based on computational models (64, 319). The three presented here are the binary operators *until* $\mho$, *unless* $\mathcal{U}$, and the unary operator *next* $\bigcirc$. The $\mho$ operator is primitive, such that given the model (S, R, V), the interpretation s, $s \in S$, and the wffs v and w

$$\models_s (v \mho w) \Longleftrightarrow \exists t[(s,t) \in R \wedge \models_t w \wedge \forall u[((s,u) \in R \wedge (u,t) \in R) \Longrightarrow \models_u v]] \qquad (A.8)$$

From this follows the semantic definition that

$$v \mho w$$

read 'v until w', is true if and only if at some future time w will become *true*, and from the present time until that future time v remains continuously *true*, or else both v and w are true at the present time. At all other times the values of v and w are unknown.

The unless operator $\mathcal{U}$ is derived from the modal operators and the primitive operator $\mho$ by

$$(v \mathcal{U} w) \iff (\Box v \vee (v \mho w))$$

This implies that the semantic meaning of

$$v \mathcal{U} w$$

read as 'v unless w', is true if and only if either v is henceforth *true* or v is *true* until w is *true* (and w will eventually become *true*). Another equivalent meaning is that either v is henceforth *true*, or if not, then either at or before the time v becomes *false* w becomes *true*. At all other times the values of v and w are unknown. The UNITY based definition of the unless operator from Chandy and Misra (64) is slightly different, in that v is permitted to be continuously false, that is

$$v \mathcal{U} w \iff \Box \neg v \vee (\Box v \vee (v \mho w))$$

The definition of the unless operator used in this research agrees with the standard usage in the temporal logic literature (351, 267, 139).

It is possible to select a different set of primitive operators than those chosen here, although the end result, the syntax and semantics of the wffs, will be unchanged. One choice of primitive operators does appear to be minimal (fewest number) when compared to the others, and results from an observation by Sistla (319). It requires only one primitive operator, the $\mho$, with

$$\Diamond w \iff true \mho w$$

defining the $\Diamond$ and $\Box$ operators. The other operators would be derived in the same manner as presented here.

Within the temporal logics there are different subtypes, which are based on the different structures of the underlying frames (300). Within this thesis two such temporal logics are required, the *linear temporal logics* or *linear time temporal logics*, and the *branching temporal logics* or *branching time temporal logics*.

Definition A.8 *A temporal logic with frame* (S, R) *is called a* linear temporal logic *if and only if there exists an injective total function* ϕ

$$\phi : S \rightarrowtail \mathrm{N}$$

such that

$$\forall s, t[(s \in S \wedge t \in S) \Longrightarrow ((s,t) \in R \Longleftrightarrow \phi(s) \leq \phi(t))]$$

Although the injection ϕ is often the identity function, within this thesis ϕ is usually not the identity function (see Section 5.2 and 5.3). An immediate consequent of this definition is the following corollary.

Corollary A.9 *A temporal logic with frame (S,R), such that R is a linear order on S, is a linear temporal logic.*

Proof: A partial order is not sufficient to generate a linear temporal order, since it's possible to have two elements of S, say s and t, such that

$$\phi(s) \leq \phi(t)$$

but

$$\neg(sRt)$$

With a linear order, defining

$$\phi(s) \leq \phi(t) \Longleftrightarrow sRt$$

satisfies the requirement for a linear temporal logic, and does not define an inconsistent image of ϕ, since cycles (under R) are not possible. ■

With respect to this thesis, the other possibility for a temporal logic is given in the next definition.

Definition A.10 *A temporal logic with frame* (S, R) *is called a* **branching temporal logic** *if and only if there exists an injective total function* Φ

$$\Phi : S \rightarrow 2^{\mathrm{N} \times \mathrm{N}}$$

such that

$$\forall s, t[(s \in S \wedge t \in S \wedge (n, m) \in \Phi(s) \wedge (j, k) \in \Phi(t)) \Longrightarrow ((s, t) \in R \Longleftrightarrow (n = j \wedge m \leq k))]$$

where $n, m, j, k \in \mathrm{N}$.

That a branching temporal logic is also called a *branching time* temporal logic can be seen by considering that for an arbitrary s,

$$(n, m) \in \Phi(s)$$

yields an element (n, m) that is said to lie on the branch n. Thus all such elements that lie on any given branch can be linearly ordered by a relation that corresponds to the original R. If this linear order is denoted by $\hat{R}$, then

$$(n, m)\hat{R}(n, k) \Longleftrightarrow (m \leq k)$$

The function Φ has a range that consists of sets of ordered pairs because of the possibility that a given point s could lie on multiple branches. For an example see the text that accompanies Figure 5.2. Just as for linear temporal logics, this thesis typically uses branching temporal logics whose frames are *not* $(N \times N, R)$, where N is a subset of the natural numbers.

The primitive operator $\bigcirc$ does not have a standard definition of

$$\models_s \bigcirc w$$

with respect to arbitrary frames (S, R). However, the following definition satisfies the requirements of this thesis:

$$\models_s \bigcirc w \iff \exists j[s < j \wedge ((s < k \wedge k \leq j) \Longrightarrow j = k) \wedge \models_j w]$$

where w is an arbitrary wff. With a specific choice of a frame though, there exist such standard definitions for the $\bigcirc$ operator. Consider the frame chosen for the linear time logics of this thesis, the set of natural numbers along with the $\leq$ binary relation, that is $(\mathrm{N}, \leq)$. Given this frame and the interpretation $s \in S$,

$$\models_s \bigcirc w \iff \models_{s+1} w$$

defines the concept of truth associated with the next operator. Thus the idea of 'next w' means that w will be true at the very next instant of time, with its value unknown otherwise.

Note that one consequence of representing time with the natural numbers is this concise representation for $\bigcirc$. If instead a continuum had been chosen for time, then there would not have been such a representation for the $\bigcirc$ operator, since there is no known well ordering of such sets. And if instead a set dense in the continuum had been chosen, such as the rational numbers, the representation of the next time instant would not be based on the intuitive $\leq$ ordering that is associated with linear time.

The temporal operators defined to this point, with the exception of the next operator $\bigcirc$, address infinite intervals of time. For certain applications, such as the specification of real time systems (102), operators are needed that can quantify over finite intervals of time. One such formal logic is the temporal logic augmented with such additional operators, known as the *quantized temporal logic* (QTL) (288). Instead of using the exact same set of additional operators as QTL, this thesis adds a set of operators to the temporal operators such that all of the operators from QTL have equivalents in this additional set, plus the set used here is based on a more uniform symbology. These additional symbols do not add to the already defined semantics, they only assist in shortening the lengths of wffs. Although they are not traditionally considered modal operators (351), they will be called

modal operators within this thesis to simplify analytical statements. This set of additional modal operator along with their truth definitions is:

$$\models_s \Box_{<k} w \iff \forall j[(j \in S \wedge j \geq s \wedge j < k) \Longrightarrow \models_j w]$$

$$\models_s \Diamond_{\leq k} w \iff \exists j[j \in S \wedge j \geq s \wedge j \leq k \wedge \models_j w]$$

$$\models_s \Box_{\geq k} w \iff \forall j[(j \in S \wedge j \geq s \wedge j \geq k) \Longrightarrow \models_j w]$$

$$\models_s \Diamond_{>k} w \iff \exists j[j \in S \wedge j \geq s \wedge j > k \wedge \models_j w]$$

where $s \in S$, given the frame (S, R), and

$$(j,k) \in R \iff j \leq k$$

$$((j,k) \in R \wedge (k,j) \notin R) \iff j < k$$

defines the use of the '$\leq$' and the '$<$' symbols for an arbitrary frame. The meaning of

$$\Box_{<\hat{t}} P$$

is that the predicate P is true at the present and will remain true until the time $\hat{t}$ is reached, at which instant the value of P is unknown. Likewise,

$$\Diamond_{\leq \hat{t}} P$$

means that at some instant within the interval starting with now and ending at $\hat{t}$ the predicate P will be true, with its value unknown at all other times. So

$$\Box_{\geq \hat{t}} P$$

means that the predicate P will be true at the time instant $\hat{t}$ and continuously after that, with its value unknown until then; whereas

$$\Diamond_{>\hat{t}} P$$

means that the predicate P will be true at some instant after $\hat{t}$, with its value unknown at all other times.

The QTL wff

$$[k]w$$

means that w holds continuously in the closed time interval starting with the present and ending k time units after the present. This statement is equivalent to

$$\Box_{<(k+1)}w$$

Likewise, the QTL wff

$$\langle k\rangle w$$

which means that w must hold at least once in the closed interval starting with the present and ending k time units after the present, is equivalent to the wff

$$\Diamond_{\leq k}w$$

The QTL wff

$$v\mathcal{U}(k)w$$

which means that w must hold at least once in the closed invterval between the present and k time units from the present, and v holds continuously until them, is equivalent to

$$\exists j[j \leq k \wedge \Diamond_{\leq j}w \wedge \Box_{<j}v]$$

Finally, the QTL wff

$$\bigcirc(k)w$$

which means that w will hold in k time units from the present, is equivalent to

$$MARK$$

As an example of a wff using these shorthand notations, consider this alternate definition of the until operator $\mho$

$$v \mho w \Longleftrightarrow \exists t[\Box_{<t} v \wedge \Diamond_{<t} w]$$

where v and w are wffs, and $t \in \mathrm{N}$. If $\hat{t}$ is the instant that w first becomes *true*, then the t in this assertion is given by $t \geq \hat{t} + 1$.

The following summarizes the syntax, binding and associativity rules including the additional modal operators introduced in this section. Given the wff P, then

$$mP$$

is also a wff, where m is any of the unary modal operators $\bigcirc$, $\Diamond$, $\Box$, $\Diamond_{\leq t}$, $\Diamond_{>t}$, $\Box_{<t}$, or $\Box_{\geq t}$. If P and Q are wffs, then

$$PbQ$$

is also a wff, where b represents one of the binary operators $\rightsquigarrow$, $\mathcal{U}$, or $\mho$, and

$$PbQcR \Longleftrightarrow Pb(QcR)$$

where R is any wff, and c is any of these binary operators. This means that these binary operators are right associative, just as $\Longrightarrow$ is. Also,

$$mPbQ \Longleftrightarrow ((mp)bQ)$$

where P and Q are wffs, b is any binary operator, and m is any of the unary operators. Thus the unary operators bind tighter than any of the binary operators.

A consequence of Definition A.6 is that for any temporal logic

$$\Box\Box P \Longleftrightarrow \Box P$$

is a valid schema for any wff P. Since for any modal logic

$$\Diamond P \Longleftrightarrow \neg\Diamond\neg P$$

it follows that for any temporal logic

$$\Diamond\Diamond P \Longleftrightarrow \Diamond P$$

is a valid schema. Additionally, the following theorem covers the other possible combinations of more than two of the $\Box$ and $\Diamond$ operators.

Theorem A.11 *Given any temporal logic and wff P, the following are valid schemata:*

$\Box\Diamond\Box P \Longleftrightarrow \Diamond\Box P$

$\Diamond\Box\Diamond P \Longleftrightarrow \Box\Diamond P$

Proof: See the book by Hailpern (147).

These assertions imply that no more than two consecutive (non $\bigcirc$) unary modal operators are required within or preceding any wff. The assertion

$$\Diamond\Box P$$

means that eventually P will become true and then henceforth remain true. One example of such a wff is

$$\forall\epsilon[\epsilon \in \mathrm{R}^+ \Longrightarrow \Diamond\Box[d(x_t, l) \leq \epsilon]]$$

which states that l is the limit point of the sequence $\{x_t\}_{t\in\mathrm{N}}$ with respect to the metric d. Note that the predicate $d(x_t, l) \leq \epsilon$ can be represented as $P(t, l)$, where t is the time variable.

Since for a temporal logic whose frame is $(\mathrm{N}, \leq)$, $\bigcirc$ is similar to the successor operator, it's possible to have multiple consecutive $\bigcirc$ operators within or preceding a wff. For example, given the predicate P and a linear temporal logic whose frame is $(\mathrm{N}, \leq)$.

$s \in \mathrm{N}$,

$$\models_s \bigcirc\bigcirc P \Longleftrightarrow \models_{s+2} P$$

which shows that there is no (finite) limit to the number of consecutive $\bigcirc$ operators that can precede a wff.

The proofs of the assertions from Theorem A.11 also result in the following, which are valid for any temporal logic, and address the distribution of the unary (not the $\bigcirc$) modal operators over the binary operators $\wedge$ and $\vee$:

$\Box(P \wedge Q) \Longleftrightarrow (\Box P \wedge \Box Q)$

$\Diamond(P \vee Q) \Longleftrightarrow (\Diamond P \vee \Diamond Q)$

$\Diamond\Box(P \wedge Q) \Longleftrightarrow (\Diamond\Box P \wedge \Diamond\Box Q)$

$\Box\Diamond(P \vee Q) \Longleftrightarrow (\Box\Diamond P \vee \Box\Diamond Q)$

where P and Q are any wffs.

Instead of repeating the proof of one or more of the above statements, the following illustrates why the $\Diamond$ does not distribute over the $\wedge$, and why the $\Box$ does not distribute over the $\vee$. The other proofs all follow the same basic pattern. Consider the two wffs P and Q within a temporal logic whose frame is (S, R). If both

$$\Box_{<t1} P$$

and

$$\Box_{>t2} Q$$

are true, where

$$t2 > t1$$

then

$$\Diamond P \wedge \Diamond Q$$

holds, but

$$\Diamond(P \wedge Q)$$

does not hold. This means that

$$\neg(\Diamond(P \wedge Q) \Longleftrightarrow (\Diamond P \wedge \Diamond Q))$$

Likewise, if

$$t2 < t1$$

then

$$\Box(P \vee Q)$$

holds, whereas

$$\Box P \vee \Box Q$$

does not hold. This implies that

$$\neg(\Box(P \vee Q) \Longleftrightarrow (\Box P \vee \Box Q))$$

The following theorem addresses the distribution of the $\bigcirc$ operator over certain binary and unary operators within a temporal logic whose frame is $(\mathrm{N}, \leq)$.

Theorem A.12 *Given a linear temporal logic whose frame is* $(\mathrm{N}, \leq)$, *the following are valid for any wffs* P *and* Q*:*

$\bigcirc(P \vee Q) \Longleftrightarrow (\bigcirc P \vee \bigcirc Q)$

$\bigcirc(P \wedge Q) \Longleftrightarrow (\bigcirc P \wedge \bigcirc Q)$

$\bigcirc(P \Longrightarrow Q) \Longleftrightarrow (\bigcirc P \Longrightarrow \bigcirc Q)$

$\bigcirc\Diamond P \Longleftrightarrow \Diamond \bigcirc P$

$\bigcirc(P \mathcal{U} Q) \Longleftrightarrow (\bigcirc P \mathcal{U} \bigcirc Q)$

Proof: The proofs of the third and fifth schemata are given, the others follow the same pattern.

To prove the third schema, consider that

$$\models_s \bigcirc(P \Longrightarrow Q)$$

if and only if

$$\models_{s+1} (P \Longrightarrow Q)$$

if and only if

$$\models_{s+1} P \Longrightarrow \models_{s+1} Q$$

if and only if

$$\models_s \bigcirc P \Longrightarrow \models_s \bigcirc Q$$

if and only if

$$\models_s (\bigcirc P \Longrightarrow \bigcirc Q)$$

For the fifth item, consider that

$$\models_s \bigcirc(P \mathcal{U} Q)$$

if and only if

$$\models_{s+1} P \mathcal{U} Q$$

if and only if

$$\exists t[(s+1) \leq t \wedge \models_t Q \wedge \forall u[(s+1 \leq u \wedge u \leq t) \Longrightarrow \models_u P]]$$

if and only if

$$\exists (t-1)[s \leq t-1 \wedge \models_{(t-1)+1} Q \wedge \forall (u-1)[(s \leq u-1 \wedge u-1 \leq t-1) \Longrightarrow \models_{(u-1)+1} P]]$$

Since if $s+1 \leq t$ then $t-1 \geq 0$, and $s+1 \leq u$ implies that $u-1 \geq 0$. Thus the last assertion is true if and only if

$$\exists \hat{t}[s \leq \hat{t} \wedge \models_{\hat{t}+1} Q \wedge \forall \hat{u}[(s \leq \hat{u} \wedge \hat{u} \leq \hat{t}) \Longrightarrow \models_{\hat{u}+1} P]]$$

if and only if

$$\models_s (\bigcirc P \mathcal{U} \bigcirc Q)$$

To conclude this section consider the wff that asserts that the predicate P satisfies the induction principle:

$$\Box(P \Longrightarrow \bigcirc P) \Longrightarrow (P \Longrightarrow \Box P)$$

Given that P has arity one, then a linear time logic with frame $(\mathrm{N}, \leq)$ such that the variable is a natural number, results in the same meaning given to this wff as the traditional meaning based on an induction variable, which is also a natural number. But the underlying frame of the logic can be changed to $(\mathrm{N} \times \mathrm{N}, R)$ to yield a branching time temporal logic such that the meaning of this wff changes. For example, consider a countable set of processes running concurrently, where P represents the predicate given by $P(t) = \mathit{true}$ iff some process stops at time t. Then the induction principle states:

If is is henceforth true that whenever a process stops then another process stops in the next instant,

then it is true that if a process stops now there will henceforth be at least one process stopping at every instant.

If the number of processes is infinite, then this wff is true, since no finite time duration can cause all of the processes to terminate; but if instead the number is finite, then the induction principle would not be true, since once the last process stops there can be no more processes to stop. Thus the presence of a *boundary condition* (on the number of processes not yet stopped) causes this wff to be true until this boundary condition (no more processes to stop) is reached.

A.3 Predicate Logic

Instead of presenting the predicate logic separately, this thesis defines the predicate logic as a special case of the modal logic defined in Section A.1. Although other definitions of predicate logic are more standard (217, 225, 310), the one presented here is equivalent to the others and fits in better with the overall framework of this thesis. The only difference between the predicate logic of this section and the standard second order predicate logic is that this section omits the term operator

$$if < wff > then < term > else < term >$$

and the predicate (or wff) operator

$$IF < wff > THEN < wff > ELSE < wff >$$

since these two operators are not needed for this thesis. The standard usage of the term 'predicate' agrees with the definition given in Section 3.2 as a function whose codomain is the set $\{true, false\}$, while the term 'proposition' applies to a predicate of arity zero (296), that is a predicate whose evaluation does not depend upon any variables.

The basic idea of the predicate logic is that the wffs contain no modal operators (such as $\Box$ or $\rightsquigarrow$). Since the modal operators convey the idea that a wff can change its valuation by changing the interpretation, then predicate logic deals with wffs that have fixed interpretations and thus fixed values (once the variables are instantiated, see following paragraphs) of either *true*, *false*, or undefined. The following definition presents this concept more formally.

Definition A.13 *A modal logic with frame* (S, R) *is called a* **predicate logic** *if the set* S *contains a single element.*

Note that Definition A.13 is 'if' only, and not an 'if and only if' definition. This follows from the result that the standard definitions of the predicate logic do not necessarily imply the existence of a frame whose underlying set is a singleton. For example, if the frame's

set contained k elements, $k > 1$, but the relation R was defined such that

$$\forall s, t[(s,t) \in R \Longleftrightarrow s = t]$$

then the resulting logic would be a traditionally defined predicate logic whose wffs could be written without the modal operators, but would not have satisfied Definition A.13 if it had been worded 'if and only if'. As the definition stands, it does not preclude such a logic from being called a predicate logic.

Consider any predicate logic whose frame (S, R) contains the singleton set S, then the only two possibilities for the relation R are

$$(s,s) \in R$$

where s is the sole member of S, or

$$R = \emptyset$$

The first possibility, that R is reflexive, yields the following.

Theorem A.14 *Given a predicate logic with frame (S, R), such that S contains one element and R is reflexive, then for any wffs v and w:*

$\Box w \Longleftrightarrow w$

$\Diamond w \Longleftrightarrow w$

$v \rightsquigarrow w \Longleftrightarrow v \Longrightarrow w$

$v \mho w \Longleftrightarrow v \wedge w$

Proof: Since the proof of the second and third are based on the first, only the proof of the first and fourth is given.

The first follows directly from

$$\models_s w \Longrightarrow \forall s, t[(s,t) \in R \Longrightarrow \models_t w]$$

for $s = t$.

The fourth follows from setting both t and u (the only such u) equal to s in the definition for the $\mho$ operator (see Equation A.8).

Thus any wff from the modal logic whose frame satisfies Theorem A.14 can be written as an equivalent wff without the modal operators, i.e. written as a wff from the predicate logic. Intuitively, the removal of the modal operators can be justified based on a linear temporal logic interpretation. For example,

$$\Box w \Longleftrightarrow w$$

means that if w is and will always be *true*, then w is necessarily *true*. This conveys the idea that a wff does not change its valuation simply by 'waiting long enough'.

However, if the frame of the logic is (S, R), where S is a singleton set but R is not reflexive (i.e. R is empty), then the results from Theorem A.14 do not all hold. Specifically,

$$\Diamond w \Longleftrightarrow w$$

is not true, since even if

$$\models_s w$$

is *true*, there is no element of S, say t, such that

$$(s, t) \in R \wedge \models_t w.$$

Since this is counter-intuitive with the semantics associated with the $\Diamond$ operator under the temporal logic of linear time, the convention used in this thesis is that unless stated otherwise, any predicate logic based on a frame (S, R) with a singleton set S, has a binary relation R that is reflexive.

Given a predicate logic with frame (S, R), consider the concept of a *state*. Although the state can be assumed to be a primitive concept within the propositional modal logic (139), this thesis will define what a state is for both temporal and predicate logics. The

predicate logic definition is presented here, and applies to all frame based predicate logics within this thesis, while the definition for each temporal logic is given within the text as required. Note that each temporal logic can have a unique definition of a state.

Although a frame based predicate logic may consist of only one interpretation, there can exist multiple sets from whose elements values are selected for the variables (either individual, functional, or predicate) within the wffs of the logic. The idea of a state stems from the different possible instantiations for these variables, such that each state of the wff corresponds to one choice of a value for each variable.

Definition A.15 *Given a predicate logic with frame (S, R), and wff w, the* state *of w is a set of assignments, each assignment denoted by*

$$v := \alpha$$

where v denotes a variable from w, and α denotes an element of the set of possible values for v, such that there is exactly one assignment for each unique variable from w. If there are fewer assignments than the number of unique variables from w, then the set is called a partial state.

Definition A.15 is presented without explicitly defining what an assignment is, or how the determination is made as to what the unique variables are in any wff. The book by Revesz (301) or the one by Barendregt (26) present detailed explanations of these concepts, but for this thesis they will be assumed to be primitive concepts. Additionally, all assignments are assumed to be Turing computable (see Appendix B), which rules out the concept of *random assignments* (139).

As an example of the states of a wff, consider the predicate logic with frame (S, R), where S is a singleton set and R is reflexive, along with the wff

$$\forall x, y[p(x, y) \Longrightarrow \neg q(r, s, x, y)]$$

If the (single) interpretation is that p is a predicate such that $p(x, y)$ is *true* if $x > y$, and q is a predicate such that $q(r, s, x, y)$ is *true* if x and y are both the greatest common divisor

(gcd) of r and s, where r, s, x, and y are natural numbers, then this wff is *true*. A partial state would be

$$\{r := 20, s := 25\}$$

Now consider the program that implements Euclid's algorithm for finding the gcd of two natural numbers (46). The following is the sequence of states that result from each iteration of the algorithm (subtraction version) starting with the values of r and s given by the above partial state.

$$\{r := 15, s := 10, x := 5, y := 10\}, \{r := 15, s := 10, x := 5, y := 5\}$$

Once the values for the variables generate a *true* value for the predicates p and q, the algorithm stops. Although the wff is always *true*, regardless of the assignments, there is only one specific set of values for the predicates comprising the wff that correspond to successful program termination, which is typical of the wffs used in the analysis of the partial correctness of programs (126, 217, 64).

The important concept here is not the assignments themselves, but the idea that the iterative algorithm can be somehow described and monitored using a sequence of states pertaining to one or more wffs. Such states, resulting from a wff that describes or monitors a program, are called *program states*. Often the program states are listed alone, without the corresponding wff (217, 64).

Formulation	Statements	States
$\Box w$	$\forall p$	$\forall s_i \wedge$ initially
$\Diamond w$	$\exists p$[1]	$\exists s_i \vee$ initially
$\Box\Diamond w$	$\exists p$	$\exists s_i$
$\Diamond\Box w$	$\forall p$	Reachable Fixed Point

1 This requirement can be satisfied by the initially section.

Figure A.1. Temporal Specifications Versus UNITY Execution Sequences

A.4 Analogy Between Temporal Specifications and UNITY

This section presents a short overview of the analogy between certain specifications written in the syntax of the temporal logic and the design and structure of UNITY programs. The class of specifications considered are shown in the left hand column of Figure A.1, where the w denotes a static well formed formula (wff). The general issue of representing temporal specifications in terms of a state based machine is analyzed in a paper by Manna and Pnueli (222), but there does not appear to be a similar summary type analysis of the relationship between such specifications and UNITY, although the proof of program properties in the Chandy and Misra book (64) is very closely related. Since the UNITY execution model is based on infinite sequences of state transitions (see Section 4.4), such an analogy between UNITY and temporal formulas is a natural consequent. Figure A.1 summarizes the material presented in this section.

Consider a specification of the form

$$\Box w$$

where w denotes a static (no explicit time dependence) well formed formula (wff) from the first order predicate calculus (the free variables correspond to named variables in the program). To design a UNITY program that satisfies this specification requires that every statement satisfy the wff w independently of the execution sequence chosen, and that the initially section must also satisfy w. If these two requirements are met, then w will be *true* for every possible state of every possible execution sequence. This is depicted in Figure A.1 by the two entries under the 'Statements' and 'States' columns on the row whose entry

under the 'Formulation' heading is the $\Box w$. The

$$\forall p$$

entry under the 'Statements' column states that for every statement p in the program, w is *true* for some state of some sequence generated by the statement p. The possible sequences of states generated by a statement include those (finite) sequences that result from any possible initial state when the statement begins execution under either the UNITY model or the standard execution model (see Section 4.4). The

$$\forall s_i$$

portion of the entry under 'States' means that w is *true* for every state s_i of any sequence that can be generated by the statement p, while the

$$\wedge \text{initially}$$

states that the initially section results in an initial state for the program execution that also satisfies the wff w. A specification of this form implies that w is what Chandy and Misra term an *invariant* (64). If w is satisfied by the UNITY statement s, such that s can be written in the form of a proper set of equations (see Sections 2.5 and 2.7 of (64)), then the specification $\Box w$ is satisfied by the UNITY program

```
Program Satisfy Invariant
        always
                s
end
```

The next row in the figure corresponds to a temporal wff of the form

$$\Diamond w$$

Since this states that in any possible execution the wff w will eventually become *true*, then either w is *true* as a consequent of the initially section, or from some state generated by

at least one statement. The entry

$$\exists p$$

under 'Statements' indicates that w is *true* for at least one state in some sequence of states generated by a statement p, subject to the provision that if w is *true* for the state resulting from the initially section, then there is not necessarily such a statement p. The entry

$$\exists s_i \vee \text{initially}$$

under 'States' means that either some state in every possible state sequence generated by the statement p satisfies w, or (inclusive or, both could be true) the initially section generates an initial state for program execution that satisfies the wff w. A specification of this form implies that w describes what Chandy and Misra call a *progress property* of the program, although the intent of a progress property is for w to be satisfied by another state in the program execution sequence besides the initial state (64).

The next entry in the figure, corresponding to the wff

$$\Box\Diamond w$$

conveys the concept that for any unbounded sequence of states generated by the program, there is an unbounded number of these states that satisfy the wff w. This means that since both the UNITY and the standard execution model require that every statement execute an unbounded number of times in every unbounded execution sequence (sequence of states), then if there is at least one statement p such that some state for every possible execution sequence of p satisfies w, then $\Box\Diamond w$ is *true*. This figure expresses this requirement by stating that there must be at least one statement p, the

$$\exists p$$

entry under 'Statements', such that there is at least one state s_i for every possible execution sequence for p, the

$$\exists s_i$$

entry under 'States', such that s_i satisfies w. This figure emphasizes that the only difference between a specification of the form $\Diamond w$ and another of the form $\Box\Diamond w$, is that the first can be satisfied by having the initially satisfy w, whereas the second cannot (although the intially section may satisfy w). Manna and Pnueli call the set of states that satisfy w an unbounded number of times in the manner described for $\Box\Diamond w$ the *recurrent states* of the computation (222).

The last entry in the figure is for specifications formulated as

$$\Diamond\Box w$$

which state that for any unbounded execution sequence for the program, there are only a finite number of states that do not satisfy w, and after the last of these states is generated, all of the remaining unbounded number of states do satisfy w. This is the concept of a reachable fixed point, which is crucial to many specifications, and usually specifies program termination. The entry

$$\forall p$$

under 'Statements' denotes that w must satisfy at least one state in some sequence generated by every statement p of the program, while the entry under 'States' denotes that the requirement as to which states of which sequences satisfies w is based on the reachable fixed point requirement from Chandy and Misra's book (64). The set of states that satisfy w as given by such a reachable fixed point consists of what Manna and Pnueli call the *stable states* of the computation (222).

Appendix B. *Computability, Chaos, and Metric Spaces*

This appendix presents assorted theorems and analysis that support and augment the material in the previous chapters with respect to computability and chaos; it also presents a topological theorem to support the material from the introduction to Chapter VI regarding the relationship between the computable real numbers and the standard metric space of the real numbers with the absolute value metric.

Section 3.3 assumed that any countable set can be well ordered, an assumption which can be derived from the Axiom of Choice for sets (189). Another assumption from Section 3.3 is that the choice function from the choice axiom is in some sense 'more powerful' than any Turing computable function. The proof of this second assumption also follows from the Axiom of Choice, and is given in the following theorem along with a more formal statement of the assumption.

Theorem B.1 *The class of choice functions (from the Axiom of Choice), whose domains are a countable collection of countable sets, contains functions that are not Turing computable.*

Proof: (Note that the actual derivation of the set T from the set S below is not required to prove this theorem, but is presented to partially demonstrate the equivalence between the Axiom of Choice and the Well Ordering Principle. This derivation follows that of Kelley (189))

Consider an arbitrary countable set S. The Axiom of Choice implies that S can be well ordered using a choice function C, such that

$$C(S) = s_1$$
$$C(S - \{s_1\}) = s_2$$
$$\vdots$$
$$C(S - (\{s_1\} \cup \ldots \cup \{s_{n-1}\})) = s_n$$
$$\vdots$$

forming the well ordered set

$$T = \{s_1, s_2, ..., s_n, ...\}$$

If S is a countable set that is not Turing enumerable (type 0), such as the set of all Turing computable numbers, then C is not a Turing computable function, since no Turing computable function can produce the set T. ■

Section 4.2 uses the concept of computability and computable numbers. Within this thesis computability refers to any finite algorithm that can be implemented on a Turing machine. Accordingly, computable numbers are defined in terms of Turing machines, based on the original concept of a computable number expressed by Turing. Turing's definition was in some sense a recursive one. The base case was that the computable numbers include all those numbers whose digits can be placed onto a Turing machine's tape in finite time. These numbers form a set with respect to the reals that would be all those real numbers with finite decimal expansions. An interesting property of this set is that it is dense in the reals, and is a proper subset of the rational numbers. The recursive part of the definition was that the computable numbers include all of the limit points of convergent computable sequences of computable numbers. The original definition by Turing was that: (334)

> We shall say that a sequence β_n of computable numbers converges computably if there is a computable integral valued function $N(\epsilon)$ of the computable variable ϵ, such that we can show that, if $\epsilon > 0$ and $n > N(\epsilon)$ and $m > N(\epsilon)$, then $|\beta_n - \beta_m| < \epsilon$.

Thus the computable numbers include the limit points of Cauchy sequences of computable numbers, such that the sequence is computable. Turing concludes that the computable numbers include all of the real algebraic numbers, along with those transcendental numbers that are defined in terms of computable series, such as e and π.

A more intuitive notion as to exactly what are computable numbers is contained in the following definition from Minsky: (246)

> The digits must be generated sequentially by a Turing machine. That is, we require that in order that the real number $.a_0a_1a_2...$ be a computable real number, there must be a Turing machine which starts with a blank tape and prints out a tape of the form
>
> $$...000xa_0xa_1xa_2...000$$
>
> We make the rule that, once an x is printed, the machine must never move to the left of, or change, that x.

Minsky concludes the definition by saying "the printing of an x is an irrevocable announcement that a digit has been computed and is in the square to the left of the x." This concept can be summarized by saying that a computable number is one for which there exists a Turing machine, such that for any natural number n, there exists some other natural number k, so that the first n digits of the number will be placed on the Turing machine's tape within the first k moves of the machine, and k can be computed in finite time by a Turing machine whose input is n.

In most cases, computable numbers input into computable functions yield computable answers, but there are exceptions. One of these is based on the proven unsolvability of the halting problem for Turing machines (217). Consider the function f, such that for any Turing machine t that corresponds to a function of a single natural number, and natural number n, the evaluation of $f(t,n)$ is a real number between 0 and 1. Further, $f(t,n)$ consists of a decimal point, and then a string of decimal digits formed in the following manner:

If, given the input n, t halts on step k of its operation, then digit number k is a 4, and all successive digits are 0.

If, given the input n, t does not halt on step k of its operation, then digit number k is a 3.

Next consider the function g that has the same domain and codomain as does f, such that $g(t,n)$ is defined as follows:

If, given the input n, t halts on step k of its operation, then digit number k is a 6, and all successive digits are 0.

If, given the input n, t does not halt on step k of its operation, then digit number k is a 6.

That $f(t,n)$ and $g(t,n)$ are computable numbers follows from the definition given by Minsky above. For any required number k of decimal digits of either number, only k steps of the Turing machine t are required. So the function h, whose domain is ordered pairs of computable numbers, is defined for

$$h(f(t,n),g(t,n))$$

for any applicable Turing machine t and natural number n. But if h is the addition function, then there exist certain t and n for which the evaluation of h cannot be computed, even though addition is a computable function.

The converse concept to this last example would be if there exist computable functions such that given uncomputable inputs, the evaluation would still be computable. The following theorem shows that such a case can occur.

Theorem B.2 *The noncomputability of the (instantiation of the) variable(s) of a computable function does* not *strictly imply that the function evaluation is noncomputable.*

Proof: Consider the function

$$\sigma : \Sigma^* \times \Sigma^* \rightarrow \mathrm{R}^+ \cup \{0\}$$

defined by Equation 4.3. If both x and y are noncomputable elements of $\{0,1\}^*$ such that

$$x = 0\cdots$$

$$y = 1\cdots$$

then

$$\sigma(x,y) = 1$$

■

In his book on CSP (165), Hoare defines a chaotic process as one which is totally nondeterministic, that is one which can perform any type of behavior. To demonstrate this concept, Hoare defines the process CHAOS as a process that is capable of behaving as any other process, that is for any fixed alphabet Σ,

$$CHAOS = \mu X.X$$

which implies that

$$traces(CHAOS) = \Sigma^*$$

UNITY's weak fair choice operator on statements implies that there is no UNITY program that is equivalent to CHAOS. Even though UNITY alone cannot duplicate CHAOS, it is possible in a sense to have a UNITY simulation of CHAOS, that is a UNITY program that uses named variables to replicate the behavior of the CHAOS process. Based on Chandy and Misra's (64) observation that fair choice can be used to construct an 'unfair' choice operator, the following UNITY program Chaos generates values for a variable denoted 'trace' that are equivalent to the possible traces of the process CHAOS. (Think of the variable 'trace' as storing the trace of the execution of the process CHAOS) The operator ';' represents trace concatenation, and Λ denotes the empty trace.

Program Chaos

initially

trace := Λ

assign

trace := trace;0 if b $\sim$ trace;1 if $\neg$b

[] b := $\neg$b

end

The 'b' denotes a boolean valued variable.

The following theorem supports the claim made in the introduction to Chapter VI that the computable numbers, which are dense in the real numbers but do not include all of the real numbers, cannot be complete with respect to the standard metric ($|\ |$) for the reals.

Theorem B.3 *Given the complete metric space (X,d), and the set Y, such that Y is a subset of X but is not equal to X, and Y is dense in X with respect to d, then (Y,d) is not complete.*

Proof: Since $X \neq Y$, then there exists an element of X, say x, that is not an element of Y. Consider a Cauchy sequence in Y whose limit point is x. Such a Cauchy sequence exists because Y is dense in X. But the limit point of this Cauchy sequence is not in Y, thus Y is not complete. ■

Appendix C. *Analysis of UNITY Program Implementations*

This appendix presents a well known theorem from program testing theory and demonstrates its application to the probabilistic analysis of actual implementations of UNITY programs. Although the theory (in the form of the theorem) is not new, the application to implementations of UNITY programs is new material. Additionally, a key mathematical result used in the proof of the theorem, which is not found in the common computer science references, is proved instead of simply stated. This appendix concludes with a proof that the execution of a UNITY program is equivalent to the execution of a sequence of first order predicate logic formulas.

The following theorem is Theorem 4.3 from Howden (171), with only minor changes in the wording that do not affect the content.

Theorem C.1 *Suppose that f is a function whose input is selected according to an operational input distribution and let F_f be a set of functions containing f and other functions with the same operational distribution as f. Suppose that f' is some function in F_f and we wish to test for the probable equivalence of f and f'. This can be done as follows. Let θ be the unknown probability that $f \neq f'$ when tested over an input element which is selected according to the operational distribution. Choose h to be such that if $\theta < 1/h$ we can think of f and f' as being probably equivalent. Now consider the following hypothesis H.*

$$H : \theta < 1/h$$

Suppose we test f and f' over n tests and do the following. We accept the hypothesis if $f = f'$ over all tests and reject it otherwise. The type 2 error (the error of accepting the hypothesis when it is false) for this hypothesis is more serious than the type 1 error (the error of rejecting the hypothesis when it is true). In the type 2 error we accept that $\theta < 1/h$ when it is not true, that is, we accept that f is probably equal to f' when it is not true. In the type 1 error we reject that $\theta < 1/h$ when in fact it is true. But rejecting the hypothesis only happens if f was found to be unequal to f' on some test, and if this happens we are no

longer interested in θ anyway, f is known to be unequal to f'. If

$$n = h\ln(h)$$

then the probability of making a type 2 error is less than 1/h, that is, if f and f' agree on $h\ln(h)$ tests then they are probably equal, to within probability 1/h.

Proof: Most of the proof is contained in (171), what follows is that part that is not, nor is it contained in the original cite given by Howden (335).

Starting with the requirement (proved in (171))

$$n > \frac{\ln(h)}{\ln(h) - \ln(h-1)}$$

to ensure that the probability of type 2 error is less than 1/h, the first step is to prove that

$$(1 - 1/k)^k < 1/e \qquad k \in \{1, 2, ...\}$$

The inequality holds for $k = 1$, while for $k > 2$ only the limiting case as $k \to \infty$ needs be considered, since the function

$$(1 - 1/k)^k$$

is strictly increasing, and the right side of the inequality is a constant. By comparing the binomial expansion (found in the CRC (41))

$$(1 - 1/k)^k = 1 - k/k + k(k-1)(1/k)^2/2! - k(k-1)(k-2)(1/k)^3/3! + ... \quad \text{(C.1)}$$

with the series expansion for e (found in the CRC)

$$e^{-1} = 1 - 1 + 1/2! - 1/3! + ... \quad \text{(C.2)}$$

the inequality follows from the observation that each numerator in Equation C.1 is less than the corresponding numerator in Equation C.2, after the $1 - k/k$ and $1 - 1$

are deleted.

Given that

$$(1 - 1/k)^k < e^{-1}$$

then it follows that

$$k \ln(1 - 1/k) < -1$$

which implies that

$$k \ln(1 - 1/k)^{-1} > 1$$

and so

$$k > \frac{1}{\ln(k/(k-1))}$$

This last inequality implies that

$$k > \frac{1}{\ln(k) - ln(k-1)}$$

which yields

$$k \ln(k) > \frac{\ln(k)}{\ln(k) - \ln(k-1)}$$

Substituting h for k in this last inequality proves the theorem. ■

Although this theorem is intended for analysis of the equivalence of two programs, it can also be applied to the analysis of an implementation of a UNITY program on a finite machine. This analysis intends to answer the question of whether a given fixed point that is always reachable for any execution sequence of the UNITY program can always be expected to be reached in the implementation if it's always reached during testing (of the implementation).

Consider the mapping of a UNITY program onto a machine that satisfies the following assumptions:

Random Execution Selection Given the set of possible finite execution sequences of the statements in the UNITY program, either in the standard UNITY execution model or the shared variable execution model (see Appendix D), each execution of

the implementation represents a random choice from this set, such that each choice is independent of the other choices.

Equi-probable Execution Subset Given the set of possible finite execution sequences of the statements in the UNITY program, either in the standard UNITY execution model or the shared variable execution model, there exists a subset of this set such that each sequence in this subset has equal (nonzero) probability of occurring as an execution sequence of the implementation.

These are reasonable assumptions for a machine such as the Intel Hypercube (34). Each execution of the Hypercube is independent of any previous executions, and also any subsequent. Additionally, within the set of reasonably expected possible interleavings of the individual node program statements, the probability of any one interleaving actually occurring can be considered equal to that for any other interleaving. Certainly there are execution sequences for the UNITY program that are never expected to occur on the Hypercube, and those sequences are excluded from the equi-probable execution subset.

The concept behind this next theorem is that for a UNITY program with a reachable fixed point, the implementation may be designed to also reach that fixed point in every possible execution (34). But short of some type of formal proof that this is true, the alternative must be testing. An important question is how much testing is enough, and also, once testing is complete, what can be actually stated.

Theorem C.2 *Given a UNITY program P with fixed point, and an implementation of P on a finite machine, if this implementation satisfies the Random Execution Selection and Equi-probable Execution Subset assumptions, then the following methodology can be used for testing and reporting on the hypothesis*

H: Every execution of the implementation reaches the fixed point.

If, for some positive integer k,

$$k \ln(k)$$

tests of the implementation, using the same operational input distribution as the UNITY program, each attain the fixed point, then it can be stated that all executions of the imple-

mentation attain the fixed point with the probability of type 2 error less than

$$1/k$$

Proof: Denote the implementation as a program f whose possible execution behavior is governed by the Random Execution Selection and Equi-probable Execution Subset assumptions. Consider the program F, as the desired implementation that behaves just as the UNITY program constrained to these two assumptions, that is the UNITY program whose set of possible executions is restricted to only those from the equi-probable execution subset. This restriction does not affect the fixed point property of the UNITY program, since fixed points are independent of the actual execution sequence (64). Both f and F are elements of the set of all possible implementations of the UNITY program that satisfy the two assumptions. This theorem then follows from Theorem C.1. ■

The execution model for UNITY requires that given a unique state immediately prior to the execution of an assignment statement, then there is another unique state immediately after the statement executes. The primary constraint on UNITY programs that ensures this requirement is satisfied is that the domains for the individual variables must be finite. This leads to the result that the execution of any UNITY program (with the assumption that there are no duplicated statements) is equivalent to the execution of an arbitrary sequence of statements from the first order predicate logic.

Theorem C.3 *Given a UNITY program with unique assignment statements, then for each assignment statement under the UNITY execution model, there exists a wff from the first order predicate logic whose individual variables are the named variables from the assignment, such that the wff is true, if and only if, the assignment statement is executable.*

Proof: Figure C.1 shows the syntax for any given assignment statement from a UNITY program (figure is from "Predicate and Temporal Logic", Gary B. Lamont and Jeffrey Simmers. Dept of Electrical and Computer Engineering, Air Force Institute of Technology, 1991). Starting with the simplest assignment statement

variable := expr

this can be represented as

$$x_{n+1} = f(x_n)$$

where the sequence

$$\{x_i\}_{i \in \mathrm{N}}$$

denotes the sequence of state vectors resulting from a given execution of the UNITY program, each state vector composed of the named program variables. The function f represents the assignment performed by the expr. If the assignment is a conditional of the form

variable := expr1 if boolean-expr1 $\sim$ expr2 if boolean-expr2

then the corresponding wff is

$$(P(x_n) \Longrightarrow x_{n+1} = f(x_n)) \wedge (Q(x_n) \Longrightarrow x_{n+1} = g(x_n)) \wedge [(\neg P(x_n) \wedge \neg Q(x_n)) \Longrightarrow x_{n+1} = x_n]$$

where P, f, Q, and g correspond to boolean-expr1, expr1, boolean-expr2, and expr2, respectively. The and operator $\wedge$ results from the requirement that if both boolean expressions are *true*, then both f and g must represent the same assignment, while if none of of the boolean expressions are *true*, then the value of x_n is not changed. If the assignment is a quantified assignment such as

$\langle\!\| \; i \; : 0 \leq i \leq N \; :: \;$ variable := expr $\rangle$

then the corresponding wff is

$$i \in \{0, ..., N\} \Longrightarrow x_{n+1} = f(x_n)$$

where f denotes the assignment performed by expr. In the more general case of

$\langle\!\|$ *quantification assignment-statement*$\rangle$

then the wff becomes

$$(P(w) \Longrightarrow \langle wff \rangle) \qquad \text{(C.3)}$$

where w is a vector variable denoting any individual variables named in the *quantification*. P is a predicate that evaluates to *true* if the *quantification* is satisfied, and

$\langle wff \rangle$ represents the wff that corresponds to *assignment-statement.* If $P(w)$ is not *true* for any w, then UNITY requires that the quantified assignment be an empty statement, so that there is no additional symbology needed in the wff of Equation C.3.

As shown in Figure C.1, any assignment component can be composed from these pieces. Given two assignment components within the same assignment statement, the the wff that corresponds to

component1 || component2

is

$$\langle wff1 \rangle \wedge \langle wff2 \rangle$$

such that $\langle wff1 \rangle$ represents the wff for component1, and $\langle wff2 \rangle$ represents the wff for component2. The and operator $\wedge$ results from the simultaneous execution of the two components.

Thus the wff corresponding to any assignment statement can be inductively constructed. ■

assign-section ::= statement-list
statement-list ::= assignment-statement { [] *assignment-statement*}
assignment-statement ::= assignment-statement | *quantified-statement-list*
quantified-statement-list ::= ⟨ [] *quantification statement-list* ⟩
assignment-statement ::= assignment-component
{ || *assignment-component*}
assignment-component ::= enumerated-assignment | *quantified-assignment*
enumerated-assignment ::= variable-list := expr-list
variable-list ::= variable {,*variable*}
expr-list ::= simple-expr-list | *conditional-expr-list*
simple-expr-list ::= expr {,*expr*}
conditional-expr-list ::= simple-expr-list if boolean-expr
{∼ *simple-expr-list if boolean-expr*}
expr ::= ⟨ *op quantification expr* ⟩
expr ::= basic-expr
quantified-assignment ::= ⟨ || *quantification assignment-statement* ⟩
quantification ::= variable-list : boolean-expr ::
op ::= min | *max* | + | × | ∧ | ∨ | ≡

Figure C.1. Syntax for *assign-section*

Appendix D. *Representation of UNITY Using Petri Nets*

This appendix presents a generalized Petri net (278, 279); that represents the execution model for UNITY. Since UNITY represents a formal specification language that is statement based, this appendix bridges the gap between UNITY and another formal specification language that is graphical, Petri nets.

There are two key properties of Petri nets as relate to the UNITY execution model. The first is that no two transitions can 'fire' simultaneously, so that *sequences* of states are generated, and no two assignments can simultaneously access the same nonlocal variable (see Definition IV.67). The second is that whenever there is more than one arc leading from a place, the weak fair choice principle applies, that is for any unbounded number of choices made as to which arc will 'fire', each arc is chosen an unbounded number of times, and each arc must only wait a finite number of choices before being chosen.

Given the UNITY program P,

```
Program P
    assign
        a || b
        [] c || d
    end
```

where a, b, c, and d denote assignments, then Figure D.1 depicts the Petri net that generates the same sequences of states as the execution model for the UNITY program P. The transitions represent an assignment execution, and the multiple arcs emanating from the place labeled 'w' represent the weak fair choice execution (see Section 4.4. Thus the places represent instantiations of the named variables, that is they denote the program states, and a sequence of states corresponds to a sequence of places. The transition labeled 'a' depicts that the firing of this transition generates the same state as the execution of the assignment a in the program P. This same analogy applies to the transitions labeled with the other assignments of P.

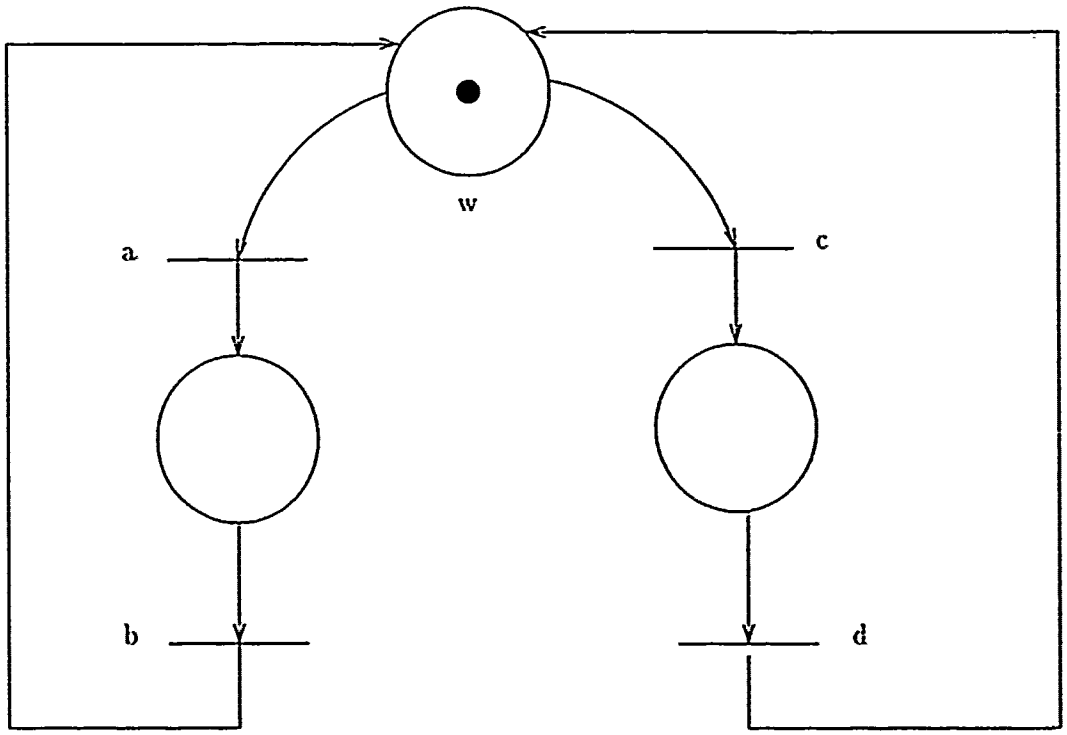

Figure D.1. Petri Net Model for UNITY Program P

Bibliography

1. M. Abadi, J. Feigenbaum, and J. Kilian. On Hiding Information From an Oracle. In *Proceedings of the 19th Annual ACM Symposium on the Theory of Computing*, pages 195–203, New York, 1987. ACM.
2. K. Abrahamson. Modal Logic of Concurrent Nondeterministic Programs. In *Semantics of Concurrent Computation, Lecture Notes in Computer Science 70*, pages 21–33. Springer-Verlag, New York, 1979.
3. J. R. Abrial and S. A. Schuman. Non-Deterministic System Specification. In *Semantics of Concurrent Computation, Lecture Notes in Computer Science 70*, pages 34–50. Springer-Verlag, New York, 1979.
4. G. A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems* The MIT Press, Cambridge, MA, 1986.
5. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1985.
6. P. America and J. Rutten. Solving Reflexive Domain Equations in a Category of Complete Metric Spaces. *Journal of Computer and System Sciences*, 39(3):343–375, December 1989.
7. P. America, J. Rutten, J. de Bakker, and J. N. Kok. Denotational Semantics of a Parallel Object-Oriented Language. *Information and Computation*, 83:152–205, 1989.
8. T. M. Apostol. *Mathematical Analysis.* Addison-Wesley Publishing Co., Reading, MA, 1974.
9. K. R. Apt. Correctness Proofs of Distributed Termination Algorithms. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 147–168. Springer-Verlag, Berlin, 1985. Volume 13, Series F of the NATO ASI Series.
10. K. R. Apt and C. Delporte. An Axiomatization of th Intermittent Assertion Method Using Temporal Logic. In *Automata, Languages and Programming 10th Colloquium*, pages 15–27. Springer-Verlag. Berlin, 1983. Lecture Notes in Computer Science 154.
11. K. R. Apt, N. Francez, and W. F. de Roever. A Proof System for Communicating Sequential Processes. *ACM Transactions on Programming Languages and Systems*, 2(3):359–385, July 1980.
12. A. Arnold. Topological Characterizations of Infinite Behaviours of Transition Systems. In *Automata, Languages and Programming 10th Colloquium*, pages 28–38. Springer-Verlag, Berlin, 1983. Lecture Notes in Computer Science 154.
13. Arvind and K. A. Iannucci. Two Fundemental Issues in Multiprocessing. In R. Dierstein et. al., editor, *Parallel Computing in Science and Engineering, Lecture Notes in Computer Science 295*, pages 61–88. Springer-Verlag, New York, 1987.

14. Arvind and R. S. Nikhil. A Dataflow Approach to General-purpose Parallel Computing. Computation Structures Group Memo 302, MIT Laboratory for Computer Science, July 1989.

15. R. G. Babb, editor. *Programming Parallel Processors.* Addison-Wesley, Reading, MA, 1988.

16. R. J. Back and R. Kurki-Suonio. Distributed Cooperation with Action Systems. *ACM Transactions on Programming Languages and Systems*, 10(4):513-554, October 1988.

17. R. J. R. Back. *Correctness Preserving Program Refinements: Proof Theory and Applications.* Mathematisch Centrum, Amsterdam, 1980.

18. R. C Backhouse. *Program Construction and Verification.* Prentice Hall International, Englewood Cliffs, NJ, 1986.

19. R. C. Backhouse. Constructive Type Theory: A Perspective from Computing Science. In E. W. Dijkstra, editor, *Formal Development of Programs and Proofs*, pages 1–32. Addison-Wesley, Reading, MA, 1990.

20. J. Backus. Can Programming Be Liberated From The von Neumann Style? A Functional Style And Its Algebra Of Programs. *Communications of the ACM*, 12(8):613-641, August 1978.

21. P. D. Bailor. *A Theory for Graph-Based Language Specification, Analysis, and Mapping with Application to the Development of Parallel Software.* PhD thesis, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, OH, 1989.

22. H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3):261-322, September 1989.

23. R. Balzer. Transformational Implementation: An Example. *IEEE Transactions on Software Engineering*, 7(1):3–13, January 1981.

24. R. Balzer, T. E. Cheatham, and C. Green. Software Technology in the 1990's: Using a New Paradigm. *IEEE Computer*, 16(11):16–22, November 1983.

25. R. Balzer, N. Goldman, and D. Wile. On the Transformational Implementation Approach to Programming. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 337–344, Washington, DC, 1976. IEEE Computer Society Press.

26. H. P. Barendregt. *The Lambda Calculus - Its Syntax and Semantics.* North-Holland, New York, 1984.

27. M. Barr and C. Wells. *Category Theory for Computer Science.* Prentice Hall, New York, 1990.

28. W. A. Barrett, R. M. Bates, D. A. Gustafson, and J. D. Couch. *Compiler Construction Theory and Practice.* Scientific Research Associates, Chicago, 1986.

29. H. Barringer. *A Survey of Verification Techniques for Parallel Programs.* Springer-Verlag, Berlin, 1987. Lecture Notes in Computer Science 191.

30. H. Barringer, R. Kuiper, and A. Pnueli. Now You May Compose Temporal Logic Specifications. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, pages 51–63, New York, 1984. ACM Press. Held at Washington DC on April 30-May 2, 1984. Sponsored by SIGACT.

31. F. L. Bauer et al. Techniques for Program Development. In *Software Engineering Techniques*, pages 25–50. Infotech International, Maidenhead, England, 1977.

32. F. L. Bauer et al. Programming in a Wide Spectrum Language: A Collection of Examples. *Science of Computer Programming*, 1:73–114, 1981. Published by Elsevier Science (North-Holland).

33. R. A. Beard, J. M. Sartor, G. A. Sawyer, and G. B. Lamont. Compendium of Parallel Programs for the Intel iPSC Computers. Technical Report Volume II Version 1.5 Research, Dept of Electrical and Computer Engineering, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, OH, 1990.

34. R. A. Beard, J. M. Sartor, G. A. Sawyer, and G. B. Lamont. Compendium of Parallel Programs for the Intel iPSC Computers. Technical Report Volume I Version 1.5 Research, Dept of Electrical and Computer Engineering, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, OH, 1990.

35. F. Belik. Deadlock Avoidance. *IEEE Transactions on Computers*, 39(7):882–888, July 1990.

36. J. A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification.* ACM Press, New York, 1989.

37. P. Bernays. A System of Axiomatic Set Theory - Part I. *The Journal of Symbolic Logic*, 2(2):65–77, June 1937.

38. A. Bernstein and P. K. Harter. Proving Real-Time Properties of Programs With Temporal Logic. In *ACM SIGOPS 8th Annual ACM Symposium on Operating Systems Principles*, pages 1–11, December 1981.

39. D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation.* Prentice-Hall In..., Englewood Cliffs, NJ, 1989.

40. A. T. Berztiss and M. A. Ardis. Formal Verification of Programs. Technical Report SEI-CM-20-1.0, Software Engineering Institute, Carnegie Mellon University, 1988.

41. W. H. Beyer, editor. *CRC Standard Mathematical Tables.* CRC Press, Boca Raton, FL, 1984.

42. N. P. Bhatia and O. Hajek. *Local Semi-Dynamical Systems.* Springer-Verlag, Berlin, 1969.

43. S. L. Bloom. All Solutions of a System of Recursion Equations in Infinite Trees and Other Contraction Theories. *Journal of Computer and System Sciences*, 27(2):225 255, October 1983.

44. L. G. Bouma and H. R. Walters. Implementing Algebraic Specifications. In J. A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, chapter 5. ACM Press, New York, 1989.

45. R. S. Boyer and J. S. Moore. *A Computational Logic.* Academic Press, New York, 1979.

46. G. Brassard and P. Bratley. *Algorithmics Theory and Practice.* Prentice-Hall, Englewood Cliffs, NJ, 1988.

47. S. D. Brookes. On The Relationship of CCS and CSP. In *Automata, Languages and Programming 10th Colloquium*, pages 83–96. Springer-Verlag, Berlin, 1983. Lecture Notes in Computer Science 154.

48. S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A Theory of Communicating Sequential Processes. *Journal of the ACM*, 31(3):560–599, July 1984.

49. M. Broy and B. Krieg-Bruckner. Derivation of Invariant Assertions During Program Development by Transformation. *ACM Transactions on Programming Languages and Systems*, 2(3):321–337, July 1980.

50. M. Broy and C. Langauer. On Denotational Versus Predicative Semantics. *Journal of Computer and System Sciences*, 42(1):1–29, February 1991.

51. M. Broy and P. Pepper. Program Development as a Formal Activity. *IEEE Transactions on Software Engineering*, 7(1):14–22, January 1981.

52. B. Bruegge. Program Development for a Systolic Array. *SIGPLAN Notices*, 23(9):31–41, September 1988. This issue was the Proceedings of the ACM/SIGPLAN PPEALS 1988.

53. R. M. Burstall. Proving Properties of Programs by Structural Induction. *The Computer Journal*, 12(1):41–47, February 1969.

54. R. M. Burstall. Program Proving as Hand Simulation With a Little Induction. In *Information Processing, Proceedings of IFIP 74*, pages 308–312. North-Holland, Amsterdam, 1974.

55. R. M. Burstall. Electronic Category Theory. In P. Dembinski, editor, *Mathematical Foundations of Computer Science 1980.* Springer-Verlag, Berlin, 1980. Lecture Notes in Computer Science 88.

56. R. M. Burstall and J. Darlington. Some Transformations for Developing Recursive Programs. In *Proceedings International Conference on Reliable Software*, pages 465 472, Los Angeles, CA, 1975. IEEE.

57. R. M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, January 1977.

58. R. M. Burstall and J. A. Goguen. Algebras, Theories and Freeness: An Introduction for Computer Scientists. In M. Broy and G. Schmidt, editors, *Theoretical Foundations of Programming Methodology*, pages 329–350. D. Reidel, Dordrecht, Holland, 1982. Volume 91, Series C of the NATO ASI Series.

59. R. M. Burstall and J. Thatcher. The Algebraic Theory of Recrusive Program Schemes. In *Proceedings. Symposium on Category Theory Applied to Computation and Control*, pages 126–131. Springer-Verlag, Berlin, 1975. Lecture Notes in Computer Science 25.

60. N. Carriero and D. Gelernter. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, 21(3):323–358, September 1989.

61. L. Carroll. *Symbolic Logic.* Clarkson N. Potter, Inc., New York, 1977.

62. K. M. Chandy and J. Misra. Proofs of Networks of Processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, July 1981.

63. K. M. Chandy and J. Misra. A Paradigm for Detecting Quiescent Properties in Distributed Computations. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 325–341. Springer-Verlag, Berlin, 1985.

64. K. M. Chandy and J. Misra. *Parallel Program Design.* Addison-Wesley, Reading, MA, 1988.

65. M. Chandy and J. Misra. An Example of Stepwise Refinement of Distributed Programs: Quiescent Detection. *ACM Transactions on Programming Languages and Systems*, 8,(3):326–343, July 1986.

66. C-L. Chang and R. C-T. Lee. *Symbolic Logic and Mechanical Theorem Proving.* Academic Press, New York, 1973.

67. A. C. Chen and C l. Wu. A Parallel Execution Model of Logic Programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):79–92, January 1991.

68. M. Chen, Y. Choo, and J. Li. Crystal: From Functional Description to Efficient Parallel Code. In *Volume I of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 417–433. ACM Press, 1988.

69. N. Chomsky. Three Models for the Description of Language. *I.R.E. Transactions on Information Theory*, IT-2(3):113–124, September 1956.

70. N. Chomsky. On Certain Formal Properties of Grammars. *Information and Control*, 2:137–167, 1959.

71. N. Chomsky and M. P. Schutzenberger. The Algebraic Theory of Context-Free Languages. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 118–161. North-Holland, Amsterdam, 1963.

72. N. Christofides. *Graph Theory An Algorithmic Approach.* Academic Press, New York, 1975.

73. A. Church. *Introduction to Mathematical Logic.* Princeton University Press, Princeton, NJ, 1956.

74. E. M. Clarke, M. C. Browne, E. A. Emerson, and A. P. Sistla. Using Temporal Logic for Automatic Verification of Finite State Systems. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 3–26. Springer-Verlag, Berlin, 1985. Volume 13, Series F of the NATO ASI Series.

75. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite State Concurrent Systems Using Temporal Logic. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

76. W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, Berlin, 1981.

77. J. R. B. Cockett. Distributive Theories. School of Mathematics and Computing, Macquarie University, Australia N. S. W. 2109, September 1989.

78. N. H. Cohen. *Ada As A Second Language.* McGraw-Hill, New York, 1986.

79. J. S. Collofello. Introduction to Software Verification and Validation. Technical Report SEI-CM-13-1.1, Software Engineering Institute, Carnegie Mellon University, 1988.

80. L. Colussi. Recursion as an Effective Step in Program Development. *ACM Transactions on Programming Languages and Systems*, 6,(1):55-67, January 1984.

81. P. A. Cook. *Nonlinear Dynmaical Systems.* Prentice Hall, Englewood Cliffs, NJ, 1986.

82. R. Courant and H. Robbins. *What is Mathematics?* Oxford University Press, New York, 1941.

83. W. J. Cullyer. High Integrity Compu · ıg. In M. Joseph, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 1-35. Springer-Verlag, New York, 1988. Lecture Notes in Computer Science 331.

84. H. C. Cunningham and G-C. Roman. A UNITY-Style Programming Logic for Shared Dataspace Programs. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):365-376, July 1990.

85. P-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming.* Pitman (John Wiley and Sons), London (New York), 1986.

86. O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming.* Academic Press, New York, 1972.

87. J. Darlington and R. M. Burstall. A System Which Automatically Improves Programs. *Acta Informatica*, 6:41-60, 1976.

88. J. Darlington and R. A. Kowalski. Declarative Systems Architecture. Technical Report SERC-DOI IKBS Architecture Study, Volume 2, U. K. Government Department of Trade and Industry, London, England, 1983.

89. M. Davis. Why Gödel Didn't Have Church's Thesis. *Information and Control*, 54(1/2):3-24, July/August 1982.

90. M. D. Davis and E. J. Weyuker. *Computability, Complexity, and Languages.* Academic Press, New York, 1983.

91. P. J. Davis and R. Hersh. *The Mathematical Experience.* Houghton Mifflin Co., Boston, MA, 1981.

92. J. M. Day. Expository Lectures on Topological Semigroups. In M. A. Arbib, editor, *Algebraic Theory of Machines, Languages, and Semigroups*, pages 269 296. Academic Press, New York, 1968.

93. J. de Bakker. *Mathematical Theory of Program Correctness.* Prentice-Hall, Englewood Cliffs, NJ, 1980.

94. J. W. de Bakker and J. I. Zucker. Denotational Semantics of Concurrency. In *Proceedings of the 14th ACM Symposium on Theory of Computing*, pages 153–158, New York, 1982. ACM.

95. J. W. de Bakker and J. I. Zucker. Processes and the Denotational Semantics of Concurrency. *Information and Control*, 54(1/2):3–24, July/August 1982. Some of the same material appeared in their paper "Denotational Semantics of Concurrency" in Proceedings of the 14th Annual ACM Symposium on Theory of Computing, San Francisco, CA, May 5-7, 1982, pages 153-158.

96. F. S. de Boer. Compositionality in the Temporal Logic of Concurrent Systems. In *Lecture Notes in Computer Science 366*, pages 406–423. Springer-Verlag, New York, 1989. Proceedings of PARLE '89.

97. W. P. de Roever. The Cooperation Test: A Syntax-Directed Verification Method. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 213–260. Springer-Verlag, Berlin, 1985. Volume 13, Series F of the NATO ASI Series.

98. E. Deak. A Transformational Derivation of a Parsing Algorithm in a High Level Language. *IEEE Transactions on Software Engineering*, 7(1):23–31, January 1981.

99. P. Degano and U. Montanari. A Model for Distributed Systems Based on Graph Rewriting. *Journal of the ACM*, 34(2):411–449, April 1987.

100. L. Farinas del Cerro. Resolution Modal Logics. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 27–55. Springer-Verlag, Berlin, 1985.

101. P. J. Denning, J. B. Dennis, and J. E. Qualitz. *Machines, Languages, and Computation.* Prentice-Hall, Englewood Cliffs, NJ, 1978.

102. W. P. DeRoever and R. Koymans. Real Time Programming and Asynchronous Message Passing. In *Proceedings of the 2nd ACM Symposium on Principles of Distributed Programming*, pages 187–197, New York, 1983. ACM Press.

103. E. W. Dijkstra. A Constructive Approach to the Problem of Program Correctness. *BIT (Published in Denmark)*, 8:174–186, 1968.

104. E. W. Dijkstra. Guarded Commands, Non-determinacy, and Formal Derivation of Programs. *Communications of the ACM*, 18(8):453–457, August 1975.

105. E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1976.

106. E. W. Dijkstra. On the Cruelty of Really Teaching Computing Science. *Communications of the ACM*, 32(12):1398–1404, December 1989.

107. G. Dromey. *Program Derivation. The Development of Programs From Specifications.* Addison-Wesley, New York, 1989.

108. J. Dugundji. *Topology.* Allen and Bacon, Rockleigh, NJ, 1966.

109. R. Duncan. A Survey of Parallel Computer Architectures. *IEEE Computer*, 23(2):5 17, February 1990.

110. W. Dunham. *Journey Through Genius: The Great Theorems of Mathematics.* John Wiley & Sons, New York, 1990.

111. S. Eilenberg and C. C. Elgot. *Recursiveness.* Academic Press, New York, 1970.

112. S. Eilenberg and J. B. Wright. Automata in General Algebras. *Information and Control*, 11:452–470, 1967.

113. C. C. Elgot. Some Geometrical Categories Associated with Flowchart Schemes. Research Report RC-6534, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, May 11 1977.

114. C. C. Elgot and J. E. Mezei. On Relations Defined by Generalized Finite Automata. *IBM Journal of Research and Development*, 9(1):47–68, January 1965.

115. E. A. Emerson and J. Y. Halpern. 'Sometimes' and 'Not Never' Revisited: On Branching Versus Linear Time Temporal Logic. *Journal of the ACM*, 33(1):151–178, January 1986.

116. Z. Esik and F. Gecseg. Type Independent Varieties and Metric Equivalence of Tree Automata. *Fundamenta Informaticae*, 9(2):205–216, 1986.

117. M. W. Evans, P. Piazza, and J. P. Dolkas. *Principles of Productive Software Management.* John Wiley and Sons, New York, 1983.

118. E. Ewald, D. E. Shasha, and A. Pnueli. Temporal Verification of Carrier-Sense Local Area Network Protocols. In *Proceedings of the 11th ACM Symposium on Principles of Programming Languages (POPL)*, pages 54–65, New York, 1984. ACM Press.

119. M. E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, 15(3):219–248, 1976.

120. R. Farrow, K. Kennedy, and L. Zucconi. Graph Grammars and Global Program Data Flow Analysis. In *17th Annual Symposium on Foundations of Computer Science*, page 42–56, Long Beach, CA, 1976. IEEE Computer Society.

121. M. S. Feather. A System for Assisting Program Transformation. *ACM Transactions on Programming Languages and Systems*, 4(1):1–20, January 1982.

122. J. H. Fetzer. Program Verification: The Very Idea. *Communications of the ACM*, 31(9):1048–1063, September 1988.

123. S. F. Fickas. *Automating the Transformational Development of Software.* PhD thesis, Dept. of Computer Science, University of California at Irvine, CA, 1982.

124. J. P. Finance et al. SPES: A Specification and Transformation System. In *Software Engineering Practice and Experience: Proceedings of the Second Software Engineering Conference*, pages 145–151, Oxford, 1984. North Oxford Academic. June, 1984 at Nice, France.

125. M. Fitting. *Computability Theory, Semantics, and Logic Programming.* Clarendon Press, Oxford, 1987.

126. R. W. Floyd. Assigning Meanings to Programs. In J. T. Schwartz, editor. *Mathematical Aspects of Computer Science*, pages 19–32. American Mathematical Society. Providence, RI, 1967.

127. G. C. Fox et al. *Solving Problems on Concurrent Processors Volume I.* Prentice Hall, Englewood Cliffs, NJ. 1988.

128. P. Fradet and D. L. Metayer. Compilation of Functional Languages by Program Transformation. *ACM Transactions on Programming Languages and Systems*, 13(1):21–51, January 1991.

129. A. A. Fraenkel. The Recent Controversies About the Foundations of Mathematics. *Scripta Mathematica*, 13:17–36, 1947.

130. N. Francez, B. Hailpern, and G. Traubenfeld. Script: A Communication Abstraction Mechanism and its Verification. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 169–212. Springer-Verlag, Berlin, 1985. Volume 13, Series F of the NATO ASI Series.

131. N. Francez and D. Kozen. Generalized Fair Termination. In *Proceedings of the 11th Annual ACM Symposium on Principles of Programming Languages*, pages 46–53, New York, 1984. ACM Press. Held at Salt Lake City, Utah on January 15-18, 1984.

132. G. Frege. *The Basic Laws of Arithmetic.* University of California Press, Los Angeles, 1964.

133. H. Friedman and M. Sheard. The Equivalence of the Disjunction and Existence Properties for Modal Arithmetic. *The Journal of Symbolic Logic*, 54(4):1456–1459, December 1989.

134. N. Gehani and A. McGettrick. *Software Specification Techniques.* Addison-Wesley, Reading, MA, 1986.

135. D. Gelernter. Multiple Tuple Spaces in Linda. In *Lecture Notes in Computer Science 366*, pages 20–27. Springer-Verlag, New York, 1989.

136. S. L. Gerhart. Correctness-Preserving Program Transformations. In *Conference Record of the Second ACM Symposium on Principles of Programming Languages*, pages 54–66, New York, 1975. ACM.

137. S. L. Gerhart. Proof Theory of Partial Correctness Verification Systems. *SIAM Journal on Computing*, 5(3):355–377, September 1976.

138. A. Ginzburg. *Algebraic Theory of Automata.* Academic Press, New York, 1968.

139. R. Goldblatt. *Logics of Time and Computation.* Center for the Study of Language and Information, Leland Stanford Junior University, 1987.

140. M. J. Gordon. *The Denotational Description of Programming Languages.* Springer-Verlag, New York, 1979.

141. K. J. Gough. *Syntax Analysis and Software Tools.* Addison-Wesley, Reading, MA. 1988.

142. J. S. Gourlay, W. C. Rounds, and R. Statman. On Properties Preserved by Contractions of Concurrent Systems. In *Semantics of Concurrent Computation. Lecture Notes in Computer Science 70*, pages 51–65. Springer-Verlag, New York. 1979.

143. G. Gratzer. *Universal Algebra.* Springer-Verlag, New York, 1979.

144. S. A. Greibach. *Theory of Program Structures: Schemes, Semantics, Verification.* Springer-Verlag, New York, 1975. Lecture Notes in Computer Science No. 36.

145. D. Gries. *The Science of Programming.* Springer-Verlag, New York, 1981.

146. I. Guessarian. *Algebraic Semantics.* Lecture Notes in Computer Science 99. Springer-Verlag, Berlin, 1981.

147. B. T. Hailpern. *Verifying Concurrent Processes Using Temporal Logic.* Springer Verlag, Berlin, 1982.

148. R. Hale and B. Moszkowski. Parallel Programming in Temporal Logic. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE Parallel Architectures and Languages Europe, Volume II*, pages 277–296. Springer-Verlag, Berlin, 1987. Lecture Notes in Computer Science 259.

149. P. B. Hansen. *Operating System Principles.* Prentice-Hall, Englewood Cliffs, NJ, 1973.

150. P. Brinch Hansen. Distributed Processes: A Concurrent Programming Concept. *Communications of the ACM*, 21(11):934–940, November 1978.

151. S. L. Hantler and J. C. King. An Introduction to Proving the Correctness of Programs. *ACM Computing Surveys*, 8(3):331–353, September 1976.

152. D. Harel. *Algorithmics The Spirit of Computing.* Addison-Wesley, Wokingham, England, 1987.

153. M. Hennessy and R. Milner. Algebraic Laws for Nondeterminism and Concurrency. *Journal of the ACM*, 32(1):137–161, January 1985.

154. M. H. Hennessy. *Algebraic Theory of Processes.* The MIT Press, Cambridge, MA, 1988.

155. F. Hennie. *Introduction to Computability.* Addison-Wesley, Reading, MA, 1977.

156. H. Herrlich and G. E. Strecker. *Category Theory.* Allyn and Bacon Inc., Boston, MA, 1973.

157. I. N. Herstein. *Topics in Algebra.* Xerox College Publishing, Lexington, MA, 1975.

158. O. Herzog. Static Analysis of Concurrent Processes for Dynamic Properties Using Petri Nets. In *Semantics of Concurrent Computation, Lecture Notes in Computer Science 70*, pages 66–90. Springer-Verlag, New York, 1979.

159. C. E. Hewitt and H. Baker. Actors and Continuous Functionals. In E. J. Neuhold, editor, *Proceedings of the IFIP Working Conference on Formal Description of Programming Concepts*, pages 367–387. IFIP, 1977.

160. G. F. Hice, W. J. Turner, and L. F. Cashwell. *System Development Methodology.* North-Holland, New York, 1981.

161. C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–583, October 1969.

162. C. A. R. Hoare. Proof of a Program: FIND. *Communications of the ACM*, 14(1):39–45, January 1971.

163. C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, October 1974.

164. C. A. R. Hoare. Farewell Dinner Speech. In M. Broy and G. Schmidt, editors, *Theoretical Foundations of Programming Methodology*, page vii. D. Reidel, Dordrecht, Holland, 1982. Volume 91, Series C of the NATO ASI Series. This quote appeared in the Preface.

165. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, Englewood Cliffs, NJ, 1985.

166. D. R. Hofstadter. *Godel, Escher, Bach: An Eternal Golden Braid.* Vintage Books, New York, 1980.

167. C. J. Hogger. *Introduction to Logic Programming.* Academic Press, London, 1984.

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

169. G. M. Hopper. A Glossary of Computer Terminology. *Computer and Automation*, 3(5):14–22, May 1954. The Grace Hopper.

170. S. W. Hornick. A Unified Approach to the Analysis and Synthesis of Systolic Arrays. Master's thesis, Applied Computation Theory Group, University of Illinois, Urbana, IL, 1985. Also listed as Report ACT-56

171. W. E. Howden. *Functional Program Testing and Analysis.* McGraw-Hill, New York, 1987.

172. C. H. Huang and C. Lengauer. An Implemented Method for Incremental Systolic Design. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE Parallel Architectures and Languages Europe, Volume I*, pages 160–177. Springer-Verlag, Berlin, 1987. Lecture Notes in Computer Science 258.

173. C. H. Huang and C. Lengauer. The Derivation of Systolic Implementations of Programs. *Acta Informatica*, 24(6):595–632, 1987.

174. C-H. Huang and C. Lengauer. An Incremental Mechanical Development of Systolic Solutions to the Algebraic Path Problem. *Acta Informatica*, 27(2):97–124, 1989.

175. P. Hudak. Conception, Evolution, and Application of Functional Programming Languages. *ACM Computing Surveys*, 21(3):359–411, September 1989.

176. G. E. Hughes and M. J. Cresswell. *An Introduction to Modal Logic.* Methuen and Co. Ltd., London, 1972.

177. V. I. Istratescu. *Fixed Point Theory.* D. Reidel Publishing Co., Dordrecht, Holland, 1981.

178. K. E. Iverson. Notation as a Tool of Thought. *Communications of the ACM*, 23(8):444–465, August 1980.

179. M. S. Jaffe and N. G. Leveson. Completeness, Robustness, and Safety in Real-Time Software Requirements Specification. In *Proceedings of the 11th International Conference on Software Engineering*, pages 302 311, Washington, DC, 1989. IEEE Computer Society Press.

180. F. Jahanian and A. K. Mok. Safety Analysis of Timing Properties in Real-Time Systems. *IEEE Transactions on Software Engineering*, SE-12(9):890–904, September 1986.

181. R. Janicki. A Characterization of Concurrency-Like Relations. In *Semantics of Concurrent Computation, Lecture Notes in Computer Science 70*, pages 109–122. Springer-Verlag, New York, 1979.

182. T. Jech and K. Hrbacek. *Introduction to Set Theory.* Marcel Dekker, New York, 1978.

183. C. B. Jones. Formal Development of Correct Algorithms: An Example Based on Earley's Recognizer. *ACM SIGPLAN Notices*, 7(1), January 1972.

184. C. B. Jones. *Software Development: A Rigorous Approach.* Prentice Hall International, Englewood Cliffs, NJ, 1980.

185. C. B. Jones. Tentative Steps Towards a Development Method for Interfering Programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, October 1983.

186. G. W. Jones. *Software Engineering.* John Wiley & Sons, New York, 1990.

187. D. W. Jordan and P. Smith. *Nonlinear Ordinary Differential Equations (2nd Edition).* Clarendon Press, Oxford, 1987.

188. T. E. Cheatham Jr. and B. Wegbreit. A Laboratory for the Study of Automating Programming. In *Proceedings of AFIPS 1972 Spring Joint Computer Conference*, pages 11–21, Montvale, NJ, 1972. AFIPS Press.

189. J. L. Kelley. *General Topology.* D. Van Nostrand Co., Princeton, NJ, 1955.

190. A. J. Kfoury, R. N. Moll, and M. A. Arbib. *A Programming Approach to Computability.* Springer-Verlag, New York, 1982.

191. N. Klarlund and F. B. Schneider. Verifying Saftey Properties Using Nondeterministic Infinite-state Automata. Interim report for the Office of Naval Research, September 1989. DTIC Number ADA212 598.

192. S. C. Kleene. *Introduction to Metamathematics.* North-Holland, Amsterdam, 1952.

193. E. Knapp. An Exercise in the Formal Derivation of Parallel Programs: Maximum Flow in Graphs. *ACM Transactions on Programming Languages and Systems*, 12(2):203–223, April 1990.

194. D. E. Knuth. The Dangers of Computer Science. In P. Suppes et. al., editor, *Logic, Methodology, and Philosophy of Science IV*, pages 189–196. North Holland, Amsterdam, 1973.

195. R. Kowalski. *Logic for Problem Solving.* North-Holland, New York, 1979.

196. F. Kroger. *Temporal Logic of Programs.* Springer-Verlag, Berlin, 1987.

197. D. J. Kuck. *The Structure of Computers and Computations.* John Wiley & Sons, New York, 1978.

198. R. Kunze and K. Hoffman. *Linear Algebra.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1971.

199. J. Lambek. Deductive Systems and Categories III. Cartesian Closed Categories, Intuitionist Propositional Calculus, and Combinatory Logic. In F. W. Lawvere, editor, *Toposes, Algebraic Geometry and Logic*, pages 57-82. Springer-Verlag, New York, 1972.

200. L. Lamport. Proving the Correctness of Mutiprocess Programs. *IEEE Transactions on Software Engineering*, 3(2):125-143, March 1977.

201. L. Lamport. An Axiomatic Semantics of Concurrent Programming Languages. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 77-122. Springer-Verlag, Berlin, 1985. Volume 13, Series F of the NATO ASI Series.

202. S. Mac Lane. One Universe as a Foundation for Category Theory. In *Lecture Notes in Mathematics 106*, pages 192-200, Berlin, 1969. Springer-Verlag.

203. S. Mac Lane. *Categories for the Working Mathematician.* Springer-Verlag, New York, 1971.

204. J. P. LaSalle. *The Stability of Dynamical Systems.* SIAM, Philadelphia, PA, 1976.

205. D. J. Lehman. Categories for Fixpoint Semantics. In *17th Annual Symposium on Foundations of Computer Science*, pages 122-126, Long Beach, CA, 1976. IEEE Computer Society.

206. P. M. Lenders. Distributed Computing with Single Read-Single Write Variables. *IEEE Transactions on Software Engineering.* 15(5):569-574. May 1989.

207. C. Lengauer and J. W. Sanders. The Projection of Systolic Programs. In *Proceedings of the International Conference on the Mathematics of Program Construction: 375th Anniversary of the Groningen University*, pages 307-324, Berlin, 1989. Springer-Verlag.

208. H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall Inc., Englewood Cliffs, NJ, 1981.

209. J. Loeckx and K. Sieber. *The Foundations of Program Verification.* John Wiley and Sons, New York, 1984.

210. D. B. Lomet. Subsystems of Processes with Deadlock Avoidance. *IEEE Transactions on Software Engineering*, 6(3):297-304, May 1980.

211. R. L. London. Proof of Algorithms: A New Kind of Certification. *Communications of the ACM*, 13(6):371-373, June 1970.

212. D. B. Loveman. Program Improvement by Source to Source Transformation. *Journal of the ACM*, 24(1):121-145, January 1977.

213. Inmos Ltd. *Occam Programming Manual.* Prentice-Hall. Englewood Cliffs, NJ, 1985.

214. G. H. MacEwen and D. B. Skillicorn. Using Higher-Order Logic for Modular Specification of Real-Time Distributed Systems. In M. Joseph, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 1-35. Springer-Verlag, New York, 1988. Lecture Notes in Computer Science 331.

215. B. Maher. Automatic Program Improvement: Variable Usage Transformations. *ACM Transactions on Programming Languages and Systems*, 5(2):236–264, April 1983.

216. E. G. Manes and M. A. Arbib. *Algebraic Approaches to Program Semantics.* Springer-Verlag, New York, 1986.

217. Z. Manna. *Mathematical Theory of Computation.* MacGraw Hill, New York, 1974.

218. Z. Manna and A. Pnueli. Axiomatic Approach to Total Correctness of Programs. *Acta Informatica*, 3:243–263, 1974.

219. Z. Manna and A. Pnueli. Temporal Verification of Concurrent Programs: The Temporal Framework For Concurrent Programs. In R. S. Boyer and J. S. Moore, editors, *The Correctness Problem in Computer Science*, chapter 5. Academic Press, London, 1981.

220. Z. Manna and A. Pnueli. Verification of Concurrent Programs, Part I: The Temporal Framework. Technical Report STAN-CS-81-836, Department of Computer Science, Stanford University, June 1981.

221. Z. Manna and A. Pnueli. How To Cook a Temporal Proof System For Your Pet Language. In *Symposium on Principles of Programming Languages*, pages 141–154, Austin, Texas, January 1983.

222. Z. Manna and A. Pnueli. Specification and Verification of Concurrent Programs by ∀-automata. Technical Report STAN-CS-88-1230, Stanford University Department of Computer Science, Stanford, CA, 1988.

223. Z. Manna and R. Waldinger. Is 'Sometime' Sometimes Better Than 'Always'?: Intermittent Assertions in Proving Program Correctness. *Communications of the ACM*, 21:159–172, 1978.

224. Z. Manna and R. Waldinger. Is 'Sometimes' Better Than 'Always'? *Communications of the ACM*, 21(2):139–172, February 1978.

225. Z. Manna and R. Waldinger. *The Logical Basis for Computer Programming. Volume 1.* Addison-Wesley Publishing Co., Reading, MA, 1985.

226. Z. Manna and P. Wolper. Synthesis of Communicating Processes from Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 6(1):68–93, January 1984.

227. M. J. Manthey and B. M. E. Moret. The Computational Metaphor and Quantum Physics. *Communications of the ACM*, 26(2):137–145, February 1983.

228. A. J. Martin. A Message Passing Model for Highly Concurrent Computation. In *Volume I of the Third Conference on Hypercube Concurrent Computers and Applications*, pages 520–527. ACM Press, 1988.

229. H. Maurer. New Aspects of Homomorphisms. In K. Weihrauch, editor, *Theoretical Computer Science 4th GI Conference, Lecture Notes in Computer Science 67*, pages 10–24. Springer Verlag, New York, 1979.

230. J. McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine. *Communications of the ACM*, 3(4):184–195, 1960.

231. J. McCarthy. Towards a Mathematical Science of Computation. In C. M. Popplewell, editor, *Information Processing, Proceedings of IFIP Congress 62*, pages 21–28. North-Holland, Amsterdam, 1962.

232. J. McCarthy. A Basis For a Mathematical Theory of Computation. In P. Braffort and D. Hirschberg, editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam, 1963.

233. J. McCarthy and J. Painter. Correctness of a Compiler for Arithmetic Expressions. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, pages 33–41. American Mathematical Society, Providence, RI, 1967.

234. C. E. McDowell and D. P. Helmbold. Debugging Concurrent Programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.

235. R. McNaughton. Testing and Generating Infinite Sequences by a Finite Automaton. *Information and Control*, 9:521–530, 1966.

236. L. J. Mekly. Software Design Representation Using Abstract Process Networks. *IEEE Transactions on Software Engineering*, 6(5):420–434, September 1980.

237. S. M. Merritt. *The Role of High Level Specification in Programming by Transformations: Specification and Transformation by Parts.* PhD thesis, Dept. of Computer Science, Courant Institute, New York University, New York, 1982.

238. R. E. Miller and T. Kasai. Comparing Models of Parallel Computation by Homomorphisms. In *Proceedings of the IEEE Third International Computer Systems and Applications Conference*, pages 794–799, New York, 1979. IEEE Computer Society.

239. R. A. De Millo, R. J. Lipton, and A. J. Perlis. Social Processes and Proofs of Theorems and Programs. *Communications of the ACM*, 22(5):271–280, May 1979.

240. H. D. Mills, V. R. Basili, J. D. Gannon, and R. C. Hamlet. Mathematical Principles for a First Course in Software Engineering. *IEEE Transactions on Software Engineering*, 15(5):550–559, May 1989.

241. R. Milner. Implementation and Application of Scott's Logic for Computable Functions. In *Proceedings of the ACM Conference*, New York, 1972. ACM.

242. R. Milner. *A Calculus of Communicating Systems.* Lecture Notes in Computer Science 92. Springer Verlag, New York, 1980.

243. R. Milner. An Algebraic Definition of Simulation Between Programs. Technical Report Stanford Artificial Intelligence Memo AIM-142, Stanford University, February 1983. Also listed as Computer Science Dept Report CS-205.

244. R. Milner. Calculi for Synchrony and Asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

245. M. L. Minsky. Some Methods of Artificial Intelligence and Heuristic Programming. In *Mechanisation of Thought Processes*, pages 3 28, London, 1959. Her Majesty's Stationery Office. National Physical Laboratory Symposium No. 10, also contains

the article by Selfridge that describes his Pandemonium System, from which comes the term and concept of a 'demon'.

246. M. L. Minsky. *Computation: Finite and Infinite Machines.* Prentice-Hall Inc., Englewood Cliffs, NJ, 1967.

247. B. Mitchell. *Theory of Categories.* Academic Press Inc., New York, 1965.

248. F. C. Moon. *Chaotic Vibrations An Introduction for Applied Scientists and Engineers.* John Wiley and Sons, New York, 1987.

249. E. F. Moore. *Sequential Machines: Selected Papers.* Addison-Wesley, Reading, MA, 1964.

250. P. Morrison and E. Morrison. *Charles Babbage and His Calculating Engines.* Dover, New York, 1961.

251. B. Moszkowski. A Temporal Logic For Multilevel Reasoning About Hardware. *Computer*, 18(2):10–19, February 1985.

252. B. C. Moszkowski. *Executing Temporal Logic Programs.* Cambridge University Press, Cambridge, England, 1986.

253. W. Myers. Build Defect-Free Software, Fagan Urges. *Computer (IEEE)*, 23(8):112–113, August 1990.

254. A. T. Nakagawa and K. Futatsugi. Stepwise Refinement Process with Modularity: An Algebraic Approach. In *Proceedings of the 11th International Conference on Software Engineering*, pages 166–177, Washington, DC, 1989. IEEE Computer Society Press.

255. P. Naur. Programming Languages, Natural Languages, and Mathematics. In *Conference Record of the Second ACM Symposium on Principles of Programming Languages*, pages 137–148, New York, 1975. ACM.

256. A. W. Naylor and G. R. Sell. *Linear Operator Theory in Engineering and Science.* Springer-Verlag, New York, 1982.

257. V. Nguyen, D. Gries, and S. Owicki. A Model and Temporal Proof System for Networks of Processes. In *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Languages*, pages 121–131, New York, 1985. ACM Press. Held at New Orleans, January 14-16, 1985.

258. H. R. Nielson and F. Nielson. Functional Completeness of the Mixed Lambda-Calculus and Combinatory Logic. Technical Report R 88-23, Institut for Elektroniske Systemer Afdeling for Matematik og Datalogi, Aalborg Universitetscenter, Aalborg, Denmark, July 1988.

259. M. Nielson, G. Plotkin, and G. Winskel. Petri Nets, Event Structures and Domains. In *Semantics of Concurrent Computation, Lecture Notes in Computer Science 70*, pages 266–284. Springer-Verlag, New York, 1979.

260. N. J. Nilsson. *Principles of Artificial Intelligence.* Tioga Publishing Co., Palo Alto, CA, 1980.

261. M. Nivat. Behaviors of Processes and Synchronized Systems of Processes. In M. Broy and G. Schmidt, editors, *Theoretical Foundations of Programming Methodology*, pages 473–554. D. Reidel, Dordrecht, Holland, 1982. Volume 91, Series C of the NATO ASI Series.

262. E-R. Olderog. Specification-oriented Programming in TCSP. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 397–438. Springer-Verlag, Berlin, 1985.

263. J. S. Ostroff. *Temporal Logic for Real-Time Systems.* Research Studies Press, Ltd (John Wiley & Sons), Taunton, England (New York), 1989.

264. M. A. Ould and C. Unwin. *Testing in Software Development.* Cambridge University Press, Cambridge, England, 1986.

265. S. Owicki. *Axiomatic Proof Techniques for Parallel Programs.* PhD thesis, Dept. of Computer Science, Cornell University, NY, 1975. PhD Dissertation TR75-251.

266. S. Owicki and D. Gries. Verifying Properties of Parallel Programs: An Axiomatic Approach. *Communications of the ACM*, 19(5):279–285, May 1976.

267. S. Owicki and L. Lamport. Proving Liveness Properties of Concurrent Programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, July 1982.

268. S. S. Owicki and D. Gries. Verifying Properties of Parallel Programs: An Axiomatic Approach. *Communications of the ACM*, 19(5):279–285, May 1976.

269. J. Parrow. Synchronisation Flow Algebra. Technical Report ECS-LFCS-87-35, Laboratory for Foundations of Computer Science, Dept of Computer Science, University of Edinburgh, August 1987.

270. J. Parrow. The Expressive Power of Simple Parallelism. In *Lecture Notes in Computer Science 366*, pages 389–405. Springer-Verlag, New York, 1989.

271. H. Partsch. Transformational Program Development in a Particular Problem Domain. *Science of Computer Programming*, 7:99–241, 1986. Published by Elsevier Science (North-Holland).

272. H. Partsch and R. Steinbruggen. Program Transformation Systems. *ACM Computing Surveys*, 15(3):199–236, September 1983.

273. A. Pazy. *Semigroups of Linear Operators and Applications to Partial Differential Equations.* Springer-Verlag, New York, 1983.

274. R. Penrose. *The Emperor's New Mind.* Oxford University Press, Oxford, 1989.

275. R. H. Perrott. Languages for Parallel Computers. In R. M. McKeag and A. M. Macnaghten, editors, *On the Construction of Programs.* Cambridge University Press, Cambridge, England, 1979.

276. R. H. Perrott. *Parallel Programming.* Addison-Wesley, Wokingham, England, 1987.

277. I. Peterson. Finding Fault, The Formidable Task of Eradicating Software Bugs. *Science News*, 139(7):104–106, February 16 1991.

278. J. L. Peterson. Petri Nets. *ACM Computing Surveys*, 9(3):223-252, September 1977.

279. J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981.

280. A. Pettorossi. A Powerful Strategy for Deriving Efficient Programs by Transformation. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, pages 74–84, New York, 1984. ACM Press.

281. B. P. Phillips and S. G. Howe. Verification - The Practical Problems. In B. K. Daniels, editor, *Achieving Safety and Reliability with Computer Systems*, pages 89-99. Elsevier Applied Science, London, 1987. Proceedings of the Safety and Reliability Society Symposium 1987.

282. D. Pilaud and N. Halbwachs. From a Synchronous Declarative Language to a Temporal Logic Dealing with Multiform Time. In M. Joseph, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 99–110. Springer-Verlag, New York, 1988. Lecture Notes in Computer Science 331.

283. D. H. Pitt et al., editors. *Category Theory and Computer Science.* Lecture Notes in Computer Science 283. Springer Verlag, Berlin, 1987. Proceedings of the Edinburgh, U.K. Conference, September, 1987.

284. G. A. Place. A Graphical Notation for the Formal Specification of Software. Master's thesis, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, OH, 1990.

285. A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th Symposium on the Frontiers of Computer Science*, pages 46–57, Providence, RI, 1977. IEEE.

286. A. Pnueli. The Temporal Semantics of Concurrent Programs. In *Semantics of Concurrent Computation, Lecture Notes in Computer Science 70*, pages 1-20. Springer-Verlag, New York, 1979.

287. A. Pnueli. In Transition From Global to Modular Temporal Reasoning About Programs. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 123–146. Springer-Verlag, Berlin, 1985. Volume 13, Series F of the NATO ASI Series.

288. A. Pnueli and D. Harel. Applications of Temporal Logic to the Specification of Real-Time Systems. In M. Joseph, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 84–98. Springer-Verlag, New York, 1988. Lecture Notes in Computer Science 331.

289. L. L. Pollock and M. L. Soffa. Incremental Compilation of Locally Optimized Code. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages (POPL)*, pages 152–164, New York, 1985. ACM Press.

290. C. D. Polychronopoulos. Automatic Restructuring of Fortran Programs for Parallel Execution. In R. Dierstein et. al., editor, *Parallel Computing in Science and Engineering, Lecture Notes in Computer Science 295*, pages 107-130. Springer-Verlag, New York, 1987.

291. M. B. Pour-El and I. Richards. The Wave Equation with Computable Initial Data such that its Unique Solution is not Computable. *Advances in Mathematics*, 39:215-239, 1981.

292. V. R. Pratt. Semantical Considerations on Floyd-Hoare Logic. In *17th Annual Symposium on Foundations of Computer Science*, pages 109-121, Long Beach, CA, 1976. IEEE Computer Society.

293. R. S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill Series in Software Engineering and Technology. McGraw-Hill, New York, 1982.

294. A. J. Pritchard and R. F. Curtain. *Functional Analysis in Modern Applied Mathematics.* Academic Press, London, 1977.

295. W. V. Quine. Set-Theoretic Foundations for Logic. *The Journal of Symbolic Logic*, 1(2):45–57, June 1936.

296. W. V. Quine. Towards a Calculus of Concepts. *The Journal of Symbolic Logic*, 1(1):2–25, March 1936.

297. L. H. Reeker. State Graphs and Context Free Languages. In Z. Kohavi and A. Paz, editors, *Theory of Machines and Computations*, pages 143-152. Academic Press, New York, 1971.

298. H. Reichel. *Initial Computability, Algebraic Specifications, and Partial Algebras.* Clarendon Press, Oxford, 1987.

299. M. Rem. Trace Theory and Systolic Computations. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE Parallel Architectures and Languages Europe, Volume I*, pages 14–33. Springer-Verlag, Berlin, 1987. Lecture Notes in Computer Science 258.

300. N. Rescher and A. Urquhart. *Temporal Logic.* Springer-Verlag, New York, 1971.

301. G. E. Revesz. *Lambda-Calculus, Combinators, and Functional Programming.* Cambridge University Press, Cambridge, 1988.

302. C. Rich and R. C. Waters. *The Programmer's Apprentice.* ACM Press and Addison-Wesley, New York, 1990.

303. R. D. Richtmyer and K. W. Morton. *Difference Methods for Initial-Value Problems.* Interscience Publishers, New York, 1967.

304. A. Robinson. *Nonstandard Analysis.* North Holland, Amsterdam, 1966.

305. H. L. Royden. *Real Analysis.* MacMillan Publishing Co., New York, 1968.

306. Bertrand Russell. On Order in Time. *Proceedings of the Cambridge Philosophical Society*, 32:216–228, 1936.

307. T. L. Saaty. *Modern Nonlinear Equations.* McGraw-Hill, New York, 1967.

308. S. R. Schach. *Software Engineering.* Aksen Associates (Irwin), Homewood, IL, 1990.

309. P. Schnupp and L. W. Bernhard. *Productive Prolog Programming.* Prentice-Hall, Englewood Cliffs, NJ, 1987.

310. Uwe Schoning. *Logic for Computer Scientists.* Birkhauser, Boston, 1989.

311. D. Scott. Outline of a Mathematical Theory of Computation. In *Proccedings of the Fourth Princeton Conference on Information Sciences and Systems*, pages 169 176. Princeton, NJ Press, 1970. Princeton University.

312. D. Scott. Continuous Lattices. In F. W. Lawvere, editor, *Toposes, Algebraic Geometry and Logic*, pages 97–136. Springer-Verlag, New York, 1972.

313. D. S. Scott. Lectures on a Mathematical Theory of Computation. In M. Broy and G. Schmidt, editors, *Theorctical Foundations of Programming Methodology*, pages 145–292. D. Reidel, Dordrecht, Holland, 1982. Volume 91, Series C of the NATO ASI Series.

314. D. S. Scott and C. Strachey. Toward a Mathematical Semantics for Computer Languages. In J. Fox, editor, *Proceedings of the Symposium on Computers and Automata*, pages 19–46, New York, 1971. Polytechnic Institute of Brooklyn Press.

315. C. E. Shannon. The Lattice Theory of Information. *I.R.E. Transactions of the Professional Group on Information Theory*, PGIT-1:105–106, February 1953.

316. E. Shapiro. Concurrent Prolog: A Progress Report. *IEEE Computer*, 19(8):44–58, August 1986.

317. J. R. Shoenfield. *Mathematical Logic.* Addison-Wesley, Reading, MA, 1967.

318. J. Shurkin. *Engines of the Mind.* W. W. Norton, New York, 1984.

319. A. P. Sistla and E. M. Clarke. The Complexity of Propositional Linear Temporal Logics. *Journal of the ACM*, 32(3):733–749, July 1985.

320. D. B. Skillicorn. A Taxonomy for Computer Architectures. *IEEE Computer*, 21(11):46–57, November 1988.

321. D. B. Skillicorn. Architecture-Independent Parallel Computation. *IEEE Computer*, 23(12):38–51, December 1990.

322. I. Sommerville. *Software Engineering.* International Computer Science Series. Addison-Wesley, Wokingham, England, 1989.

323. N. Soundararajan. Axiomatic Semantics of Communicating Sequential Processes. *ACM Transactions on Programming Languages and Systems*, 6(4):647 662, October 1984.

324. J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, Cambridge, MA [illegible].

325. J. E. Stoy. Mathematical Foundations. In D. Bj/orner and C. B. Jones, editors, *Formal Specification and Software Development*, pages 47 81. Prentice-Hall International, Englewood Cliffs, NJ, 1982.

326. C. B. Stunkel, B. Janssens, and W. K. Fuchs. Address Tracing for Parallel Machines. *IEEE Computer*, 24(1):31–38, January 1991.

327. T. A. Sudkamp. *Languages and Machines.* Addison-Wesley Publishing Co., Reading, MA, 1988.

328. A. E. Taylor and D. C. Lay. *Introduction to Functional Analysis.* John Wiley and Sons, New York, 1980.

329. R. D. Tennent. The Denotational Semantics of Programming Languages. *Communications of the ACM*, 19(8):437–453, August 1976.

330. J. W. Thatcher. Notes on Algebraic Fundementals for Theoretical Computer Science. In J. W. de Bakker and J. V. Leeuwen, editors, *Foundations of Computer Science III, Part 2: Languages, Logic, Semantics*, pages 84-164. Mathematisch Centrum, Amsterdam, 1979. Mathematical Centre Tracts 109.

331. P. Thomas, H. Robinson, and J. Emms. *Abstract Data Types: Their Specification, Representation, and Use.* Clarendon Press, Oxford, 1988.

332. B. Thomsen. A Calculus of Higher-Order Communicating Systems. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 143–154, New York, 1989. ACM Press. Held at Austin, Texas on January 11-13, 1989.

333. F. G. Tricomi. *Integral Equations.* Dover Publications, New York, 1985.

334. A. M. Turing. On Computable Numbers, With an Application to the Entscheidungsproblem. In M. Davis, editor, *The Undecidable*, pages 115-153. Raven Press, Hewlett, NY, 1965.

335. L. G. Valiant. A Theory of the Learnable. *Communications of the ACM*, 27(11), 1984.

336. N. W. P. van Diepen. Implementation of Modular Algebraic Specifications. Technical Report CS-R8801, Center for Mathematics and Computer Science, Amsterdam, The Netherlands, January 1988.

337. N. W. P. van Diepen. Small - Dynamic Semantics of a Language with GOTOs. In J. A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, chapter 3. ACM Press, New York, 1989.

338. R. van Glabbeek and F. Vaandrager. Petri Net Models for Algebraic Theories of Concurrency. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE Parallel Architectures and Languages Europe, Volume II*, pages 224–242. Springer-Verlag, Berlin, 1987. Lecture Notes in Computer Science 259.

339. A. van Wijngaarden. Recursive Definition of Syntax and Semantics. In T. B. Steel Jr., editor, *Formal Language Description Languages for Computer Programming*, pages 13–24. North-Holland, Amsterdam, 1966.

340. A. Veevers, E. Petrova, and A. C. Marshall. Statistical Methods for Software Reliability Assessment, Past, Present and Future. In B. K. Daniels, editor. *Achieving Safety and Reliability with Computer Systems*, pages 89 99. Elsevier Applied Science, London, 1987. Proceedings of the Safety and Reliability Society Symposium 1987.

341. S. R. Vegdahl. A Survey of Proposed Architectures for the Execution of Functional Languages. *IEEE Transactions on Computers*, 33(12):1050 1071, December 1984.

342. P. Vishnubhotla. The ALPS Model of Concurrency. Technical Report OSU-CISRC-12/89 TR57, Computer and Information Sciences, Ohio State University, Columbus, OH, December 1989.

343. F. W. von Henke. Formal Transformations and the Development of Programs. In *Mathematical Foundations of Computer Science 1977*, pages 288–296. Springer-Verlag, Berlin, 1977. Lecture Notes in Computer Science 53.

344. R. J. Waldinger and Z. Manna. Towards Axiomatic Program Synthesis. *Communications of the ACM*, 14(3):151–165, March 1971.

345. M. Wand. Fixed-Point Constructions in Order-Enriched Categories. Technical Report Tech. Report Number 23, Dept. of Computer Science, Indiana University, April 1975.

346. R. C. Waters. Automatic Transformation of Series Expressions into Loops. *ACM Transactions on Programming Languages and Systems*, 13,(1):52–98, January 1991.

347. B. Wegbreit. Goal-Directed Program Transformation. In *Proceedings of the 3rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 153–170, New York, 1976. ACM Press.

348. H. Wertz. *Automatic Correction and Improvement of Programs.* Ellis Horwood Limited, Chichester, England, 1987.

349. S. J. Westfold, L. Z. Markosian, and W. A. Brew. *Knowledge-Based Software Development from Requirements to Code.* Reasoning Systems Inc., Palo Alto, CA. Describes the concepts behind REFINE 2.0.

350. D. Whitfield and M. L. Soffa. An Approach to Ordering Optimizing Transformations. *SIGPLAN Notices*, 25(3):137–146, March 1990. This issue was the Proceedings of the ACM/SIGPLAN Second Symposium on Principles and Practice of Parallel Programming (PPOPP).

351. J. Widom and J. Hooman. A Temporal-Logic Based Compositional Proof System for Real-Time Message Passing. In *Lecture Notes in Computer Science 366*, pages 424–441. Springer-Verlag, New York, 1989.

352. S. Wiggins. *Introduction to Applied Nonlinear Dynamical Systems and Chaos.* Springer-Verlag. New York, 1990.

353. D. S. Wile. Type Transformations. *IEEE Transactions on Software Engineering*, 7(1):32–39, January 1981.

354. J. M. Wing. A Specifier's Introduction to Formal Methods. *IEEE Computer*, 23(9):8–26, September 1990.

355. N. Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14(4):221–227, April 1971.

356. N. Wirth. *Systematic Programming: An Introduction.* Prentice Hall, Englewood Cliffs, NJ, 1973.

357. N. Wirth. On the Composition of Well Structured Programs. *ACM Computing Surveys*, 6(4):247–260. December 1974.

358. N. Wirth. *Programming in Modula-2.* Springer-Verlag, New York. 1983.

359. D. Wood. *Theory of Computation.* Harper and Row, New York. 1987.

360. S. Yang and J. Juang. Message Flow Analysis and Run-Time Verification for Parallel Programs. In *Proceedings of the 1989 International Conference on Parallel Processing Volume II*, pages 19-22, University Park, PA, 1989. Pennsylvania State University Press.

361. R. T. Yeh. Guest editorial. *ACM Computing Surveys*, 8(3):301-303, September 1976.

362. P. Zave. The Operational Versus the Conventional Approach to Software Development. *Communications of the ACM*, 27(2):101–115, February 1984.

363. M. V. Zelkowitz. A Functional Correctness Model of Program Verification. *IEEE Computer*, 23(11):30–40, November 1990.

364. E. Zermelo. Investigations In The Foundations of Set Theory I. In J. van Heijenoort, editor, *From Frege to Godel*, pages 199-215. Harvard University Press, Cambridge, MA, 1967. This article is the English translation of 'Untersuchungen uber die Grund lagen der Mengenlehre I' which appeared in Mathematische Annalen, volume 65, 1908, pages 261-281. In this same book appears the English translation 'A New Proof of the Possibilities of a Well-Ordering', pages 183-198.

Vita

Jeffrey A Simmers was born in Harrisburg, Pennsylvania on January 9, 1953. After completing two years of college at Texas A&M University as a geophysics major he enlisted in the United States Air Force in 1976. In 1978 he was selected to finish his undergraduate schooling at the Univeristy of Illinois under the Airman Education and Commissioning Program, graduating with a B.S.E.E. in 1979. After two years at the Foreign Technology Division at Wright-Patterson AFB he was selected to attend the Air Force Institute of Technology from which he graduated with a M.S.E.E. in 1983. His next assignment was with the Air Force Operational Test and Evaluation Center at Eglin AFB, after which he reentered the Air Force Institute of Technology in 1988, with electrical and computer engineering as the major field of study and mathematics as a minor.

Permanent address: 22B Scott Circle
Hanscom AFB, MA 01731