5-2008

CAPS: Concurrent Automatic Programming System

Ken Kennedy
Clemson University, eken@clemson.edu

CAPS: CONCURRENT AUTOMATIC PROGRAMMING SYSTEM

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Science

---

by
Ken Edward Kennedy
May 2008

Accepted by:
Dr. D. E. Stevenson, Committee Chair
Dr. S. T. Hedetniemi
Dr. S. M. Hedetniemi
Dr. Dan Warner
Dr. Kelly Smith

Abstract

In the past few years, the focus in microprocessors has shifted from increasing speed to creating processors that contain multiple cores. In order to effectively use the new processors, concurrent specifications and applications must be developed. Additionally, there are many applications that require the specifications to be provably correct. CAPS (Concurrent Automatic Programming System) is designed to aid the user in the creation, execution, and formal verification of concurrent specifications.

The specification language of CAPS (CAPSL) is a very high-level language designed for concurrency and automatic conversion to a colored Petri net (CP-net). For each statement of the language, there exists a mapping to a colored Petri net. Once in the form of a CP-net, the specifications can be formally verified or a simulation can be run.

Acknowledgments

Foremost, I would like to thank God. It is said that through Him all things are possible, and it is only with His help that I was able to finish.

I would like to thank my wife, and best friend, Nell. She kept me going when I did not want to and made sure that I did not quit. I know that at times her task was not pleasant, but she never gave up on me.

I appreciate all of the work and patience of my advisor, Dr. Stevenson. I did not imagine that it would take so long when I first started, but he was always there to help me.

Last of all, I would like to thank my parents who have always stood behind me and supported my decisions. Since I was a child, I have known that I will always have their love and support.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Automatic programming (AP) is an approach that used by researchers to improve the quality of software development. Other approaches include structured [23], object-oriented [17], and extreme programming [10]. Object-oriented programming is a change in the design of the programming language, while structured and extreme programming are practices used by programmers. Some extreme programming practices include pair programming and small releases. There are two meanings associated with automatic programming. The first is concerned with programs that rewrite themselves in order to learn [83]. The second meaning is of systems that aid a user in constructing a program. This research is concerned with the latter. Three different approaches taken in the design of an automatic programming system are deductive synthesis, inductive logic programming, and genetic programming. Each of these methods is examined in Chapter 2.

Formal verification has been an important component of automatic programming since the first systems [32] [88]. Initially, the specifications were written in first-order predicate calculus, and a proof could be obtained using a theorem prover. Additionally, a program could be constructed either from the proof or from examples that were returned. The idea of using a proof to construct a program is known as *Proofs-as-Programs* [21].

However, construction of a program using a theorem prover limits the complexity of the specifications. To allow for more complex specifications, automatic programming systems began using *specification transformation* [52]. In this process, a specification is transformed until an executable program is obtained using a set of rules that at each stage preserve the correctness of the specification. Although automatic programming systems that utilize this process can still verify specifications via a theorem prover, the verification is not necessary in order to obtain an executable form of the specifications. As a result of being able to create

larger and more complex specifications, more attention was placed on the input language of the automatic programming systems. This work included creating higher level input languages that could use examples, subsets of natural language [33], or partial specifications when the complete specifications were not known [6].

As automatic programming systems began to be used in various projects, the importance of domain knowledge became apparent [8]. For example, an automatic programming system designed for creating software to drill for oil wells must have knowledge pertaining to this field. Domain knowledge is now included for many automatic programming systems [51] [24] that are designed for applications in a particular domain.

Despite the advances in automatic programming, concurrent computing has not become a focus of AP systems. With the recent introduction of multiple core processors, concurrency has become an important, and immediate, issue. This research is of an automatic programming system, CAPS, designed for the creation and formal verification of concurrent specifications.

The rest of the chapter is as follows: a statement of the problem is given in Section 1.1. A brief overview of the automatic programming system CAPS, developed in this research, is given in Section 1.2. The aims and contributions of the research are discussed in Section 1.3, and an outline of the rest of the dissertation is given in Section 1.4.

1.1 Statement of the Problem

Concurrent computing is concerned with tasks that operate in parallel. These tasks may come from one application, multiple applications on one machine, or multiple applications on multiple machines. With the recent shift in processor development to multiple core processors, concurrency has become one of the most important issues in software engineering. Concurrent computing is typically achieved through one of two methods: shared memory or message passing.

Shared memory refers to data shared across multiple processes. While multiple processes may access the shared memory, a locking mechanism is usually needed to ensure that memory is accessed by only one process during a write operation.

Message passing is the other method of concurrent computing. In this approach, messages are sent between different processes. The processes involved may reside on the same machine or could be on separate machines.

With the introduction of concurrency, applications become more difficult to develop and prove correct. The developer must now consider multiple tasks that are interacting in ways that may not have been foreseen. This can lead to problems that rarely appear but result in application errors.

Additionally, in many cases, the application must be shown to be provably correct. Formal verification is the process of proving, or disproving, that a specification is correct with regard to a set of stated requirements. The two primary methods of verifying specifications are theorem proving and model checking; however, there are a number of other formal verification methods.

Theorem proving, also known as *deductive inference*, verifies a specification that is typically given in the form of either propositional logic or first-order predicate calculus. Theorem proving of most specifications is NP-complete; therefore, specifications exist for which formal verification via a theorem prover is not a practical solution.

In *model checking*, the specifications are typically given using temporal logic. One of the issues of model checkers is the state space explosion problem in which the growth of the number of states becomes exponential. Thus, as with theorem provers, there are certain specifications for which verification through model checking is intractable.

Formal verification methods have been developed for both Petri and colored Petri nets. These methods involve the construction of occurrence and strongly connected component graphs (see Section 4.3). As with model checking, there exist colored Petri nets for which formal verification is intractable.

A number of systems and methods have been developed for concurrent computing [39] [27] [15] and formal verification [54] [30] [79]. However, few systems have been developed for both concurrent computing and formal verification. One such system is SMV (see §4.2.2). The focus of this research is to develop an automatic programming system for the creation and formal verification of concurrent specifications.

1.2 Concurrent Automatic Programming System

CAPS (Concurrent Automatic Programming System) is an automatic programming system developed for the creation, execution, and formal verification of concurrent specifications. A graphical representation of CAPS is shown in Figure 1.1 and explained below:

The central feature of CAPS is the transformation of concurrent specifications into colored Petri nets. This conversion is done automatically and retains the semantics of the specifications. Thus, if the specifications are correct, the colored Petri nets will be correct. This extends the work done by others in

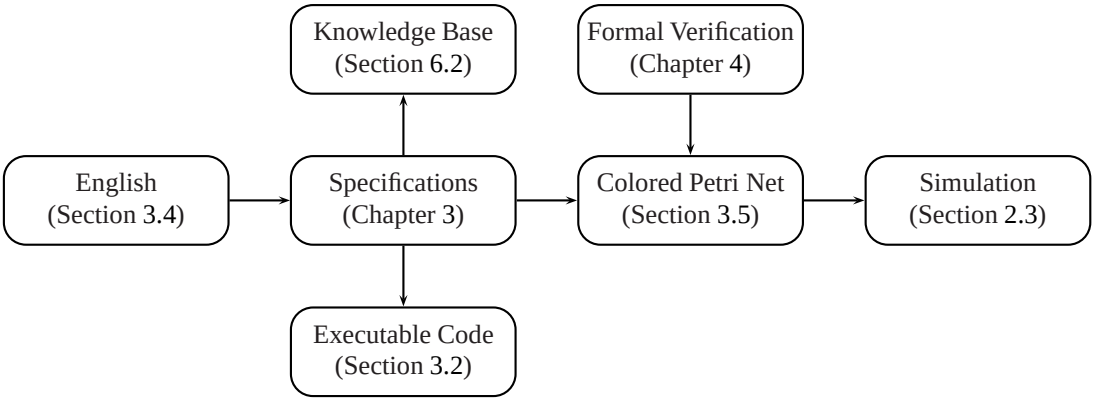

Figure 1.1: Graphical representation of CAPS.

software engineering [63] and is a new approach in automatic programming.

A new concurrent specification language, CAPSL, has been created. The specification language has been designed such that each structure and statement of the language can be converted automatically into a colored Petri net. Previous methods of converting code into Petri nets used existing languages and were only able to convert a subset of the language.

A goal of automatic programming systems is to improve the input language of the system to a higher level such that it can be used by people other than programmers. CAPSL is a very high-level language designed to hide the details of concurrency from the user. A natural language processing (NLP) system has also been developed to convert English phrases and sentences into specification structures and statements.

After the specifications have been converted into colored Petri nets, the specifications can be formally verified in which all possible states of the specification are examined. The verification is performed on the colored Petri nets. Since the semantics of the colored Petri nets are the same as the original specifications, if the colored Petri nets are correct, the specifications are correct. The implementation of the colored Petri nets is based on [42]; however, some modifications have been made to the colored Petri nets and their formal verification.

Specifications developed in an automatic programming system should either be executable or there should be an automatic transformation of the specifications into an executable form that preserves their correctness. A platform independent interpreter has been developed for the specification language of CAPS; additionally, a simulator has been written for the colored Petri net engine.

1.3 Aims and Contributions of the Research

This research is aimed at advancing the state of automatic programming. The specific area of focus is on concurrent computing. This is achieved through a new specification language and the use of colored Petri nets (see Section 1.2). The thesis statement of the research is the following:

> This research focuses on the development of an automatic programming system that aids a user in the creation, execution, and formal verification of concurrent specifications.

The primary contribution of this research is a concurrent automatic programming system. Current automatic programming systems are not designed for concurrent specifications and few systems are designed for both concurrent computing and formal verification. In order to achieve concurrency and formal verification, this research uses a new approach to automatic programming. The specifications are converted into colored Petri nets, which are ideal for modeling concurrency. Once in the form of CP-nets, the specifications can be formally verified.

The second contribution is the mapping from the specification language into colored Petri nets. This mapping is more extensive than previous work that has been done in converting programming languages to CP-nets [63]. A mapping to a colored Petri net exists for each structure and statement of the specification language. This was one of the primary design goals of CAPS and influenced both the development of the specification language as well as the custom colored Petri net engine.

The third contribution of the research is a new specification language designed for concurrent specifications. It is more difficult to create a concurrent application than a sequential application, and it is still unclear as to what method is best for the creation of concurrent applications. The specification language CAPSL can be used independently of CAPS and is a very high-level programming language designed to hide the details of concurrency from the user.

The fourth contribution includes modifications to colored Petri nets. The custom colored Petri net engine developed in this research is based on [42]. However, modifications have been made to hierarchical colored Petri nets along with the formal verification properties of the modified CP-nets. Additionally, modifications have been made to the formal verification properties of the colored Petri nets to focus on only a particular subset of the places and colors.

1.4 Outline of Dissertation

The remainder of the dissertation is composed of four parts: relevant background information is given in Chapter 2, the different components of CAPS are discussed in Chapters 3 and 4, case studies of CAPS are given in Chapter 5, and the conclusions of the dissertation are presented in Chapter 6.

Chapter 2 presents background information for different automatic programming (AP) systems as well as the components of CAPS. An AP system is usually one of three types: deductive synthesis, inductive logic programming, or genetic programming. CAPS is best described as a deductive synthesis system. The background information on the different components of CAPS includes colored Petri nets, specification languages, and natural language processing.

Chapter 3 presents CAPSL, the specification language of CAPS. CAPSL, is a very high-level programming language with a focus on concurrent specifications. The specification language was designed for two purposes: concurrency and automatic conversion to a colored Petri net. The natural language processing system of CAPS, which handles conversion of English into specifications, is discussed as is conversion of the specification language into a colored Petri net. For each structure and statement of CAPSL, there is a mapping to a colored Petri net. This allows for an automatic conversion of CAPSL code. Once in the form of colored Petri nets, the specifications can either be formally verified or a simulation of the specifications can be run.

Chapter 4 describes the formal verification properties of CAPS. Formal verification of CAPSL code is a combination of CAPSL statements and the formal verification properties of colored Petri nets. The formal verification of CAPS is designed to handle many of the details for the user.

Chapter 5 gives two case studies for CAPS. The first case study is of the producer-consumer problem. This problem was chosen for two reasons: the problem requires a concurrent solution and the producer-consumer problem is historically important with regard to Petri nets. The second case study is for the domination number of a tree T. This problem was chosen to illustrate the use of collections in the specification language of CAPS.

Chapter 6 contains the conclusions of the research. Included in the conclusions is a summary of the research, the contributions that have been made by CAPS, and areas of future work for the different components of CAPS.

Chapter 2

Background

This chapter presents background information relevant to CAPS (Concurrent Automatic Programming System). A literature review of automatic programming methodologies and systems is given Section 2.1. Section 2.2 contains an overview of specification languages. Colored Petri nets, the foundation of CAPS, are discussed in Section 2.3. Natural language processing is explained in Section 2.4, and a summary of the chapter is given in Section 2.5.

2.1 Automatic Programming

Computer scientists have realized that writing correct software is a fundamental problem. Many models have been proposed to improve the quality of software. Some of the traditional models of software development include the waterfall, spiral, and fourth generation model. In addition, programming methodologies have been developed such as structured, object-oriented, and extreme programming.

The most ambitious proposal for improving software quality is *automatic programming* (AP). There is some ambiguity regarding what constitutes an automatic programming system; but in general, the goal of an AP system is to raise the languages used by programmers to a higher level; however, the exact level, as well as other features such a system should have, are in dispute.

There are two meanings associated with automatic programming. The first approach is that of programs that can rewrite themselves in order to learn new information. This approach was mentioned by Turing in 1950 when describing the Turing test. "By observing its own behaviours it can modify its own programmes so as to achieve some purpose more effectively" [83]. The second common meaning of automatic program-

ming is that of systems that aid a person in constructing a program. This dissertation is concerned with this meaning.

In 1954, Backus and IBM presented a preliminary paper on Fortran, which at the time, was considered to be an automatic programming system [5]. Since that time, the idea of an AP system has evolved into more than just a compiler, beginning with the work of Friedberg [26]. Friedberg's work was an early, and less formalized version, of genetic programming; however, it was not until decades later that others would take an approach similar to Friedberg's in automatic programming.

Current automatic programming research falls into one of three areas: deductive synthesis (§2.1.1), inductive logic programming (§2.1.2), and genetic programming (§2.1.3). For each of these methods, relevant background information is presented, a history of the development of systems in the area is given, and a list of state-of-the-art systems is provided.

2.1.1 Deductive Synthesis

Deductive synthesis (DS) was the first significant automatic programming paradigm. In deductive synthesis, $P \vdash C$. In this formula, P is a set of *premises* that can constitute a specification and C is a logically derived *conclusion* or synthesized program. The *turnstile* $\vdash$ can be read as P yields C. The user is either given, or can derive, P. In most DS systems, P is a specification and C is a synthesized program.

Background information relevant to many deductive synthesis systems is presented below. Readers already familiar with logic, deductive inference, and specification transformation, these areas can skip to the history of deductive synthesis and the list of state-of-the-art systems.

2.1.1.1 Logic

Many of the initial deductive synthesis systems used logic for defining their specifications. The logic most commonly used by these systems was *first-order predicate calculus* (FOPC).

FOPC is an extension of propositional calculus and has sentences and terms. A sentence is a fact that is given. A term is an object. Additionally, FOPC includes connectives, quantifiers, constants, variables, predicates, functions, and the not ($\neg$) operator. The connectives of FOPC are implication ($\Rightarrow$), equivalence ($\Leftrightarrow$), and ($\wedge$), along with or ($\vee$). The quantifiers are the universal ($\forall$) and the the existential ($\exists$). An example

of FOPC is:

$$\forall x(Man(x) \Rightarrow Mortal(x)) \tag{2.1}$$

The equation states that all men are mortal. In the example, $Man(x)$ and $Mortal(x)$ are predicates and x is a variable. Now take the following statement:

$$Man(Socrates) \tag{2.2}$$

From the first implication, it can be reasoned that Socrates is mortal. For more information on FOPC, the reader may consult nearly any logic or introductory artificial intelligence book, such as [71].

2.1.1.2 Deductive Inference

Deductive inference, generally known as *theorem proving*, is used to verify a given specification. Often, the specification will be given in FOPC, but theorem provers for other logics have been developed. In deductive inference, $P \vdash C$. There are many inference rules in deductive inference. Here, only two rules are examined. The first rule is known as *modus ponens* and is defined as:

$$\frac{\begin{array}{c} A \vdash B \\ A \end{array}}{\therefore B} \tag{2.3}$$

The items $A \vdash B$ and A are the premises. If both of the premises are true, it can be concluded that B is true, the conclusion, must be true. The second rule is *modus tollens*, which is defined as:

$$\frac{\begin{array}{c} A \vdash B \\ \neg B \end{array}}{\therefore \neg A} \tag{2.4}$$

The premises are $A \vdash B$ and $\neg B$. If the premises are both true, one can conclude $\neg A$, the conclusion, is true as well. More information about the other rules of deductive inference can be found in [22] or any general logic text.

2.1.1.3 Specification Transformation

Specification transformation has been widely used in the field of deductive synthesis for converting a logical specification into an executable program. To transform a specification, a set of rules are applied that move a specification closer to an executable program and at the same time preserve correctness. Therefore, if the specification is initially correct, then a program obtained from a transformation of that specification will also be correct.

Many automatic programming systems use specification transformation because a program can usually be synthesized faster by transforming the specifications rather than by using a constructive theorem prover. However, just a transformation of the specification alone does not guarantee that the specification is correct, it only guarantees that the code is correct if the specification is as well. Therefore, in order to verify the specification, there still needs to be some way to check its correctness, such as using a theorem prover or model checker.

2.1.1.4 Deductive Synthesis History

There were two key developments in the field of artificial intelligence in the 1960's that allowed for the development of the field of deductive synthesis. The first of these developments was theorem proving. The second development was the work on question-answering systems.

A *question-answering system* is a program that when given a question checks to see if an answer is derivable from its knowledge base. A standard example posed to these systems is how can a monkey reach a banana if the banana is out of reach but a movable box is present [71]. The question-answering system should be able to provide the answer that the monkey moves the box under the banana, gets on the box, and then grabs the banana.

GPS The quintessential question-answering system was *GPS* (General Problem-Solver) [62]. GPS used *means-ends analysis* to answer questions posed by a user. Means-ends analysis is a search heuristic that uses both top-down and bottom-up methods in searching.

In 1963, Simon used GPS as a heuristic compiler, where compilation was considered a special case of problem solving [75]. GPS and Simon's heuristic compiler led to work on other question-answering systems that were capable of automatically synthesizing code. Many of these systems are mentioned below.

DEDUCOM Slagle's *DEDUCOM* (DEDUctive COMmunicator) [76] was based on GPS. Rather than means-ends analysis, DEDUCOM used deduction to answer questions in which the search method was depth-first, which searches to the end of a branch and then backtracks to search another branch. In addition, DEDUCOM was capable of synthesizing short programs in a manner similar to Simon's heuristic compiler. However, DEDUCOM performance was poor, and in some cases, the system would not find an answer even if one existed.

QA1 through QA3 After DEDUCOM, Green and Raphael introduced the question-answering systems *QA1* and *QA2* [34]. These systems represented a significant departure from previous question-answering systems in that they used a theorem prover to answer questions. Slagle's approach in DEDUCOM was deductive but used heuristics. Of QA1 and QA2, QA2 was the more important system in that it used resolution to answer questions.

In *QA3*, the successor to QA2, Green showed that the system could be used not only to answer questions but to synthesize programs [32]. QA3's input language was first-order predicate calculus, although a natural language system by Coles could be used [20]. The proof was constructive; therefore, if the question $(\exists x)STUDENT(x)$ was asked, then all x's satisfying the condition of being a student would be returned, and the examples that were returned could then be used to construct a program.

PROW At the the same time as QA3, Waldinger and Lee introduced *PROW* (PROgram Writing) [88]. PROW originated from the question-answering systems of Simon, Slagle, Green, and Raphael. However, PROW was slightly different from QA3. With PROW, a proof of the program's correctness was constructed, then the program was constructed from the proof. The idea of using a proof to construct a program is known as *Proofs-as-Programs* [21] and is still of interest today. An advantage of having a theorem prover create the program is that no verification of the program is necessary because it is correct by construction. Validation of the program still remains an issue because the specifications from which the proof was created may not have been what was intended.

DEDALUS The early deductive synthesis systems relied upon a theorem prover for program synthesis; however, due in part to the speed of computers at the time, as well as the computational complexity of theorem provers, the type of programs that could be synthesized was greatly limited. Therefore, many systems started to use specification transformation to convert a specification into an executable program. The first system to take this approach was Manna and Waldinger's *DEDALUS* (DEDuctive ALgorithm Ur-Synthesizer) [52].

Given a specification, DEDALUS transformed the specification until an executable program was obtained.

PSI One of the best known systems in the field of deductive synthesis was Green's *PSI* [33]. PSI was composed of several different expert systems. Of these systems, two were devoted to handling input. PSI could accept a large subset of natural language, program traces, or examples. Two of the systems used by PSI were PECOS and LIBRA, which are discussed below.

The synthesis system for PSI was known as *PECOS* [7]. Like DEDALUS, PECOS used specification transformation for code synthesis.

During the 1970's, the focus in deductive synthesis was on the correctness of the programs synthesized and on the type of programs that could be synthesized. As deductive synthesis matured and became able to synthesize larger programs, efficiency became a concern. Kant's system *LIBRA* was created specifically to deal with this issue [44]. LIBRA contained efficiency rules in addition to coding rules. Using the efficiency rules, possible programs were represented in a tree, and the most promising branches were further explored.

SAFE Another system that used specification transformation was Balzer's *SAFE* system [6]. Balzer pointed out that requiring complete specifications before a program can be synthesized was unrealistic, and he suggested an incremental development model. Through this method, a user could update specifications as necessary without requiring the system to verify or transform all of the previous specifications.

KIDS An advanced specification transformation system was Douglas Smith's *KIDS* (Kestrel Interactive Development System) [77]. Like other transformation systems, KIDS used specification morphisms to transform one set of specifications into another until a executable program was obtained, but KIDS also used schemas, such as divide-and-conquer, to help synthesize efficient programs.

ϕ_0 and ϕ_{NIX} Through the use of specification transformation, deductive synthesis systems could construct programs of greater complexity than ever before. But the efficiency of program construction is not the only issue involved with automatic programming systems. The first automatic programming systems were very general and the initial expectation was that they could be used to create a program for any domain. However, the importance of specific domain knowledge started to become apparent as the systems were used for solving problems that were no longer trivial. Barstow published results from his ϕ_0 and ϕ_{NIX} systems in which he focused on domain knowledge about oil wells [8]. His insight was that if the goal of automatic programming is to remove the programmer, then domain knowledge must be encoded into the AP system to allow users to

effectively produce programs. Figure 2.1 is an image adapted from Rich and Waters' [70] where they also discuss the necessity of domain knowledge.

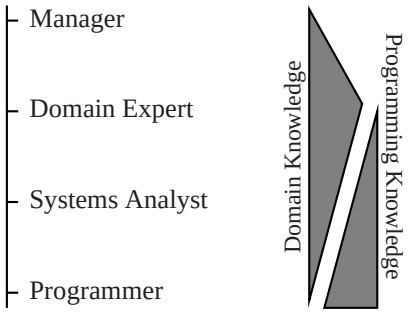

Figure 2.1: Comparison of domain knowledge to programming knowledge.

According to Figure 2.1, if the programmer is to be replaced by an automatic programming system, that system will require the domain knowledge of a systems analyst. Experiments from the 1960's with expert systems have shown that coding domain knowledge is a tedious and difficult task, and for many domains, it is still unclear how this information should be represented. However, work on data structures such as semantic networks and conceptual graphs has helped in the storage and retrieval of domain knowledge.

KASE Much of the work in deductive synthesis has focused on the synthesis of algorithms. More recently, the idea of *architectural design* has been proposed. When using an architectural design, the focus is not on algorithms, but on how the system is to operate. *KASE* (Knowledge Assisted Software Engineering) is a system based on this approach [11]. KASE is capable of creating tools that can be used for a certain problem that has the general architecture for which the tools were created.

2.1.1.5 Deductive Synthesis State-of-the-Art

Some of the leading edge research in deductive synthesis is at the Kestrel Institute and the Automated Software Engineering (ASE) group of NASA Ames. The Kestrel Institute was founded by Green for the purpose of producing high-quality software through formal means. Two projects of the Kestrel Institute, SpecWare and PlanWare, are mentioned below. The ASE group of NASA Ames focuses on creating reliable software for both end users and intelligent systems. Two of the ongoing projects of the ASE group, AutoBayes and Amphion, are also discussed below.

SpecWare Kestrel Institute's *SpecWare* is among the most advanced deductive synthesis systems [53]. SpecWare is a specification transformation system that is based on category theory and uses many concepts from Smith's KIDS. It is capable of producing code that is correct by construction and can synthesize programs that are several thousands of lines of code in length. The user focuses on writing a correct specification that is more abstract than a high-level program, and SpecWare then converts the specification into a program.

PlanWare *PlanWare*, also developed by the Kestrel Institute, is an extension to SpecWare [12]. While SpecWare is a general purpose deductive synthesis system, PlanWare focuses specifically on scheduling problems for domain experts who have very little experience programming. By utilizing domain information, PlanWare is able to automate even more of the work than is possible by using only SpecWare.

AutoBayes For some time, NASA has been interested in automated software engineering. One of their current projects is *AutoBayes* [24]. AutoBayes' domain is the production of data analysis programs – specifically those that deal with probability. The synthesis of the programs is directed by a *schema*, which is "a code template with associated semantic constraints which define and restrict the template's applicability" (p. 484). The goal of the project is to produce C/C++ code that contains automatically generated comments.

Amphion A long running project of NASA is *Amphion* [51]. Like other deductive synthesis systems, Amphion takes formal specifications and creates a correct program from those specifications. However, unlike other systems, users interact with Amphion through a GUI that is determined by domain knowledge. Most of the work has gone into *Amphion/NAIF*, whose domain is concerned with the solar system. A related project is known as *Meta-Amphion*, which incorporates a theorem prover for proving correctness.

2.1.2 Inductive Logic Programming

Inductive logic programming (ILP), represents the intersection of inductive inference and logic programming. ILP begins with background knowledge, a set of positive examples, and a set of negative examples. The examples, combined with the background knowledge, are used to create the specifications. The idea behind ILP is that a user might be able to give very good examples of what a specification should do (or should not do) but may have a very difficult time writing the specification itself.

Background information on inductive inference and logic programming is provided below. For the reader familiar with these topics, these areas can be skipped to a history of the field of ILP and a list of

state-of-the-art systems.

2.1.2.1 Inductive Inference

In this context, *inductive inference* is a form of machine learning that uses examples to teach the learner. The examples may consist of only positive examples or both positive and negative examples. The explanation below is derived from [28] and [25].

In inductive inference, the learner is given background knowledge B and a set of examples E where $B \nvdash E$. It is assumed that the learner cannot derive the examples E from the background knowledge B. A hypothesis, or set of hypotheses, H is sought such that $B \cup H \vdash E$. Therefore, given the background knowledge B and a hypothesis, or set of hypotheses, H, all of the examples E should be derivable. $B \cup E \nvdash \neg H$ is used to test the consistency of H with respect to B and E.

Once a hypothesis H has been generated, it can be used for either explaining examples or predicting outcomes. For more information on inductive inference and other forms of machine learning, the reader should consult [55].

2.1.2.2 Logic Programming

Logic programming was developed in order to allow programmers to base their world-view on first-order predicate calculus. One of the best known logic programming languages is Prolog [19], which is a declarative language. In a *declarative language*, the user tells the interpreter/compiler what to do, rather than how to do it.

A fact in Prolog would look like $student(jill)$. If the user then asks $student(X)$, the variable X will be unified with all possible answers. In this case, X would have the value of $jill$. For additional information on Prolog, Bratko presents how the language can be used to solve common artificial intelligence problems in [13].

2.1.2.3 Inductive Logic Programming History

Inductive logic programming is the intersection of inductive inference (one type of machine learning) and logic programming. First, a brief examination of the development of the fields of inductive inference and logic programming is given. Then there will be a discussion on some of the more influential ILP systems.

The field of machine learning was first seriously considered by Alan Turing in [82] and [83] with the first in-depth study of inductive inference in [78]. Solomonoff's paper is considered to be one of the

foundations of inductive logic programming.

Green's paper [31] is often credited as the inspiration of logic programming even though it was later shown that Green's method was not feasible because of combinatorial explosion. The area of logic programming as was developed by Colmerauer and Roussel working with Kowalksi. In 1972, Prolog (Programmation en Logique) was created. At first, logic programming was inefficient; however, work by Warren in compiling Prolog [89] was able to make Prolog as efficient as compiled Lisp.

The convergence of early ILP systems was made possible because of theoretical results established by Gold in [29] concerning what classes of languages can be identified in the limit for different learnability models. If a language can be identified in the limit, then after a finite number of examples, the learning algorithm will always answer correctly.

Another important result for ILP was independently discovered by Plotkin in [66] and [67] and Reynolds in [69]. Plotkin describes the least generalization. This "is a generalization which is less general than any other such generalization" (p. 153) [66]. In [67], Plotkin developed the *rlgg* (relative least general generalization). The rlgg is the learning model that was used in many of the initial ILP systems and is still in use in some of the current systems.

In the 1980's, Valiant proposed a new learning model [85] [65]. The new model was PAC (probably approximately correct) learning. When using *PAC learning*, identification in the limit is not guaranteed as with Gold's model, but as the PAC name implies, there is a high probability that the model will correctly identify an unknown language. The efficiency of PAC learning, combined with the probability that it will usually arrive at the correct answer, has resulted in it becoming the most used learning model in modern ILP systems.

It was not until 1991 that the term inductive logic programming was first used. Up until this time, the area was defined by using logic programming for machine learning. Muggleton is responsible for naming the field in [57].

MIS The first ILP system was developed by Shapiro and was named *MIS* (Model Inference System) [74]. The theoretical foundation of MIS was based upon the work of Gold and Plotkin. Shapiro was able to prove that his system would eventually create a correct program if given enough examples. The system would start with an empty theory and then add hypotheses. Using resolution, MIS could determine if the theory was too general. If the theory was too specific, then a new hypothesis was generated and added to the theory.

Marvin At roughly the same time Shapiro was working on MIS, Sammut was working on *Marvin* [72] [73]. Marvin started with a trial T_0 and would use generalization until a trial T_n was reached such that any more generalization would result in an inconsistency. This form of generalization was used in several later ILP systems. Marvin also used specialization that influenced some later systems. It took some time though (until 1986) before Sammut realized that Marvin was, in effect, an ILP system.

DUCE and CIGOL Another ILP system was *DUCE* that was created by Muggleton [56]. DUCE had six operators for transformation that were based on propositional calculus. Muggleton realized that this could be strengthened, and in 1988, Muggleton and Buntine developed *CIGOL* (logic spelled backwards) [60]. CIGOL built upon ideas in the DUCE system and used *inverse resolution*. The name inverse resolution is used because it deals with induction, whereas resolution deals with deduction. The operators of DUCE were also inverse resolution but at a propositional level. Inverse resolution soon became popular and helped to stimulate the growing field of ILP.

FOIL There were systems though that did not use inverse resolution. One such system was Quinlan's *FOIL* (First-Order Inductive Learning) [68]. FOIL used search heuristics to learn Horn clauses. While FOIL was more efficient than many of the earlier systems, using the heuristics meant that some positive examples might not be covered and some negative examples might.

GOLEM At this point, Valiant's PAC learning was being used to a large extent in ILP. However, some people preferred to use Plotkin's rlgg; however, the problem with rlgg is that it may contain an infinite number of literals. Muggleton and Feng came up with the idea of *h-easy rlgg* and demonstrated it in *GOLEM* [61]. The advantage of h-easy rlgg is that the clauses are of finite length.

SYNAPSE *SYNAPSE* (SYNthesis of logic Algorithms from PropertieS and Examples) is a system that is based off of the work of Pierre Flener [25] and straddles the fields of deductive synthesis and inductive logic programming. SYNAPSE uses both deductive and inductive synthesis to create logic programs.

2.1.2.4 Inductive Logic Programming State-of-the-Art

Below, some of the more recent innovations in inductive logic programming are given. The innovations include incorporating probability distributions, natural language processing, and the continuing improvement of older ILP systems.

Stochastic Logic Programs Since the early 1990's, there have been several papers written on adding probability distributions to ILP. This work has come to be known as *SLP* (Stochastic Logic Programs) [58] [59]. Many of today's problems are either statistical in nature or are most easily represented using probability. SLP allows for these problems to be more readily solved using ILP.

CHILL Natural language processing is still a difficult problem and is one of the concerns of this dissertation (see Chapter 3). Work in ILP has been introduced to help automate the learning of a natural language processing system. Zelle and Mooney's *CHILL* (Constructive Heuristics Induction for Language Learning) program has done well in tests of natural language processing when compared with statistical methods that currently give the best results [91].

Other Systems A number of older ILP systems have also been upgraded with new features. Two of these are LINUS and FOIL.

Lavrac and Flach have extended LINUS so that it can handle non-determinate DHDB clauses [49]. The extension to LINUS increases its ability to learn, so it is better at classification and concept learning. However, a shortcoming of LINUS is that it cannot synthesize programs.

Quinlan's FOIL has been continually improving and is now at version 6 offering more features, and running faster, than before.

2.1.3 Genetic Programming

Genetic programming, GP, is a form of evolutionary computation that is based upon genetic algorithms. Evolutionary computation generally involves a population of possible solutions that evolve over time to better solve a particular problem. The mechanisms of evolution are loosely based upon biological systems. Two other evolutionary computation methods are *evolutionary programming* and *Tierra*. Both of these have similarities to genetic programming. A discussion of genetic algorithms is given along with a brief history and some state-of-the-art work in genetic programming.

2.1.3.1 Genetic Algorithms

Genetic algorithms (GA) were created by Holland in [40]. Initially, the artificial intelligence community paid little attention to Holland's work. It was not until many years later that interest in genetic algorithms, and other evolutionary computation methods, surged. Since that time, the use of genetic algorithms

has become widespread. Traditional searches like depth-first and breadth-first work well on small search spaces, but when the search space becomes too large, traditional methods of search become intractable and other methods must be used.

It has been shown that genetic algorithms perform very well on a large range of problems. GAs were inspired from how it is thought DNA (deoxy-ribonucleic acid) operates. A bit vector corresponds to the DNA chain, and there are crossover and mutation operations on this vector. The crossover operator takes information from the bit vectors of two parents and produces a child. The mutation operator changes a single point in the vector. Crossover is the operator primarily used, but mutation cannot be overlooked because it helps to keep the algorithm from getting stuck at a local maxima.

For genetic algorithms to work, there must be a group of individuals. The size of this group is largely dependent upon the problem. Which individuals survive to the next generation, and reproduce offspring, is based upon their *fitness*. The fitness is determined by how well the problem is solved by the individual.

2.1.3.2 Genetic Programming History

The general idea behind genetic programming can be traced back to Friedberg [26]. While Friedberg did not have the idea of using a method such as a genetic algorithm to produce a program, he did talk about random changes in a program in order to produce a new program. In practice, this would not be a very good approach because of the huge number of programs that might be generated before an appropriate one was encountered. However, if there was something to guide the search, like the fitness function of genetic algorithms, then this becomes a possible method of automatic programming.

The field of genetic programming began with Koza's book [48]. There were other writings about using genetic algorithms to create programs before Koza's work; however, it was Koza that defined the field of genetic programming. Since that time, well over 3,000 papers have been written on the field of genetic programming [47].

Genetic programming, as envisioned by Koza, takes a set of random programs and evolves them over time to select the ones that best fit the desired output. Thus the fitness function is seeing how well the program works. Koza dealt mainly with trees, and while they remain the dominant representation, lists and graphs can also be used. The tree structure for GP is like the bit vector for GAs with crossover and mutation operators for the evolution of programs.

Introns are also a consideration with GP. Introns are code that do not affect the output of the program. They are the equivalent of junk DNA. At this point, it is unclear if introns actually help in the generation of a

program or hinder it. It has been observed that close to the completion of the creation of a program, introns experience exponential growth. While this initially would seem to be a negative effect, some believe that the introns help to preserve the best parts of the program. After a program has been produced, the introns can be removed without any side effects.

Both genetic algorithms and genetic programming must deal with parameters. One parameter is the size of the population. For smaller problems, a population size might be 1,000, and for difficult problems, a population could be in the millions. Another issue is how many generations should be run. The heuristic for this is that 50 or less is usually appropriate. The crossover to mutation ratio must also be decided. Normally, the mutation operator will have a small probability, and crossover will be the dominant operation. Another issue is decided the size of the largest program that is allowed. With an exponential growth in introns, programs could quickly cause a computer to run out of memory if the population size was large enough. For many of these parameters, it is not clear what the best values should be until some values are tested on a specific problem.

2.1.3.3 Genetic Programming State-of-the-Art

The most significant recent advancement in genetic programming is the use of clusters to help solve difficult problems.

Human Competitive Results The home page for the field of genetic programming [47] maintains a list of instances in which a genetic programming system has developed a program that is competitive to one produced by a person. At the time of this writing, there were 36 such instances.

Parallel Genetic Programming Even though genetic programming is designed to arrive at a solution in a short period of time, problems of significant complexity can be infeasible to solve using a genetic programming algorithm on a conventional computer. There has been recent work in parallel genetic programming to increase the amount of computational power available to a genetic programming system. Genetic Programming Inc. has set up 1,000 computers in a Beowulf type cluster. They also have a number of smaller Beowulf type clusters as well. More information about Genetic Programming Inc., and their work in parallel genetic programming, can be found at [47].

2.2 Specification Languages

The specification language of CAPS serves three purposes: representation, execution, and formal verification through transformation of the specifications. In §2.2.1, three different classes of specification representations are examined. Various systems and methods that have been used to design and implement concurrent specifications are discussed in §2.2.2.

2.2.1 Representations of Specifications

A variety of methods have been used to represent specifications. In this section, three classes of representations are examined: high-level languages, very high-level languages, and formal languages.

2.2.1.1 High-Level Languages

High-level languages (HLLs), such as C and Java, are more suited for implementing specifications than efficiently describing them. This is due to the level of detail necessary in any high-level language. In addition to the details of the software specification, the implementer of a specification in one of these languages must be concerned with syntax and semantics. Advantages to using a HLL for representing specifications are that the specification becomes executable and languages such as C and Java are widely used; therefore, it is easier to share the specification with others.

2.2.1.2 Very High-Level Languages

Very high-level languages (VHLLs) are an abstraction of high-level languages. The minimum set of attributes that define a VHLL are dynamic typing and memory management [1]. In addition, many very high-level languages will have other features. For example, the specification language CAPS is a VHLL and focuses on reducing the complexity of multi-threading. An approach taken by many very high-level languages is to automatically convert the VHLL code representing the specifications into executable code. This is known as executable specifications [4] [87].

The input language of any very high-level language will more closely resemble a high-level language than a generic specification (such as a UML specification). Despite this, the programmer is able to work at a more abstract level than is possible with high-level languages; therefore, more focus is placed on the specification rather than the implementation language. A disadvantage of this approach is that there will be

a loss in performance. For this reason, many specifications are prototyped in a very high-level language, and once the specification is deemed correct, the specification is converted into a high-level language such as C.

2.2.1.3 Formal Languages

Both high-level and very high-level languages are in fact formal languages; however, here formal languages indicate mathematical formal languages. Two formal languages that are often used to describe specifications are first-order predicate calculus (FOPC) and temporal logic. FOPC can be verified using a theorem prover [30] and temporal logic can be verified using a model checker [16]. While these languages can describe software specifications, they were not designed to do so. Therefore, using FOPC and temporal logic in this manner can be cumbersome. This is the reason that other formal languages such as Z [90] have been devised.

The problem in using a formal language such as Z to describe the specifications is that in many cases the actual product must also be created. In the case of software, the product is a computer program. While the specifications described in Z may be correct, there is no guarantee that their translation by programmers will remain correct. Ideally, a specification language should be capable of proving the correctness of the specifications and either executing the specifications or transforming the them until they are executable [52].

2.2.2 Concurrent Specifications

In the 1960's, C. A. Petri developed Petri nets, which allowed for concurrent execution [64]. This development led to greater interest in the area of concurrent computing. Below, some systems and methods designed for the implementation and representation of concurrent specifications are discussed.

CSP/Occam CSP (Communicating Sequential Processes) is a theoretical description for how processes should interact. CSP was developed by C. A. R. Hoare in [38] and later refined by Hoare, Brookes, and Roscoe in [39]. Processes interact with one another in CSP via message-passing. The most practical application of CSP was the development of the parallel programming language *occam* by INMOS Ltd. Since that time, a few other systems such as CSP++, JCSP, and Limbo have implemented ideas from CSP.

Linda Gelernter and Carriero developed an extension to other languages in order to make them parallel [27]. The extension is known as *Linda* and has four operations: *in*, *out*, *rd*, and *eval*. Messages are passed as tuples into *tuple space*. Tuple space is a global area of memory containing tuples. Every process can access

the tuple space and a tuple will remain there until it is removed by a process. There exist Linda extensions for several languages including C and Java.

UNITY UNITY was developed by Chandy and Misra in [15] for concurrent computing. UNITY is concerned with correctness and uses a subset of temporal logic for this purpose. In a concurrent program, there are issues such as liveness and safety for which temporal logic can be used to create proofs. A correct UNITY program will eventually reach a fixpoint.

MPI The basis for MPI (Message Passing Interface) was initially developed in April of 1992 at the Workshop on Standards for Message Passing in a Distributed Memory Environment. MPI is used for concurrent computing on distributed systems and can best be described as a message-passing library specification. The specification describes how a MPI library should be created and how programs should interact with the library. Libraries for MPI have been developed on many different platforms and for various programming languages such as C, Fortran, and Ada.

SMV The SMV system is capable of modeling concurrent specifications and consists of an input language and a model checker. The input language is based upon CTL (Computation Tree Logic), a temporal logic. With the input language being based upon CTL, verification can be done for deadlock freedom, liveness, fairness, and safety [54].

POSIX Threads POSIX (Portable Operating System Interface for uniX) threads, also known as *pthreads*, were designed to be portable across different hardware, different Unix-based operating systems, and different implementations of programming languages. Many applications, including the specification language of CAPS, use POSIX threads to achieve concurrent execution. Examples of POSIX threads can be seen in the C implementation of the producer-consumer problem (see Chapter 5).

2.3 Colored Petri Nets

Colored Petri nets (CP-nets) have become a popular method for modeling concurrent specifications. Additionally, formal verification properties (see Section 4.3) have been developed for colored Petri nets. For these reasons, CAPS has been designed to be based upon CP-nets. In §2.3.1, a short history of colored Petri

nets is presented. §2.3.2 contains a discussion of the different elements of a colored Petri net, and simulation of a CP-net is explained in §2.3.3.

2.3.1 History

Colored Petri nets were developed by Kurt Jensen [41] as an extension of the Petri nets of C. A. Petri [64]. Other popular extensions of Petri nets include Condition/Event nets and Place/Transition nets. Initially, extensions were made to Petri nets in order to increase their power. The early Petri nets were not Turing-complete and were therefore unable to model certain systems such as the producer-consumer problem discussed in Chapter 5. The addition of inhibitor arcs to Place/Transition nets made Petri nets Turing-complete [35] thus allowing them to model any system that can be modeled using a Turing machine. An inhibitor arc is used to indicate that a place must be empty for a transition to be enabled.

The purpose of the colored Petri net extension is to allow for a place to contain distinguishable tokens. In Petri nets, representing individual tokens must be done in an indirect manner and this results in a large increase in the number of places, transitions, and arcs of the net. CP-nets allow for this information to be represented more efficiently, i.e. using fewer places, transitions, and arcs.

2.3.2 Elements of the Colored Petri Net

Colored Petri nets are an extension of Petri nets to allow distinguishable tokens. A Petri net is represented by the five-tuple (P, T, A, W, M_0) in which P is the set of places, T is the set of transitions, and A is the set of directed arcs where $A \subseteq (P \times T) \cup (T \times P)$. An arc can start at a place and go to a transition, or an arc can start at a transition and go to a place. W represents the weight of each arc $a \in A$ and is defined as $W : A \rightarrow \mathbb{N}$. The weight of the arc determines the number of tokens moved to, or from, a place during the firing of a transition. If the arc is from a place to a transition, tokens are removed; otherwise, the arc is from a transition to a place and tokens are added. M_0 represents the initial marking and is defined as $M_0 : P \rightarrow \mathbb{N}$. The initial marking specifies the number of tokens contained by each place $p \in P$.

The colored Petri nets used in CAPS are represented by the seven-tuple $(\Sigma, P, T, A, C, E, M_0)$. Sets P, T, A, and M_0 have the same definition as in traditional Petri nets. The set Σ represents the colors of the net. Σ is generally a subset of all possible colors. In a CP-net, a color has both type and value (e.g. an integer with value 7). These colors are used in the place of tokens and can then be distinguished from one another. C represents the colors of a particular place $p \in P$ and is defined as $C : P \rightarrow \Sigma$. E is the set of arc expressions

for a colored Petri net. An arc expression may be a color, variable (able to bind to any color in a place $p \in P$), a function, an if-then-else expression, or a case expression.

To illustrate these concepts, a CP-net of a distributed database from [42] is shown in Figure 2.2. In order to make the CP-net appearance easier to read, some details have been left out such as the types of colors and the definition of the *Mes* function.

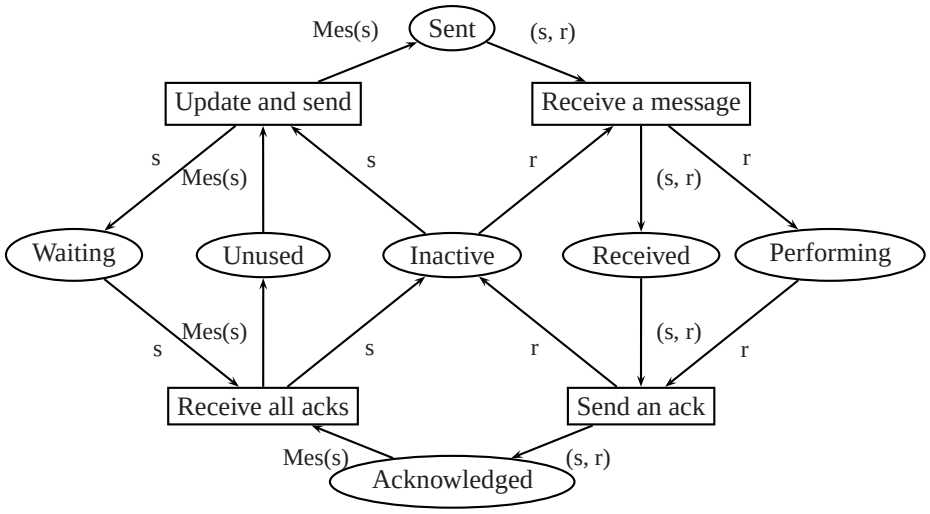

Figure 2.2: A colored Petri net example of a distributed database.

2.3.2.1 Places

A place in a CP-net, as in a Petri net, holds tokens. The places in Figure 2.2 are represented by ovals. This representation is consistent with how places are also usually shown in Petri nets. In both CP-nets and Petri nets, a marking of the net is determined by the collection of tokens over all of the places in the net. Since tokens can be distinguished in a CP-net, a marking is composed of not only the number of tokens a place contains but also the type of tokens.

A CP-net will contain a list of declarations. The declarations are color sets of the CP-net. Every place in the CP-net must have as its type one of the declarations. A declaration can be general – such as the set of all integers or strings, or a declaration can be specific, wherein the valid ranges over the color sets are limited.

2.3.2.2 Colors

The CP-nets of CAPS contain a subset of the color types of the CPN ML language [41]. Jensen's colored Petri nets use an extension of standard ML. The simple color types are unit, boolean, integer, real, string, and character. The only compound color type is product.

A unit color can have only one value. Boolean is an integer in which 0 represents False and everything else represents True. The integer color is represented as a 4 byte integer and upper and lower bounds can be set on an integer color for the range of integers it may represent. Real is an 8 byte float that can have upper and lower bounds. The string color is an array of characters. Bounds on the string color include upper and lower bounds for the size of the string as well as for the characters that can be contained within the string. Product is a combination of other colors which might be either simple or compound.

2.3.2.3 Arcs

Arcs in Petri nets are used to determine the number of tokens to be removed from, or added to, a place. Arcs in colored Petri nets serve the same purpose but provide more control. This is achieved through arc expressions. Figure 2.2 contains several different arc expressions. The two classes of expressions in the figure are constants (*s*, *r*, and the product *(s, r)*) as well as functions (*Mes(s)*).

The classes of arc expressions provided in CAPS are constants, variables, booleans, if-then-else, case, and functions. These expressions are discussed below.

A constant expression will either remove a token of that color from the place (when the arc is from the place to a transition and the transition fires), or it will add a token of that color to the place (when the arc is from the transition to the place and the transition fires).

Variable expressions are used to allow a variable to represent a set of colors. When the variable is bound, then the variable acts as a constant expression does. The variable is bound by the system determining what transitions are enabled over different possible variable bindings and then randomly selecting one of the bindings that results in an enabled transition.

A boolean expression involves comparison between a variable and a constant. For instance, the expression *[X == 2]* in which *X* is a variable is a boolean expression. The expression is determined to be true or false based upon a particular binding of *X*.

The if-then-else and case expressions are equivalent to their high-level language counterparts. The if-then-else expression takes three other expressions. The first expression is a boolean expression. The

second and third expressions are used if the first expression is found to be true or false respectively. The case expression takes a variable expression as its argument. There are then 0 to n expressions used in the evaluation of the variable expression.

A function is an expression that is included in the declarations. The purpose of the function is to allow commonly used expressions to be located in one location and also to help reduce the amount of information presented with a CP-net.

2.3.2.4 Transitions

Transitions in Figure 2.2, and with CP-nets in general, appear as rectangles while transitions in Petri nets often appear as either rectangles or lines. A transition is responsible for moving tokens from the input places of the transition to the output places. An input place is a place from which an arc goes to the transition. An output place is a place to which the arc goes from the transition.

The nondeterminism of a colored Petri net arises from the firing of transitions. At a given step of a CP-net simulation, any enabled transition may fire, but only one transition may fire at a time. For a transition to be enabled, there must be a binding that will satisfy all of the arc expressions from the input places.

2.3.3 Simulation

One of the advantages of using CP-nets as an intermediate representation of the specifications is that the CP-net can be simulated. This simulation allows the user to interact with the specification in a visual manner. At each step of the simulation, it must be determined which transitions are enabled. If multiple transitions are enabled, then a randomly selected transition is fired. CAPS uses the C drand48 random number generator for selecting a transition at random. If no transitions are enabled, the simulation will end.

CAPS has 5 controls for controlling the simulation of a CP-net. These controls are (i) next step, (ii) previous step, (iii) run n next steps per second, (iv) run n previous steps per second, and (v) stop. If no transition is enabled, there is no next step, and if no steps have yet to be taken, then there is no previous step.

Graphical dialogs are built into CAPS that allow for places and transitions to be examined and edited. This includes dialogs for adding/deleting places, transitions, and arcs to/from the model, a dialog for adding/deleting color types to/from the model, adding/deleting colors to/from places, changing the type of color a place might contain, and changing arc expressions. In addition, the locations and names of places and transitions can be changed.

2.4 Natural Language Processing

Natural language processing (NLP) has been used in several automatic programming (AP) systems. The first AP system to incorporate NLP was QA3 [32] [20]. Natural language processing is one method of human-computer interaction (HCI) and is a broad discipline that includes text-to-speech (TTS) synthesis, automatic speech recognition (ASR), and natural language understanding (NLU).

An NLU system parses and analyzes natural language phrases and sentences while trying to create a machine representation of the natural language that retains the semantics. This process is made difficult by the ambiguity of natural language. For most NLU systems, the first step of analysis involves tagging the words of the phrase or sentence with the correct parts of speech.

CAPS incorporates a natural language processing system in order to allow the user to focus more on the specifications and less on the syntax of the specification language. In §2.4.1, part-of-speech tagging is discussed.

2.4.1 Part-of-Speech Tagger

Part-of-speech (POS) tagging is concerned with automatically tagging each word of a given sentence with its correct part-of-speech. Three different methods for POS tagging are examined: rule-based tagging, stochastic part-of-speech tagging, and transformation-based tagging, which is a combination of the previous methods.

2.4.1.1 Rule-Based Part-of-Speech Tagger

Rule-based tagging was the first method developed to automatically tag parts-of-speech [36]. This method relies upon a set of rules as well as a dictionary that contains words along with their possible parts-of-speech. Rule-based tagging is broken up into two stages. In the first stage, a word is looked up in the dictionary and all possible parts-of-speech for that word are returned. The set of rules are then used in the second stage to remove ambiguity so that each word is assigned only one part-of-speech. The number of rules needed for the second stage can reach over 1,000 in more complicated systems [86].

2.4.1.2 Stochastic Part-of-Speech Tagger

For decades, probability theory has been used to categorize the words of English [80] using tags such as those in the Penn Treebank tagset. The ambiguity of English, combined with the limits of any

training corpus, has resulted in the widespread use of *stochastic part-of-speech tagging* methods. In general, stochastic part-of-speech tagging can be split into two different methods. The difference in these methods involves the context of the word to be tagged.

The simpler method avoids context altogether. First, a training corpus is used to calculate several probabilities. Then, when a word to be tagged is encountered, such as *book*, the parts-of-speech of the word are compared to one another in order to determine which part-of-speech has the greatest probability of occurrence. The computation of the probability of a word given a particular part-of-speech is done using Bayes' theorem:

$$P(tag|word) = \frac{P(word|tag) * P(tag)}{P(word)} \tag{2.5}$$

For example, the probability that an occurrence of the word *book* is a singular or mass noun (NN) can be calculated using the following equation:

$$P(NN|book) = \frac{P(book|NN) * P(NN)}{P(book)} \tag{2.6}$$

The probabilities of the equation on the right hand side can all be calculated from a training corpus. $P(NN)$ is the probability of a tag being a noun. If there is a corpus of 100,000 words, of which 30,000 of the words are nouns, then the probability that a given word will be a noun is:

$$P(NN) = \frac{30,000}{100,000} = 0.3 \tag{2.7}$$

$P(book)$ is the probability of a given word being *book*. If out of the 100,000 words of the training corpus, the word *book* appears 500 times, then $P(book)$ is:

$$P(book) = \frac{500}{100,000} = 0.005 \tag{2.8}$$

The expression $P(book|NN)$ can be read as: given that the word is a noun, what is the likelihood that the noun is the word *book*? This probability, also known as conditional probability, simplifies to:

$$P(book|NN) = \frac{P(book \cap NN)}{P(NN)} \tag{2.9}$$

An expression $P(book \cap NN)$ can be calculated by determining how many times the word *book* is

a noun in the training corpus; therefore, if *book* appears as a noun 350 times:

$$P(book \cap NN) = \frac{350}{100,000} = 0.0035 \tag{2.10}$$

All of the unknowns have been calculated and Equation 2.6 becomes:

$$P(NN|book) = \frac{P(book|NN) * P(NN)}{P(book)} = \frac{\frac{P(book \cap NN)}{P(NN)} * 0.3}{0.005} = \frac{\frac{0.0035}{0.3} * 0.3}{0.005} = 0.7 \tag{2.11}$$

Therefore, given an occurrence of the word *book*, the probability that the word is a noun is 0.7. If the probability was less than or equal to 0.5, the other probabilities would need to be calculated, such as $P(VB|book)$. Once all of the probabilities have been calculated, the part-of-speech with the greatest probability is selected. Using this method, approximately 90 percent of the words are tagged with the correct part-of-speech.

The second stochastic part-of-speech tagging method considers the context of a word. This method of tagging if often called *hidden Markov model (HMM) tagging* because of its use of a HMM to determine the correct part-of-speech for the given word. Continuing with the prior example of determining if the word *book* is a noun, calculate:

$$P(book|NN) * P(NN|previous - n - tags) \tag{2.12}$$

Calculation using the tag *NN* would then be compared with all calculations for other possible tags of the word *book*. The tag that resulted in the largest probability would then be selected as the most probable.

The probability of the previous n tags given a particular tag, such as *NN*, is calculated from a corpus. In practice, the number of previous tags considered in a stochastic POS tagger is usually one (*bigram-HMM tagging*), though there are some *trigram-HMM taggers*. The accuracy of this method of tagging is typically around 95 percent.

2.4.1.3 Transformation-Based Tagger

CAPS uses a *transformation-based tagger* [14] to tag the words of a given sentence. As the tagger parses the sentence, the words are assigned part-of-speech tags from the Penn Treebank tagset. The transformation-based tagging algorithm can be separated into two parts. The first part involves stochastic

tagging that does not consider context. The second part uses rules to refine the tags in a particular context.

Stochastic tagging involves the use of a dictionary to find the most probable tag for a given word. The tags are those that were encountered for a particular word in a training corpus (Brown and WSJ). For example, the word *founding* has the following tags in CAPS dictionary:

- *NN*: Noun, Singular or Mass
- *VBG*: Verb, Gerund or Present Participle
- *JJ*: Adjective

The tag that appears first, *NN*, is the most probable tag. A complete list of tags, and their meanings, is given in Table 3.12. The other tags appear in no particular order. That is, it cannot be determined from their order if *VBG* is more likely than *JJ*. One can only know that in the training corpus that was used, the word *founding* appeared as a *NN*, *VBG*, and *JJ* with the *NN* form the most prominent. Let us now examine using stochastic tagging to tag all of the words of the phrase:

The founding fathers of the country ... (2.13)

Each word of the phrase is assigned its most probable tag as given in the dictionary. This method does not consider the context in which the word was used. If the probability of a word being a *NN* is 0.51 and the probability of it as a *VB* is 0.49, every time at this stage the word will be given the tag *NN*. As mentioned earlier, this method of tagging results in approximately 90 percent accuracy. The complete tagging of the phrase using this method is:

The/DT founding/NN fathers/NNS of/IN the/DT country/NN ... (2.14)

In the phrase above, all of the words except *founding* have been given the correct tag. The most probable tag for *founding* is *NN*, but in the phrase, *founding* is used as an adjective, and the tag should therefore be *JJ*. The wrong tag for *founding* was given because the context was not taken into account. To deal with context, transformation-based tagging uses a set of contextual rules to further refine the initial tags. The tagger contains hundreds of contextual rules that are learned from a corpus (Brown and WSJ). One of these is a rule that will change a word that was tagged as a *NN* to a *JJ* when the next tag is a *NNS* (plural

noun). Applying this rule to the tagged phrase above gives the following:

$$\text{The}/DT\ \text{founding}/JJ\ \text{fathers}/NNS\ \text{of}/IN\ \text{the}/DT\ \text{country}/NN... \tag{2.15}$$

The phrase is now correctly tagged. In this case, only one rule had to be applied to the phrase, but in some cases, a word may have its tag changed several times. Brill's tagger often achieves accuracy around 95 to 97 percent.

2.5 Summary

This chapter has presented background information relevant to CAPS (Concurrent Automatic Programming System). The information has covered a review of automatic programming systems as well as overviews of specification languages, colored Petri nets, and natural language processing. Three approaches to automatic programming were discussed: deductive synthesis, inductive logic programming, and genetic programming. CAPS is best described as deductive synthesis system. In Chapter 3, the components of CAPS, starting with the the specification language CAPSL, are discussed.

Chapter 3

Specification Language

The focus of this chapter is on CAPSL, the specification language of CAPS. CAPSL is used to represent the specifications of a system in code. An overview of the specification language within CAPS is shown in Figure 3.1.

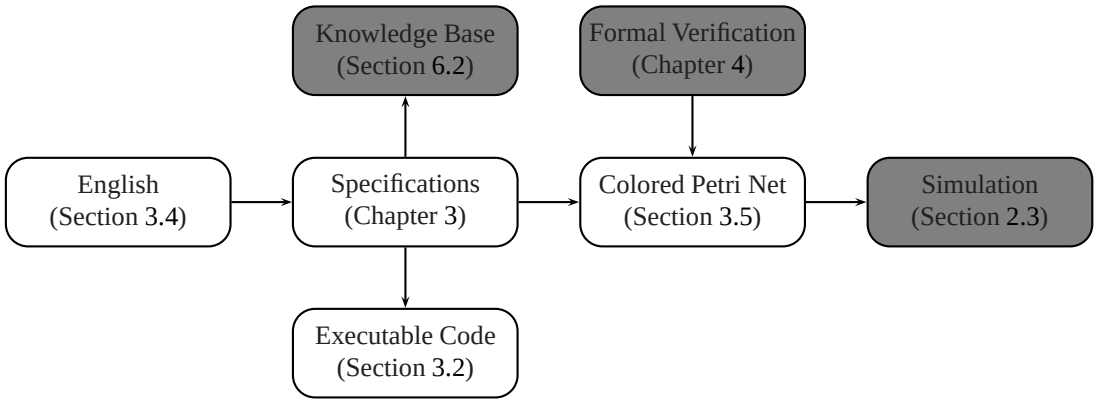

Figure 3.1: Graphical representation of CAPS focusing on the specification language.

Background information is given in Section 3.1. An overview of CAPSL is provided in Section 3.2; the statements of the specification language are discussed in Section 3.3. The natural language processing system of CAPS is described in Section 3.4, and the conversion of the specification language into colored Petri nets is discussed in Section 3.5. A summary of the chapter is given in Section 3.6.

3.1 Background

A wide variety of specification representations have been used. Some examples include high-level languages, very high-level languages, and formal languages. The specification language of a system is directly tied to the representation of the specification. More information about specification representations is given in Section 2.2.

There are different views on what qualifies as a good specification language. There is no one specification language that is suitable for all applications. For our purposes, a good specification language allows implementation of a specification with a reasonable amount of time and resources, allows easy maintenance of the specification, allows for concurrent specifications, and supports formal verification of the specifications. To this end, we have developed CAPSL, the specification language of CAPS.

3.2 Overview

CAPSL, the specification language of CAPS, is designed both as a component of CAPS and as an independent system. As one component of CAPS, the code of the specification language can be read and automatically converted into a colored Petri net for formal verification. As a separate component, the CAPSL code can be directly executed. In §3.2.1, the model of CAPSL is discussed. The parser of CAPSL converts a text file into a model and is presented in §3.2.2, and the interpreter of CAPSL, which is responsible for executing a model, is described in §3.2.3.

3.2.1 Model

A model of CAPSL may either be executed by the interpreter or converted into a colored Petri net. Once in the form of a colored Petri net, a simulation can be run or the specifications can be formally verified. The model consists of three parts: (i) a global scope, (ii) specification structures, and (iii) algorithm structures.

3.2.1.1 Global Scope

In the global scope of a model, variable declarations (see §3.3.3) can be made such that the declared variables are accessible across the entire model. An example is:

```
integer i
integer j = 2

start specification ‘‘Example 3.1’’
```

```
  i = j + 3
end specification
```

The variables `i` and `j` are declared as integers and an initial assignment to `j` is made. These variables are available across all threads that will be created during execution of the model. Variable declarations, with optional initial assignments, are the only valid statements allowed within the global scope.

3.2.1.2 Specification Structures

The syntax of the specification structure of CAPSL is derived from Metaslang, the specification language of SpecWare [53]. However, the semantics are different. Below is a simple example of two SpecWare specifications that come from [46]:

```
Symbols = spec
  type Symbol
endspec

Words = spec
  import Symbols
  type Word = List Symbol
endspec
```

In the example, the specification `Symbols` defines a type `Symbol`, a character. The specification `Words` defines a type `Word`, a list of `Symbol`s.

When a call to a specification structure is made, the code within the specification is executed in a separate thread. This allows the specification structure to be used for concurrent computing. Memory can either be shared or private between threads. Shared memory consists of global variables, while local variables are private.

Every CAPSL model must have at least one specification structure. This is known as the *prime specification* and is where execution of the model will begin. Unlike the C/C++ `main` function, or the SMV `main` module, the prime specification is identified by its location in the model; it is always the first specification listed. The format of a specification structure is:

```
start specification ‘‘name’’
  statement1
  statement2
  ⋮
end specification
```

While the global scope of a model is restricted to variable declarations, specification structures do not have this restriction; the following example demonstrates variable declarations, assignments, and arithmetic:

```
start specification ‘‘Example 3.2’’
```

```
  integer i = 0
  integer j = 2
  i = j + 3
end specification
```

A specification may also make calls to other specification or algorithm structures:

```
integer i = 0

start specification ‘‘Example 3.3’’
  specification ‘‘Spec A’’
  algorithm ‘‘Alg A’’
end specification

start specification ‘‘Spec A’’
  when i == 5
    i = 2
  end when
end specification

start algorithm ‘‘Alg A’’
  i = 5
end algorithm
```

When the call to specification `‘‘Spec A’’` is made, a separate thread is created. The variable `i` is declared in global scope; thus, it is available to all of the executing threads. The `when` statement in the example halts execution until its condition becomes true (see §3.3.10).

3.2.1.3 Algorithm Structures

The format of an algorithm structure follows that of a specification:

```
start algorithm ‘‘name’’
  statement1
  statement2
  ⋮
end algorithm
```

Algorithm and specification structures are different in that there is no prime algorithm – execution must begin in a specification, and when a call to an algorithm structure is made, a new thread is not created. To this extent, the algorithm structure behaves similar to a C/C++ function with the difference that the algorithm can access shared memory.

3.2.2 Parser

The CAPSL parser is a *lookahead LR (LALR) parser* [2]. As with C/C++, CAPSL files are stored as plain text. The parser takes in a text file and returns a CAPSL model that may include global variable

declarations, specification structures, and algorithm structures. Error handling is built into the parser to alert the user of any potential problems encountered when parsing the file. Comments may be included in CAPSL files and begin with a %. An example is:

```
start specification ‘‘Example 3.4’’
  % create a variable
  integer i = 3
end specification
```

Comments in the text file are skipped and not added to the model. A CAPSL model can also be converted into parser-readable text. The example above would be converted to:

```
start specification ‘‘Example 3.4’’
  integer i
  i = 3
end specification
```

Note that the statement `integer i = 3` is replaced by two statements: an assignment combined with a variable declaration is represented as two separate statements by the specification language.

3.2.3 Interpreter

There are two different methods of executing a CAPSL model. The first method is to convert the model into a colored Petri net (see Section 3.5). After the conversion, a simulation of the net can be run (see Chapter 2). An advantage of this approach is that the speed of the simulation can be set, and there is a visual representation of the model. The second method is to use the CAPSL interpreter. The execution of the interpreter is faster than the CP-net simulation, and the interpreter can be used separately from CAPS.

The interpreter is designed for multiple levels of concurrency. This includes handling the issues of concurrency for the specification language: creating new threads upon the call of a specification structure and enabling shared memory for global variables. With each specification call, a new thread is created. The id of the new thread must be saved in the specification or algorithm structure in which the thread was created. At the end of the specification or algorithm, a join is done on the id’s of any threads that were created. Additionally, the interpreter itself is designed for concurrency. This allows a statement to be accessed for multiple purposes at one time. This may include executing a statement, displaying a text representation of the statement, and displaying debugging information concerning the statement.

Although execution of the interpreter is faster than that of a colored Petri net simulation, the interpreter uses a significant amount of time and memory when compared to an optimized compiler for C/C++. Optimization of the interpreter is an area of future work and is discussed further in §6.2.1.

3.3 Statements

This section examines the statements of CAPSL: the CAPS specification language. Statements in CAPSL may be located at either the global level or within a specification or algorithm structure (see Section 3.2). All CAPSL code, other than specification and algorithm declarations, is a statement.

The most basic statements of CAPSL concern *constants* (§3.3.1), *variables* (§3.3.2), *variable declarations* (§3.3.3), *collections* (§3.3.4), and *assignments* to variables (§3.3.5). Binary statements, those with a left and a right-hand side, include *arithmetic* (§3.3.6), *comparison* (§3.3.7), and *boolean* statements (§3.3.8); however, it is possible for arithmetic and boolean statements to be unary. Block statements of the language include an *if-else* statement for conditional execution (§3.3.9) and a *when* statement for conditional execution during concurrency (§3.3.10). For iteration, there are *while* (§3.3.11), *break* (§3.3.12), and *continue* statements (§3.3.13). Formal verification of the specification language is achieved through colored Petri nets (CP-nets). §3.3.14 describes statements used to test CP-net properties such as fairness and liveness.

3.3.1 Constants

The constant values of CAPSL are based on C/C++ and include numbers – both integers and reals, booleans, strings, and characters. Table 3.1 contains examples of each type of constant:

Constant	Description
`7`	An integer constant
`2.3`	A real constant
`true`	A boolean constant
`''example''`	A string constant
`'a'`	A character constant

Table 3.1: Examples of CAPSL constants.

Whenever a constant value is read in by the parser (see §3.2.2), a *constant statement* is created and added to the CAPSL model (see §3.2.1) as either a statement in a specification or algorithm structure or as part of another statement.

3.3.2 Variables

Variables of CAPSL can include calls to specification or algorithm structures, numbers, booleans, strings, characters, sets, and sequences. Figure 3.2 contains a tree in which the leaves are valid variable types:

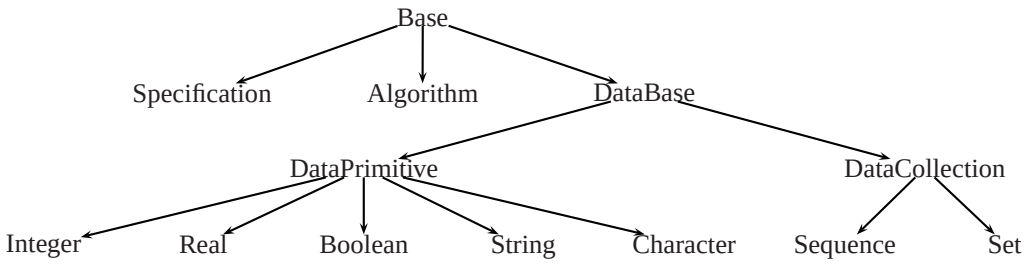

Figure 3.2: The hierarchy of the specification elements. Leaves of the tree are valid variables.

The organization of the tree reflects the underlying structure of CAPSL. For example, any DataPrimitive can be converted into another DataPrimitive. Therefore, a character can be converted into a boolean. The same holds true for a DataCollection.

3.3.3 Variable Declarations

In order to use a variable in CAPSL, there must be an explicit variable declaration. Table 3.2 contains examples of several variable declarations:

Declaration	Description
`integer i`	Declare an integer variable `i`
`real r`	Declare a real variable `r`
`boolean b`	Declare a boolean variable `b`
`string s`	Declare a string variable `s`
`character c`	Declare a character variable `c`
`sequence x`	Declare a sequence variable `x`
`set y`	Declare a set variable `y`

Table 3.2: Examples of CAPSL variable declarations.

The type of a variable can not be changed manually after it has been declared; however, in certain instances, CAPSL may automatically change the type and value of a variable for an intermediate result. For example, if an integer variable is assigned the value of a real variable, the real value will be converted into an integer. At the end of the assignment, the real variable will contain the same value as prior to the assignment.

3.3.4 Collections

While a variable holds one value, a collection may hold zero or more values. The collection type of CAPSL is known as a DataCollection. However, a DataCollection object may not be instantiated directly; a

sequence or *set* must be used. Table 3.3 gives examples of the DataCollection statements:

Statement	Description
`add x 5`	Add the value 5 to the collection `x`
`clear x`	Clear all elements out of the collection `x`
`count x`	Count the number of elements in the collection `x`
`|x|`	An alternative form of count
`remove x 5`	Remove the value 5 from the collection `x`

Table 3.3: Examples of CAPSL collection statements.

A sequence is a list of elements ordered by index. It is similar to an array in C/C++ with the exception that a sequence in CAPSL is dynamic; therefore, values can be added to a sequence without the need to manually reallocate memory. Table 3.4 gives examples of sequence statements:

Statement	Description
`x[3]`	Used to retrieve or set the 4th element of the sequence `x`
`removeat x 3`	Remove the 4th element from the sequence `x`

Table 3.4: Examples of CAPSL sequence statements.

A set in CAPSL is based on a mathematical bag. Table 3.5 gives examples of set statements; the examples include set operations that determine if a value is a member of a set and the union operation on two sets:

Statement	Description
`member y 5`	Returns a boolean indicating if 5 is in set `y`
`subset y z`	Returns a boolean indicating if set `z` is a subset of set `y`
`union y z`	Returns a set that has all of the elements of the sets `y` and `z`
`intersection y z`	Returns a set that has only the elements that are in both the sets `y` and `z`
`difference y z`	Returns a set that has all of the elements of set `y` that are not in `set z`

Table 3.5: Examples of CAPSL set statements.

CAPSL is able to automatically convert a sequence into a set or to convert a set into a sequence. Therefore, it is possible to write statements that will take the union of a set and a sequence or test if a sequence is a subset of a set.

3.3.5 Assignments

Assignment of a value to a variable, whether that value comes from a constant or another variable, is similar to assignment in C/C++ with the exception that assignment between different types will often result

in an automatic conversion. Some examples are given in Table 3.6:

Assignment	Description
`i = 1`	Variable `i` set to integer value 1
`i = 1.3`	Real value converted to integer value 1
`i = true`	Boolean value converted to integer value 1
`i = ‘‘1.3’’`	String value converted to integer value 1
`i = ‘1’`	Character value converted to 1 rather than ASCII 49
`i = x[3]`	Variable `i` set to 4th element of sequence `x`
`i = member y 5`	Variable `i` set to 0 or 1 depending on if 5 is in set `y`

Table 3.6: Examples of CAPSL assignments.

The specification language is thread-safe. This ensures that during the assignment of a variable no other thread is accessing that variable; otherwise, a race condition may occur.

3.3.6 Arithmetic Operations

Arithmetic in CAPSL is similar to arithmetic in C/C++; however, automatic conversion of data types may be performed. Arithmetic operators include *addition*, *subtraction*, *multiplication*, *division*, *modulo*, and *exponential*. Some examples are given in Table 3.7:

Arithmetic Operation	Description
`1 + 1.3`	The integer 1 is converted to a real before addition takes place
`1 + ‘‘1.3’’`	The integer and string are converted to reals before addition takes place
`i + ‘1’`	The character is converted to an integer 1 before addition takes place
`i + true`	The boolean value is converted to an integer 1 before addition takes place

Table 3.7: Examples of CAPSL arithmetic statements.

In the first two cases of the table, the results of the additions are the real value 2.3. In the last two cases, the results are the integer value 2.

3.3.7 Comparison Operations

As with arithmetic operations, comparison operations in CAPSL are similar to comparisons in C/C++ with the exception that automatic conversion of data types may be performed. Some examples of comparison operations are shown in Table 3.8:

A comparison operation will always return a boolean value, but this value can be converted into any other DataPrimitive type.

Comparison Operation	Description
`1 < 1.3`	The returned boolean value will be `true`
`1 < ''1.3''`	The returned boolean value will be `true`
`1 < '1'`	The returned boolean value will be `false`
`1 < true`	The returned boolean value will be `false`

Table 3.8: Examples of CAPSL comparison statements.

3.3.8 Boolean Operations

The boolean operations of CAPSL differ from C/C++ and include *not*, *and*, *or*, *nand*, *nor*, *xor*, and *xnor*. Table 3.9 contains some examples of boolean operations:

Boolean Operation	Description
`true or false`	The returned boolean value will be `true`
`1 < ''1.3'' and true`	The returned boolean value will be `true`
`true and false`	The returned boolean value will be `false`
`not true`	The returned boolean value will be `false`

Table 3.9: Examples of CAPSL boolean statements.

As with comparison operations, a boolean value is always returned from a boolean operation but may be converted into another DataPrimitive type.

3.3.9 If-Else

The *if-else* is a statement block that allows the statements within to be executed conditionally. All statement blocks in CAPSL must have an `end` followed by the statement type to identify the end of the block. The format of the standard *if* statement is:

```
if condition
  if-statement1
  if-statement2
  ⋮
end if
```

An *if-else* statement can also have an *else* block. If the condition of the statement is true, the statements within the *if* block will be executed; otherwise, provided there is an *else* block, the statements within the *else* block will be executed. The format of an *if-else* statement is:

```
if condition
  if-statement1
  if-statement2
  ⋮
```

```
else
  else-statement1
  else-statement2
  ⋮
end if
```

An example of a simple *if* statement is:

```
start specification ‘‘Example 3.5’’
  integer i = 0

  if i < 10
    i = i + 1
  end if
end specification
```

The value of `i` will be incremented if `i` is less than 10. The condition of an *if-else* statement must be a boolean value; however, CAPSL can automatically convert many data types to a boolean. For example, a string could be used as the condition:

```
start specification ‘‘Example 3.6’’
  integer i = 0

  if ‘‘true’’
    i = i + 1
  end if
end specification
```

3.3.10 When

The *when* statement is a statement block designed for concurrency and is based on the *wait* statement of SMV [54]. Both a *when* and an *if* statement are executed when the condition is true. However, if the condition of an *if* statement is false, execution will continue after the *if* statement (or in an *else* block if one is provided). With the *when* statement, execution will halt if the condition is false and will not resume until the condition becomes true. The execution halted is for the thread on which the *when* statement is executing. For the condition to become true, variables contained in the condition must be modified by another thread. As with other statement blocks, an `end when` is used to identify the end of a *when* statement. The format of the statement is:

```
when concurrent-condition
  when-statement1
  when-statement2
  ⋮
end when
```

A *when* statement's form is similar to an *if* statement with the exception that there are no *else* blocks.

A simple example of a *when* statement is:

```
start specification ‘‘Example 3.7’’
  integer i = 0

  when i < 10
    i = i + 1
  end when
end specification
```

In the example above, the specification or algorithm structure that contains the *when* statement will halt execution until the variable `i` is less than 10. For execution to resume, `i` must be updated in a separate specification or algorithm. Each time the variable `i` is updated by another thread, a signal is sent to the *when* statement. The condition of the *when* statement is then rechecked. Once the condition of the *when* statement is true, the statements contained within are executed.

3.3.11 While

The *while* statement is an iterative statement block and is, in fact, the only iterative statement of CAPSL. The for, forall, and do-until statements were not included as the syntax of these statements differed from the *if-else* and *when* statements. As with other statement blocks, an `end while` is used to terminate the statement. The format of the *while* statement is:

```
while condition
  while-statement1
  while-statement2
  ⋮
end while
```

Statements located within the *while* statement will be executed as long as the condition is true. An example of a simple *while* statement is:

```
start specification ‘‘Example 3.8’’
  integer i = 0

  while i < 10
    i = i + 1
  end while
end specification
```

The *while* statement above will be executed as long as `i` is less than 10. As with the *if* statement, CAPSL will try to handle conversion of the condition if it is not in the form of a boolean value.

3.3.12 Break

A *break* statement is used to force an exit of a *while*. An example of using a *break* is:

```
start specification ‘‘Example 3.9’’
  integer i = 0

  while true
    i = i + 1
    if i = 10
      break
    end if
  end while
end specification
```

In the code shown above, the condition of the *while* will always be true. The *break* statement forces the loop to stop execution when the value of `i` is equal to, or exceeds, 10.

3.3.13 Continue

A *continue* statement is used in a *while* to force execution to jump to the start of the block. An example is:

```
start specification ‘‘Example 3.10’’
  integer i = 0

  while i < 10
    if i > 0
      i = i + 1
      continue
    end if
    i = i + 2
  end while
end specification
```

In the example, if the value of `i` is less than, or equal to, 0, then `i` is incremented by 2. When the value of `i` is greater than 0 and less than 10, `i` is incremented by 1.

3.3.14 Formal Verification

Formal verification of a CAPSL model is achieved by CAPS first converting the specification model into a colored Petri net (see Section 3.5). Once in the form of a colored Petri net, the model can be formally verified (see Chapter 4). Verification of a colored Petri net involves the comparison of markings and bindings. There are statements built into CAPSL to automatically construct these objects. Examples of these statements are given in Table 3.10:

Statement	Description
`reachable i == 2`	True if integer `i` can be equal to 2
`boundedness x 7`	The number of 7s sequence `x` can have over all states

Table 3.10: Examples of CAPSL formal verification statements.

Reachability and *boundedness* are two formal verification properties of colored Petri nets. Other properties include *home*, *liveness*, and *fairness*. These formal verification properties of CAPS are further discussed in Chapter 4.

3.4 Conversion of English to Specifications

A natural language processing system is built into CAPS in order to convert English phrases into CAPSL structures and statements; however, there are many difficulties with using natural language. Natural language is often ambiguous and even a simple sentence may contain several possible interpretations. In §3.4.1, the dictionary of CAPS is discussed, and the conversion of English to CAPSL is explained in §3.4.2.

3.4.1 Dictionaries

At the core of the natural language processing engine, there is a *dictionary*. The dictionary has three distinct parts: a general use dictionary provided to the user, a dictionary for use by a part-of-speech tagger, and a dictionary for future work involving speech recognition.

3.4.1.1 Online Plain Text English Dictionary

CAPS' dictionary incorporates OPTED in order to provide a general use dictionary to the user. The Online Plain Text English Dictionary (OPTED) is a public domain dictionary based on the Webster dictionary [81]. OPTED contains more than 100,000 words and is used by many websites and desktop applications, such as clients for the MIT dictionary server. Each word may contain several parts-of-speech and each part-of-speech may contain several definitions; see Table 3.11 for the part-of-speech tags:

3.4.1.2 Penn Treebank Tagged Dictionary

The Penn Treebank tagged dictionary is created by scripts from the Brown corpus and Wall Street Journal (WSJ) articles that have been tagged using the Penn Treebank tagset found in Table 3.12. The

Noun	Adjective	Verb Transitive	Preposition
Noun Plural	Superlative	Adverb	Conjunction
Verbal Noun	Participle Adjective	Participle Past	Comparative
Dative	Verb	Participle Present	Interjection
Pronoun	Verb Intransitive	Imperfect	-

Table 3.11: List of the parts-of-speech in the Online Plain Text English Dictionary.

dictionary contains nearly 100,000 words in which each word is associated with a set of Penn Treebank tags. The tags of a word are those that have been encountered in either the Brown corpus or WSJ articles. In the set of tags for a word, the first tag is the one that occurred most frequently for the word; the other tags are in no particular order.

Tag	Description	Tag	Description
CC	Coordinating Conjunction	PP$	Possessive Pronoun
CD	Cardinal Number	RB	Adverb
DT	Determiner	RBR	Adverb, Comparative
EX	Existential *there*	RBS	Adverb, Superlative
FW	Foreign Word	RP	Particle
IN	Preposition or Subordinating Conjunction	SYM	Symbol
JJ	Adjective	TO	*to*
JJR	Adjective, Comparative	UH	Interjection
JJS	Adjective, Superlative	VB	Verb, Base Form
LS	List Item Marker	VBD	Verb, Past Tense
MD	Modal	VBG	Verb, Gerund or Present Participle
NN	Noun, Singular or Mass	VBN	Verb, Past Participle
NNS	Noun, Plural	VBP	Verb, Non-3rd Person Singular Present
NP	Proper Noun, Singular	VBZ	Verb, 3rd Person Singular Present
NPS	Proper Noun, Plural	WDT	Wh-Determiner
PDT	Predeterminer	WP	Wh-Pronoun
POS	Possessive Ending	WP$	Possessive Wh-Pronoun
PP	Personal Pronoun	WRB	Wh-Adverb

Table 3.12: The Penn Treebank Tagset. An expanded tagset of English.

Note that there are more part-of-speech tags in Table 3.12 than there are in Table 3.11. While Table 3.11 contains all of the tags necessary for English, researchers have found it difficult to create software that accurately tags a sentence using only the 19 part-of-speech tags in Table 3.11. A number of different tagsets have been developed with some sets containing as many as 146 tags. CAPS uses the Penn Treebank tagset, a popular tagset for natural language systems.

3.4.1.3 Carnegie Mellon Pronouncing Dictionary

Carnegie Mellon University's Sphinx group has published the CMU Pronouncing Dictionary (CMUDict) [84] that contains more than 125,000 English words. Each word in the dictionary is associated with a set of phonemes. These phonemes are used by an automatic speech recognition (ASR) engine, such as Sphinx, to convert spoken English into text. Table 3.13 gives the list of phonemes that are used in CMUDict:

Phone	Example	Phone	Example	Phone	Example	Phone	Example
/aa/	**Au**stere	/eh/	**E**dison	/l/	**L**iebnitz	/t/	**T**riton
/ae/	**A**tlantis	/er/	**Er**nst	/m/	**M**ythology	/th/	**Th**or
/ah/	**A**ttest	/ey/	**Eigh**t	/n/	**N**ovel	/uh/	T**u**ring
/ao/	**Au**tonomy	/f/	**F**aulkner	/ng/	Dilli**ng**er	/uw/	**Uh**lmann
/aw/	**Ou**st	/g/	**G**alileo	/ow/	**O**asis	/v/	**V**oyager
/ay/	**I**reland	/hh/	**H**awthorne	/oy/	**Eu**ler	/w/	**Wh**itman
/b/	**B**eethoven	/ih/	**I**carus	/p/	**P**acific	/y/	**Y**ork
/ch/	**Ch**ief	/iy/	**I**chiro	/r/	**R**aphael	/z/	**X**ylophone
/d/	**D**ecode	/jh/	**J**oyce	/s/	**S**cience	/zh/	**Zh**ivago
/dh/	**Th**ereby	/k/	**K**epler	/sh/	**Sch**roeder	/sil/	Silence

Table 3.13: List of the phonemes used in CMUDict. A phoneme is included to represent silence.

CMUDict has been integrated into the dictionary of CAPS in order to allow for future research on automatic speech recognition. This is discussed further in §6.2.3.

3.4.2 CAPSL Structure and Statement Interpretation

Natural language processing of CAPS consists of two parts: *part-of-speech (POS) tagging* and *template matching*. For part-of-speech tagging, a *transformation-based tagger* is used (see §2.4.1). The specific POS tagger implementation is MontyLingua [50]. Template matching is used to interpret English phrases as CAPSL structures and statements and was chosen as it generally gives better results for specific applications than that of other methods [3].

3.4.2.1 Lexicon

The natural language processing system of CAPS is designed for a specific application; therefore, a *lexicon* has been created with the application in mind. The lexicon is a subset of the CAPS dictionary (see §3.4.1). A portion of the lexicon is shown in Table 3.14:

Words in the left-hand column may be encountered during the parse of an English phrase. If the word is tagged with the part-of-speech shown in the right-hand column, then the type is set to the value in

Word	Part-of-Speech and Type	Word	Part-of-Speech and Type
specification	(NN TYPE SPECIFICATION)	plus	(CC TYPE ARITHMETIC)
specifications	(NNS TYPE SPECIFICATIONS)	less than	(JJR IN TYPE COMPARISON)
add	(VB TYPE NEW)	equals	(VBZ TYPE ASSIGNMENT)
delete	(VB TYPE REMOVE)	and	(CC TYPE BOOLEAN)
prime	(JJ TYPE PRIME)	if	(IN TYPE IF)
integer	(NN TYPE VARIABLE)	if-else	(JJ TYPE IFELSE)
integers	(NNS TYPE VARIABLES)	focus	(NN TYPE FOCUS)
variables	(NNS TYPE VARIABLES)	current	(JJ TYPE CURRENT)
call	(NN TYPE CALL)	"..."	(NAME)

Table 3.14: A subset of the lexicon for the CAPS NLP.

the right-hand column. For example, if the word *integers* is read in, and the part-of-speech is a NNS, then the word is of type VARIABLES. The tag given to the word during POS tagging is partially dependent upon the context of the word (see §2.4.1).

The lexicon includes many different words and symbols that may be used for the same type. For example, Table 3.14 shows that *add*/VB is of type NEW; *new*/JJ and *create*/VB are also of the same type. Other words, such as *call*, may be used in different contexts. Therefore, several different parts-of-speech for *call*, such as *call*/VB, are also of type CALL.

3.4.2.2 Specification and Algorithm Structures

There are several operations concerning specification and algorithm structures. These operations include creating a new specification or algorithm structure, removing a structure, removing all structures, or moving a specification in order to make it the prime specification. The templates for creating a new specification structure are:

<NEW>...<SPECIFICATION> → (NEWSPEC 2)
<NEW>...<SPECIFICATION>...<NAME> → (NEWSPEC 3)

The templates are used when a word of type NEW is found preceding a word of type SPECIFICATION. The ... indicate that there may be 0 or more words between the NEW and SPECIFICATION types. Optionally, a name may be given for a specification; a name is any string that appears in double quotes. For example, the phrase *Add specification "Example 3.11"* parses to *Add*/VB *specification*/NN *"Example 3.11"*. The types of the parse are NEW SPECIFICATION NAME, and the second template is matched. Once this occurs, a new specification with name "Example 3.11" is added to the CAPSL model. The templates for removing a specification structure are:

<REMOVE>...<SPECIFICATION> → (REMOVESPEC 2)

<REMOVE>...<SPECIFICATION>...<NAME> → (REMOVESPEC 3)

Given the phrase *Delete specification "Example 3.11"*, the resulting parse is *Delete*/VB *specification*/NN *"Example 3.11"*. The types are REMOVE SPECIFICATION NAME, the types match the second template; therefore, the specification "Example 3.11", provided there is one, is removed from the model. The template for removing all specification structures is:

<REMOVE>...<SPECIFICATIONS> → (REMOVESPECS 2)

Delete all of the specifications parses to *Delete*/VB *specifications*/NNS - note that only the words in the lexicon are tagged. The types REMOVE SPECIFICATIONS match the template, and all of the specification structures of the model are removed. There are corresponding templates for the algorithm structures as well. Additionally, there are two prime specification templates:

<PRIME> → (PRIMESPEC 1)
<SPECIFICATION>...<NAME>...<PRIME> → (PRIMESPEC 3)

The prime specification is where execution of the model begins; the prime specification is always the first specification of the model. The phrase *Make specification "Example 3.11" the prime specification* parses to *specification*/NN *"Example 3.11" prime*/JJ. The types of the parse are SPECIFICATION NAME PRIME. If there is a specification "Example 3.11", it is moved such that it becomes the first specification of the model. The first prime specification template uses the current context of the NLP system. Context is kept in terms of the current structure, block, and line. The phrase *Make current specification prime* would parse to *prime*/JJ. If the current context contained information about a specification structure, that specification would become the prime specification.

3.4.2.3 Constant Statements

Examples of constant values for CAPSL are given in Table 3.1. The natural language processing system of CAPS parses an English phrase to determine if it contains any CAPSL constants. If a constant is found, it is given a type dependent upon the constant. A CONSTANT is defined as:

<CONSTANT> →<NUMBER>
<BOOLEAN>
<CHARACTER>
<STRING>

NUMBER is defined as:

<NUMBER> →<INTEGER>
<REAL>

In the case of constants, the tags from the part-of-speech tagger are not used. The NLP system searches for constants from the original English phrase and performs some conversions. For example, the

word *3rd* is converted into the integer 3.

3.4.2.4 Variable Statements

Valid variables of CAPSL are shown in Figure 3.2. The templates for adding a new DataPrimitive or DataCollection variable are:

<NEW>...<VARIABLE> → (NEWVAR 2)
<NEW>...<VARIABLE>...<NAME> → (NEWVAR 3)
<NEW>...<VARIABLE>...<SPECIFICATION>...<NAME> → (NEWVAR 4)
<NEW>...<VARIABLE>...<NAME>...<SPECIFICATION>...<NAME> → (NEWVAR 5)

The first two templates use the current context when adding a variable to the model. An example of using the last template is the phrase *Add integer "i" to specification "Example 3.11"*. The sentence parses to *Add*/VB *integer*/NN *"i" specification*/NN *"Example 3.11"*, and the types are NEW VARIABLE NAME SPECIFICATION NAME. Note that a variable name is not specified in the first and third templates. In these cases, the name is generated automatically. The templates for removing a variable are:

<REMOVE>...<NAME> → (REMOVEVAR 2)
<REMOVE>...<VARIABLE>...<NAME> → (REMOVEVAR 3)
<REMOVE>...<NAME>...<SPECIFICATION>...<NAME> → (REMOVEVAR 4)
<REMOVE>...<VARIABLE>...<NAME>...<SPECIFICATION>...<NAME> →
(REMOVEVAR 5)

As before, the first two templates use the current context. The phrase *Delete integer "i" from specification "Example 3.11"* parses to *Delete*/VB *integer*/NN *"i" specification*/NN *"Example 3.11"*. The types of the parse are REMOVE VARIABLE NAME SPECIFICATION NAME, and the fourth template is used. It is also possible to remove more than one variable at a time:

<REMOVE>...<VARIABLES> → (CLEARVARS 2)
<REMOVE>...<VARIABLES>...<SPECIFICATION>...<NAME> → (REMOVEVARS 4)

Given the phrase *Delete all of the integers of specification "Example 3.11"*, any integer variables in specification "Example 3.11" are removed, while the phrase *Delete all of the variables of specification "Example 3.11"* results in all variables of the specification being removed.

Specification and algorithm structures can be treated as variables but calls to them are handled separately. The templates for making a specification call are:

<CALL>...<SPECIFICATION>...<NAME> → (CALLSPEC 3)
<CALL>...<SPECIFICATION>...<NAME>...<SPECIFICATION>...<NAME> →
(CALLSPEC 5)

The phrase *Make call to specification "Example 3.11" from specification "Example 3.10"* parses to *call*/NN *specification*/NN *"Example 3.11" specification*/NN *"Example 3.10"*, and the types of the parse

match the second template; therefore, a call to specification “Example 3.11” is added in specification “3.10”. The templates for removing a call to a specification structure are:

<REMOVE>...<CALL>...<SPECIFICATION>...<NAME> → (REMOVECALLSPEC 4)
<REMOVE>...<CALL>...<SPECIFICATION>...<NAME>...<SPECIFICATION>...
<NAME> → (REMOVECALLSPEC 6)

Given the phrase *Remove call to specification “Example 3.11” from specification “Example 3.10”*, the resulting parse is *Remove*/VB *call*/NN *specification*/NN *“Example 3.11” specification*/NN *“Example 3.10”*, and types of the parse match the second template.

3.4.2.5 Assignment Statements

Assignment statements were discussed in §3.3.5, and there are several templates for adding an assignment statement to a CAPSL model. These templates are:

<NAME>...<ASSIGNMENT>...<VALUE> → (ASSIGN 3)
<VARIABLE>...<NAME>...<ASSIGNMENT>...<VALUE> → (ASSIGN 4)
<NAME>...<ASSIGNMENT>...<VALUE>...<SPECIFICATION>...
<NAME> → (ASSIGN 5)
<VARIABLE>...<NAME>...<ASSIGNMENT>...<VALUE>...<SPECIFICATION>...
<NAME> → (ASSIGN 6)

The first two templates use the current context. The second and fourth templates include the type of the variable; however, this is not the same as a variable declaration. For instance, the phrase *Integer “i” equals 3* parses to *Integer*/NN *“i” equals*/VBZ *3*. The types of the parse match the second template, and the statement `i = 3` is added to the current context. The type VALUE is defined as:

<VALUE> →<CONSTANT>
<VARIABLE>...<NAME>
<ARITHMETICMINUS>...<VALUE>
<VALUE>...<ARITHMETIC>...<VALUE>
<VALUE>...<COMPARISON>...<VALUE>
<BOOLEANNOT>...<VALUE>
<VALUE>...<BOOLEAN>...<VALUE>

VALUE is used for assignment, arithmetic, comparison, and boolean statements as well as the condition of the if-else, when, and while block statements.

3.4.2.6 Arithmetic, Comparison, and Boolean Statements

Arithmetic, comparison, and boolean statements of CAPSL are discussed in §3.3.6 through §3.3.8. The templates for converting an English phrase into an arithmetic statement are:

<ARITHMETICMINUS>...<VALUE> → (ARI 2)
<VALUE>...<ARITHMETIC>...<VALUE> → (ARI 3)
<ARITHMETICMINUS>...<VALUE>...<SPECIFICATION>...<NAME> → (ARI 4)

<VALUE>...<ARITHMETIC>...<VALUE>...<SPECIFICATION>...<NAME> → (ARI 5)

The first two templates use the current context, and the first and third templates are for unary arithmetic operations. Take the phrase *Calculate "i" plus 3*, the resulting parse is *"i" plus*/CC *3*. The types of the parse are NAME ARITHMETIC INTEGER. NAME is a VALUE, and INTEGER is a NUMBER. A NUMBER is a CONSTANT, which is in turn a VALUE; thus, the types of the parse are VALUE ARTIHMETIC VALUE. This sequence of types matches the second template. As a result, the statement `i + 3` is added to the current context. The comparison templates are:

<VALUE>...<COMPARISON>...<VALUE> → (COMP 3)
<VALUE>...<COMPARISON>...<VALUE>...<SPECIFICATION>...
<NAME> → (COMP 5)

Comparison templates are similar to the arithmetic templates with the exception that there are no unary comparison operations. The boolean templates are:

<BOOLEANNOT>...<VALUE> → (BOOL 2)
<VALUE>...<BOOLEAN>...<VALUE> → (BOOL 3)
<BOOLEANNOT>...<VALUE>...<SPECIFICATION>...<NAME> → (BOOL 4)
<VALUE>...<BOOLEAN>...<VALUE>...<SPECIFICATION>...<NAME> → (BOOL 5)

Boolean templates follow the format of the arithmetic templates. The lexicon includes symbols as well as words; therefore, both *Calculate "i" plus 3* and *Calculate "i" + 3* would give the same result.

3.4.2.7 Block Statements

Block statements of CAPSL are explained in §3.3.9 through §3.3.11 and include if-else, when, and while statements. The templates for adding a new if statement to the CAPSL model are:

<NEW>...<IF> → (NEWIF 2)
<NEW>...<IF>...<VALUE> → (NEWIF 3)
<NEW>...<IF>...<SPECIFICATION>...<NAME> → (NEWIF 4)
<NEW>...<IF>...<VALUE>...<SPECIFICATION>...<NAME> → (NEWIF 5)

The first two templates use the current context while the first and third templates create a default condition with the boolean value false. Given the phrase *Add an if statement with condition "i" less than 3 to specification "Example 3.11"*, the parse of the phrase is *Add*/VB *if*/IN *"i" less*/JJR *than*/IN *3 specification*/NN *"Example 3.11"*. The types of the parse are NEW IF NAME COMPARISON CONSTANT SPECIFICATION NAME, which evaluates to NEW IF VALUE SPECIFICATION NAME. The sequences of types matches the fourth template, so an if statement with condition `i < 3` is added to specification "Example 3.11". There are similar templates for adding if-else, when, and while statements. The if-else templates are:

<NEW>...<IFELSE> → (NEWIFELSE 2)
<NEW>...<IFELSE>...<VALUE> → (NEWIFELSE 3)
<NEW>...<IFELSE>...<SPECIFICATION>...<NAME> → (NEWIFELSE 4)
<NEW>...<IFELSE>...<VALUE>...<SPECIFICATION>...<NAME> → (NEWIFELSE 5)

With an if-else template, if and else blocks are added to the CAPSL model. The templates for removing an if statement are:

<REMOVE>...<IF> → (REMOVEIF 2)
<REMOVE>...<INTEGER>...<IF> → (REMOVEIF 3)
<REMOVE>...<IF>...<SPECIFICATION>...<NAME> → (REMOVEIF 4)
<REMOVE>...<INTEGER>...<IF>...<SPECIFICATION>...<NAME> → (REMOVEIF 5)

The first two templates use the current context. The second and fourth templates remove a specific if block. For example, the phrase *Delete the 3rd if statement of specification "Example 3.11"* parses to *Delete*/VB *3 if*/IN *specification*/NN *"Example 3.11"*. The types of the parse are REMOVE INTEGER IF SPECIFICATION NAME, and the third if statement of specification "Example 3.11", provided there is one, is removed. The templates for removing if-else, when, and while blocks follow from the templates above.

3.4.2.8 Other

As the current context of the natural language processing system is used in many of the templates, a template is provided to aid the user in changing context:

<FOCUS>...<CURRENT> → (FOCUS 2)

Given the phrase *Set focus on current line*, the resulting parse is *focus*/NN *current*/JJ, and the types match the template. When the template is matched, the location of the cursor for the CAPS interface is taken as the current context.

3.5 Conversion of Specifications to a Colored Petri Net

Formal verification is one of the primary goals of CAPS. As can be seen in Figure 3.1, the specifications are converted into a colored Petri net. After the conversion, formal verification can be performed on the CP-net (see Chapter 4).

A mapping to a colored Petri net is provided for each CAPSL statement. Some background information is presented in §3.5.1. The conversion process itself is broken into four parts: conversion of specification and algorithm structures (§3.5.2), conversion of data types (§3.5.3), conversion of operators (§3.5.4), and conversion of block statements (§3.5.5).

3.5.1 Background

There has been some previous work done in converting programming languages into Petri nets [63]. The two approaches taken have been “objects in petri nets” and “petri nets in objects” [9]. The “objects in petri nets” approach embeds more information in a token. These types of Petri nets are known as high-level Petri nets of which colored Petri nets are the most prevalent example. The “petri nets in objects” approach uses a Petri net for modeling an object.

This research uses a combination of these approaches. Additionally, the specification language of CAPS was designed with conversion to a colored Petri net as one of its objectives. Thus the conversion from CAPSL code can be done automatically and is more extensive than other methods that have been used.

3.5.2 Specification and Algorithm Structures

In the specification language, the two primary objects are *specification* and *algorithm* structures. Each structure can contain statements such as variable declarations, assignments, arithmetic statements, and loops. This section examines the conversion of specification and algorithm structures into a CP-net.

3.5.2.1 Specification Structures

The conversion of a CAPS model into a colored Petri net consists of converting the global statements, the specification structures, and the algorithm structures. Every CAPS model must contain at least one specification; therefore, the simplest conversion would be of the following:

```
start specification ‘‘Example 3.11’’
end specification
```

Conversion of the CAPSL code into a colored Petri net is shown in Figure 3.3:

The gray boxes are shown in the figure in order to indicate a separation. The separation deals not with the CP-net itself but rather with the CAPS model that has been converted. The top gray box in the figure represents the global state of the model. It is at the global level that the prime specification is called. In this case, there are no global variables and the prime specification is “Example 3.11”. The lower gray box represents the “Example 3.11” specification.

Places, transitions, and arcs of the colored Petri net are separated into sets similar to the hierarchical CP-nets of [42]. For CAPS, these sets are defined as:

$$P = P_G \cup P_S \cup P_{Al} : P_S = \bigcup\nolimits_{s \in S} P_s \wedge P_{Al} = \bigcup\nolimits_{al \in Al} P_{al} \tag{3.1}$$

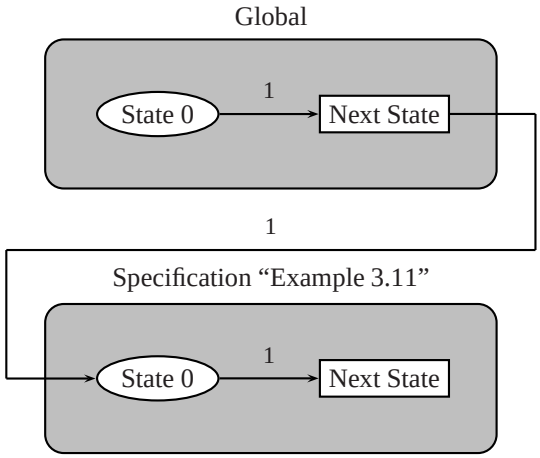

Figure 3.3: Simplest colored Petri net from conversion of a specification.

$$T = T_G \cup T_S \cup T_{Al} : T_S = \bigcup\nolimits_{s \in S} T_s \wedge T_{Al} = \bigcup\nolimits_{al \in Al} T_{al} \tag{3.2}$$

$$A = A_G \cup A_S \cup A_{Al} : A_S = \bigcup\nolimits_{s \in S} A_s \wedge A_{Al} = \bigcup\nolimits_{al \in Al} A_{al} \tag{3.3}$$

The set of places P is a union of the places representing the global state of the CAPS model P_G, the places of the specification structures P_S, and the places of the algorithm structures P_{Al}. The set P_S is the union of the places for all of the specification structures of the model while the set P_{Al} is the union of the places for all of the algorithm structures. The sets of transitions T and arcs A follow that of the places P. The global state of the model, excluding any variable declarations (see §3.5.3), can be represented by the following:

$$\exists p_0 \in P_G \wedge |C(p_0)| = 1 \wedge \exists t_0 \in T_G \tag{3.4}$$

In the equation, p_0 represents the place *State 0* in the Global box of Figure 3.3 and t_0 represents the transition in the Global box. The equation states that p_0 and t_0 are in the global place and transition sets. Additionally, p_0 contains one color. There is an arc between p_0 and t_0:

$$\exists a = (p_0, t_0) \in A_G : Type(E(a)) = C(p_0) \wedge Type(Vars(E(a))) \subseteq \Sigma \tag{3.5}$$

The arc a from p_0 to t_0 is in the set of arcs for the global state. The type of the arc expression $E(a)$ is the same as the type of color in p_0, and the type of the arc expression, including any unbound variables returned by $Vars$, is in the set of colors of the CP-net. For CAPS, the color 1 was chosen. The representation of the specification structure is similar to the global state with the exception that $|C(p_0)| = 0$.

Each call to a specification structure from within CAPSL code is represented by the equation:

$$\forall s \in S : \exists a = (t_{i-1}, p_0) \in A : Type(E(a)) = C(p_0) \wedge Type(Vars(E(a))) \subseteq \Sigma \tag{3.6}$$

S is the set of specification structures of the CAPS model, t_{i-1} is a transition representing statement i and is from either the set of transitions representing the global state or a set of transitions representing the calling specification or algorithm structure. The place p_0 is from the set of places in the called specification and represents *State 0* in the Specification "Example 3.11" box of Figure 3.3. Let us now examine a more complex example. Below is a set of three specification structures:

```
start specification ''Example 3.12''
  specification ''Spec A''
  specification ''Spec B''
end specification

start specification ''Spec A''
end specification

start specification ''Spec B''
end specification
```

Two of the specification structures are empty, and the prime specification simply calls the other specifications. Figure 3.4 shows the resulting colored Petri net:

The "Example 3.12" specification has two statements – both of which are specification calls. In the CP-net, when a transition between places *State i - 1* and *State i* fires, it represents execution of the i^{th} statement of a specification or algorithm. When the first transition in the "Example 3.12" specification box fires, it is equivalent to the statement:

```
specification ''Spec A''
```

At this point, two transitions are enabled: the transition between places *State 1* and *State 2* in the specification "Example 3.12" box and the transition in the "Spec A" specification box. When there are multiple transitions enabled, it is not known which transition will fire next. This models the behavior of the specification language.

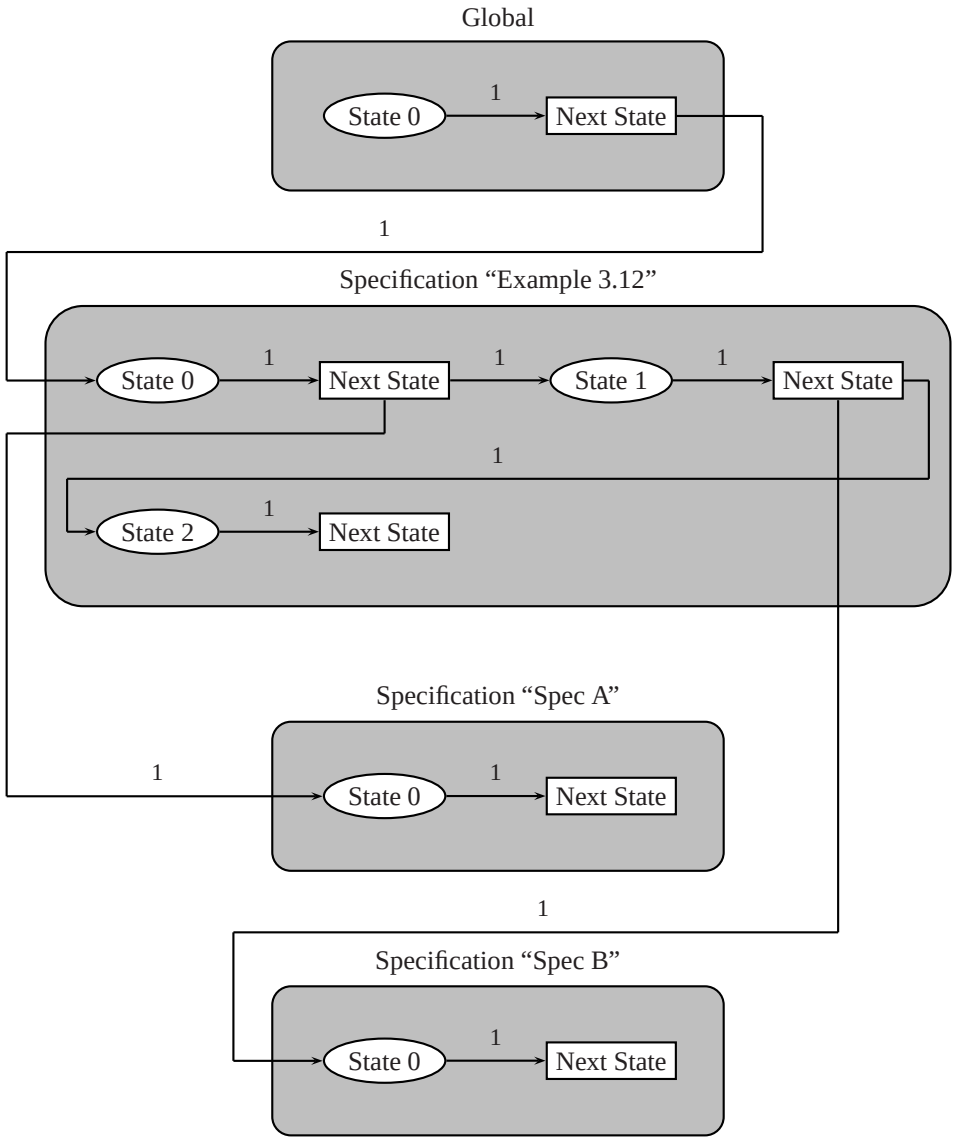

Figure 3.4: Colored Petri net from conversion of three specifications.

For each statement i added to the model, at least one place and one transition are added. With regard to a specification structure, this is represented as:

$$\exists p_i \in P_s \wedge |C(p_i)| = 0 \wedge \exists t_i \in T_s \tag{3.7}$$

The place p_i initially contains no colors. This is to simulate the sequential execution of statements within a specification or algorithm structure. There is an arc between t_{i-1} and p_i:

$$\exists a = (t_{i-1}, p_i) \in A_s : Type(E(a)) = C(p_i) \wedge Type(Vars(E(a))) \subseteq \Sigma \tag{3.8}$$

Transition t_{i-1} represents the transition added to the global state, specification, or algorithm structure for statement $i-1$. If there is no statement $i-1$, t_{i-1} represents the initial transition described in Equation 3.4 for the global state, specification, or algorithm structure. Execution of statement i is represented by the firing of transition t_{i-1}. There is also an arc from p_i to t_i:

$$\exists a = (p_i, t_i) \in A_s : Type(E(a)) = C(p_i) \wedge Type(Vars(E(a))) \subseteq \Sigma \tag{3.9}$$

The call to the prime specification from the global state is represented by:

$$\exists a = (t_n, p_0) \in A : Type(E(a)) = C(p_0) \wedge Type(Vars(E(a))) \subseteq \Sigma \tag{3.10}$$

Transition t_n represents the transition added to the global state for statement n. If no statements were added, then t_n equals t_0. Place p_0 represents the initial place of the prime specification.

3.5.2.2 Algorithm Structures

Converting an algorithm structure from the CAPS specification language into a colored Petri net is similar to the conversion of a specification structure. For example, let us convert the following specification and algorithm structures:

```
start specification ''Example 3.13''
  algorithm ''Alg A''
  specification ''Spec A''
end specification

start algorithm ''Alg A''
end algorithm
```

```
start specification ‘‘Spec A’’
end specification
```

The CP-net representing the example is shown in Figure 3.5:

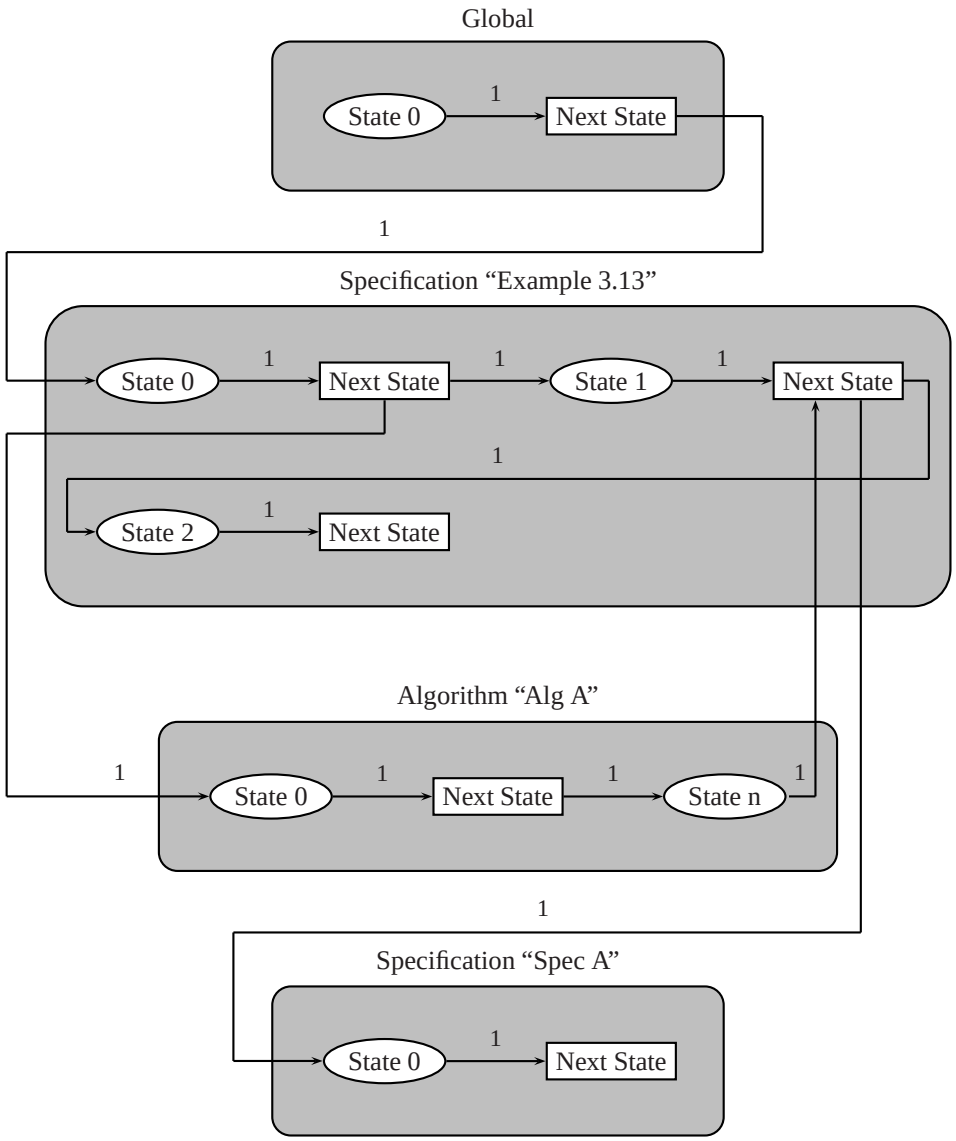

Figure 3.5: Colored Petri net from conversion of two specifications and an algorithm.

The primary difference between specification and algorithm structures is that when a specification call is made, a new thread is created for execution of the specification. When an algorithm is called, it executes on the same thread from which it was called. This is modeled in the colored Petri net by maintaining the same number of enabled transitions when a transition representing an algorithm call fires. The algorithm call in the

figure is represented by the transition between places *State 0* and *State 1* in the specification "Example 3.13" box. An algorithm call is modeled by:

$$\forall al \in Al : \exists a_1 = (t_{i-1}, p_0) \in A \wedge \exists a_2 = (p_{n+1}, t_i) \in A : Type(E(a_1)) = C(p_0) \wedge \\ Type(Vars(E(a_1))) \subseteq \Sigma \wedge Type(E(a_2)) = C(p_{n+1}) \wedge Type(Vars(E(a_2))) \subseteq \Sigma \tag{3.11}$$

Al is the set of algorithm structures, t_{i-1} and t_i are transitions from the calling specification or algorithm structure, and p_0 and p_{n+1} are places from the called algorithm. The transition t_{i-1} represents the call to the algorithm from statement i, and the transition t_i represents either the following statement or the end of the specification or algorithm structure. The place p_0 represents the initial place of the algorithm structure, *State 0*, while p_{n+1} represents the final place, *State n*. For place p_{n+1}:

$$\exists p_{n+1} \in P_{al} \wedge |C(p_{n+1})| = 0 \tag{3.12}$$

The place p_{n+1} is added to the places of the algorithm after all of the places and transitions for the statements have been added. Additionally, an arc is added from t_n to p_{n+1}:

$$\exists a = (t_n, p_{n+1}) \in A_{al} : Type(E(a)) = C(p_{n+1}) \wedge Type(Vars(E(a))) \subseteq \Sigma \tag{3.13}$$

Transition t_n represents the transition added for statement n. If no statements were added, then t_n equals t_0.

3.5.3 Data Types

Data types in the CAPS specification language fall into one of two categories: primitive data types and collection data types. Primitive data types include integers. A collection data type may consist of zero or more primitive or other collection data types. Conversion of these data types into a CP-net is discussed below.

3.5.3.1 Primitive Data Types

The primitive data types of CAPSL are *integer*, *real*, *boolean*, *string*, and *character*. An example of an integer variable declaration is:

```
integer i

start specification ‘‘Example 3.14’’
end specification
```

Figure 3.6 is the resulting CP-net:

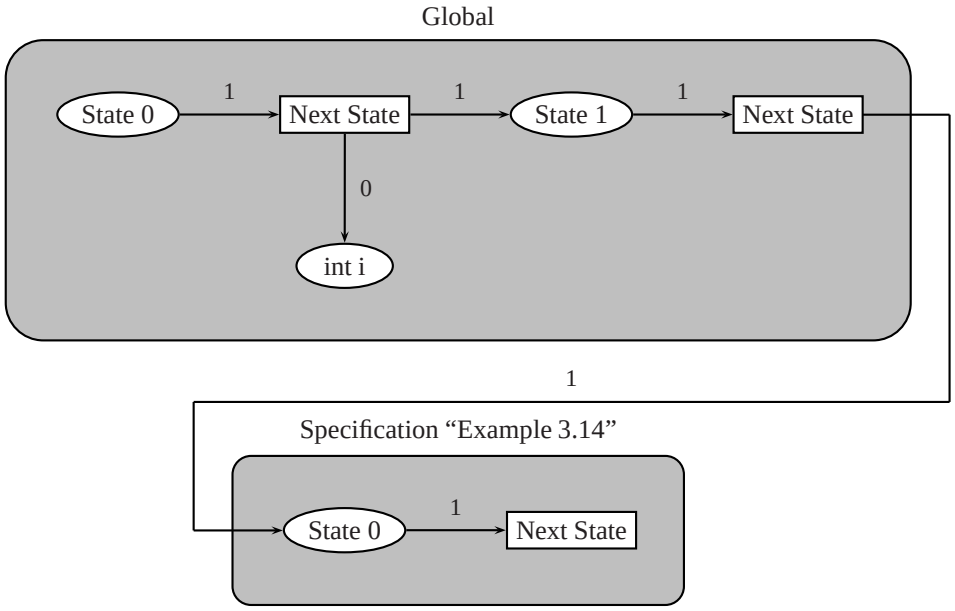

Figure 3.6: Colored Petri net example of a global variable.

In the figure, the integer `i` is represented by the place *int i*. The transition between places *State 0* and *State 1* in the global box is responsible for initializing the place representing `i`. In this case, a token that has the value of 0 is added to the place. Table 3.15 lists the default values for the different primitive data types:

Data Type	Default Value
integer	`0`
real	`0.0`
boolean	`false`
string	`‘‘’’`
character	`‘\0’`

Table 3.15: Default values of the primitive data types.

Assignments of other values to variables are discussed in §3.5.4. As the variable is declared outside of any specification or algorithm structure, the place representing the variable is located in the global box. An example of a variable declared inside of a specification is:

```
start specification ‘‘Example 3.15’’
```

```
  string s
end specification
```

The colored Petri net representation of the example is shown in Figure 3.7:

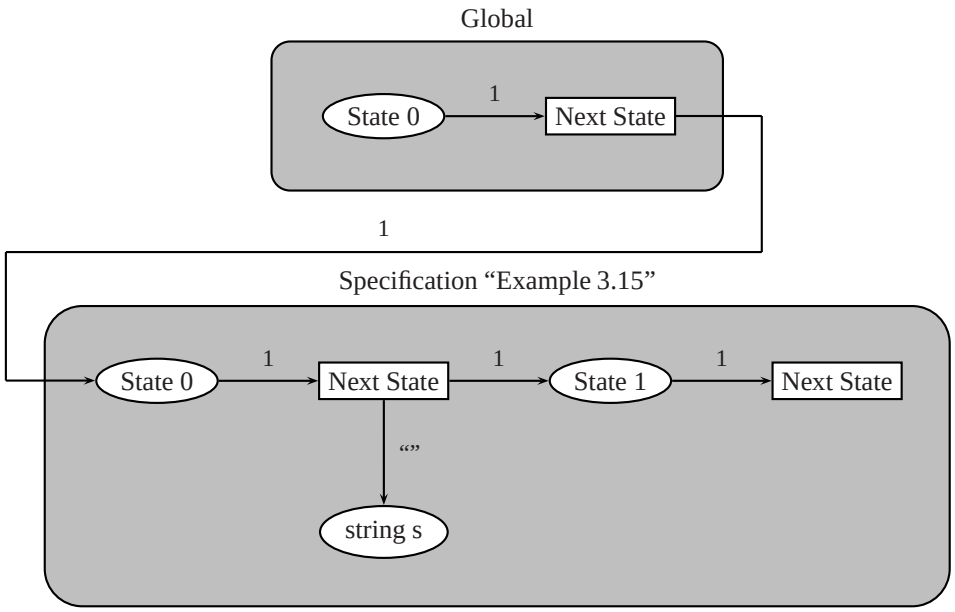

Figure 3.7: Colored Petri net example of a local variable.

In this example, the place representing the variable is in the specification “Example 3.15” box. When a primitive data type is converted into a CP-net, it is represented by a place of the appropriate color. After a token representing the initial value is added, the place should always contain a token. This is shown in later examples such as Figure 3.9. The following equation represents adding a variable to a specification $P_s \in P_S$:

$$\exists p_v \in P_s \wedge |C(p_v)| = 0 \tag{3.14}$$

Place p_v represents the variable and is a member of the set of places for the set P_s. The place should not contain a token until one has been explicitly added. This is done in order to indicate that the variable the place represents has not yet been initialized. Initialization is represented by:

$$\exists a = (t_{i-1}, p_v) \in A_s : Type(E(a)) = C(p_v) \wedge Type(Vars(E(a))) \subseteq \Sigma \tag{3.15}$$

Transition t_{i-1} represents a variable declaration for statement i in the specification P_s. The expression of the arc is one of the values in Table 3.15 and is dependent upon the type of variable.

3.5.3.2 Collection Data Types

A collection data type may consist of zero or more elements where an element can either be a primitive data type or another collection data type. There are two collection data types in CAPSL: *sequences* and *sets*. A sequence corresponds to an array in languages such as C/C++. The set collection is based on set theory.

Places of colored Petri nets contain sets of colors. A place could therefore represent a collection of primitive data types. However, this would not capture the ordinal numbers of a sequence or represent collections within a collection. To solve those issues, a product is used; a product is a combination of colors that may include other products as well.

The functions of collections, such as unions and differences, are discussed in §3.3.4 and built into the colored Petri net engine. This allows for the functions to be used as arc expressions. An example of a colored Petri net representing collections is given in §5.2.3. The default value of a collection is an empty sequence or set.

3.5.4 Operators

This section examines the conversion of the operators of the CAPS specification language into a colored Petri net. The operators include assignment, arithmetic, comparison, and boolean. The conversion of these operators into a colored Petri net is discussed below.

3.5.4.1 Assignment Operator

The purpose of the assignment operator (see §3.3.5) is to assign a value to a variable. That value may be another variable or a constant. In terms of CP-nets, the place representing the variable must have the token that represents the current value of the variable replaced with a token that represents the new value. An example of a specification demonstrating assignment is:

```
integer i
integer j = 2

start specification ‘‘Example 3.16’’
  i = j
end specification
```

Figure 3.8 shows the resulting CP-net:

As discussed in §3.5.3, the transition between places *State 0* and *State 1* and the transition between

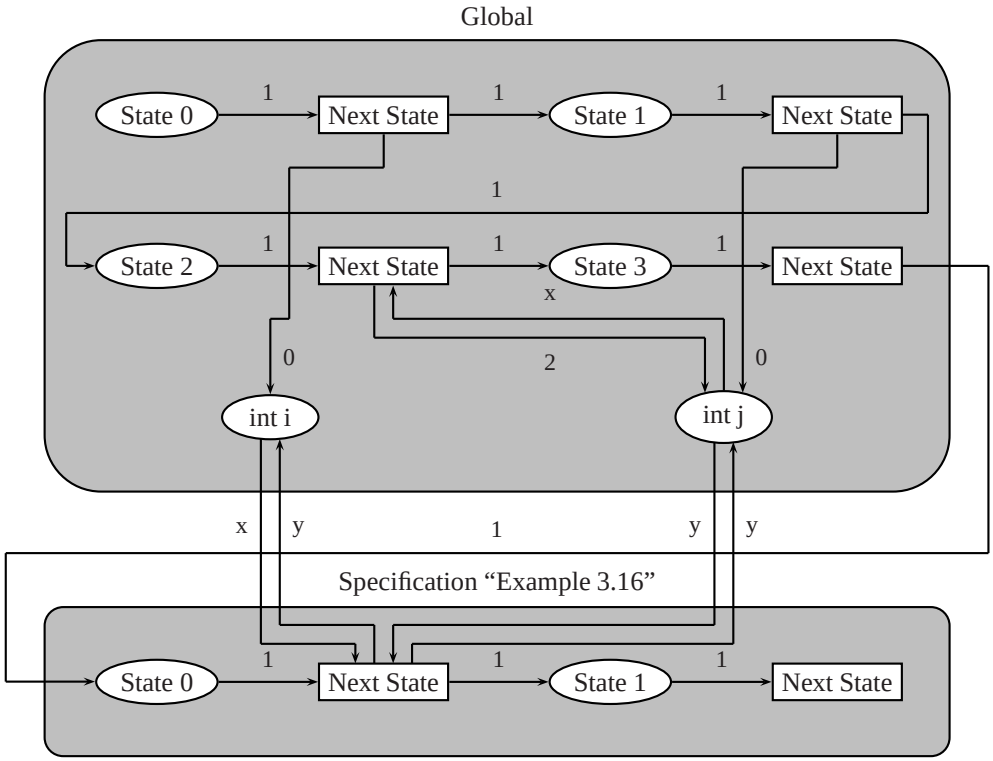

Figure 3.8: Colored Petri net demonstrating assignment to a variable.

State 1 and *State 2* of the global box are responsible for assigning initial values to the places that represent the variables `i` and `j`. The arc with the *x* from the place *int j* to the transition between places *State 2* and *State 3* retrieves the token that represents the value of `j`. The arc expression *x* is a variable and will be bound to a token in *int j* (a place representing a primitive data type will have at most one token). This has the effect of removing the token. The arc with a 2 adds a token to *int j* that represents the new value of the variable `j`. This transition models the statement `j = 2`. Note that the statement `integer j = 2` is broken into both a variable declaration and assignment. This models the internal behavior of the specification language. The arc that represents removing the value of the variable is:

$$8\exists a_1 = (p_v, t_{i-1}) \in A_G : Type(E(a_1)) = C(p_v) \wedge Type(Vars(E(a_1))) \subseteq \Sigma \tag{3.16}$$

Transition t_{i-1} represents statement i of the model. The arc expression $E(a_1)$ is a variable that will remove the token of place p_v. The arc that represents adding a value to a variable is:

$$\exists a_2 = (t_{i-1}, p_v) \in A_G : Type(E(a_2)) = C(p_v) \wedge Type(Vars(E(a_2))) \subseteq \Sigma \tag{3.17}$$

The expression $E(a_2)$ of the arc can be a constant, variable, or compound expression such as an arithmetic operation.

3.5.4.2 Arithmetic Operators

Arc expressions of colored Petri nets allow for arithmetic of both constants and variables. In the example below, the conversion of arithmetic statements dealing with constants, global variables, and local variables into a CP-net are examined:

```
integer i
integer j = 2

start specification ‘‘Example 3.17’’
  i = j + 3
  specification ‘‘Spec A’’
end specification

start specification ‘‘Spec A’’
 integer k = i + j
end specification
```

The colored Petri net representation of the code is shown in Figure 3.9:

There are two arithmetic statements in the example. The first adds a constant to a global variable

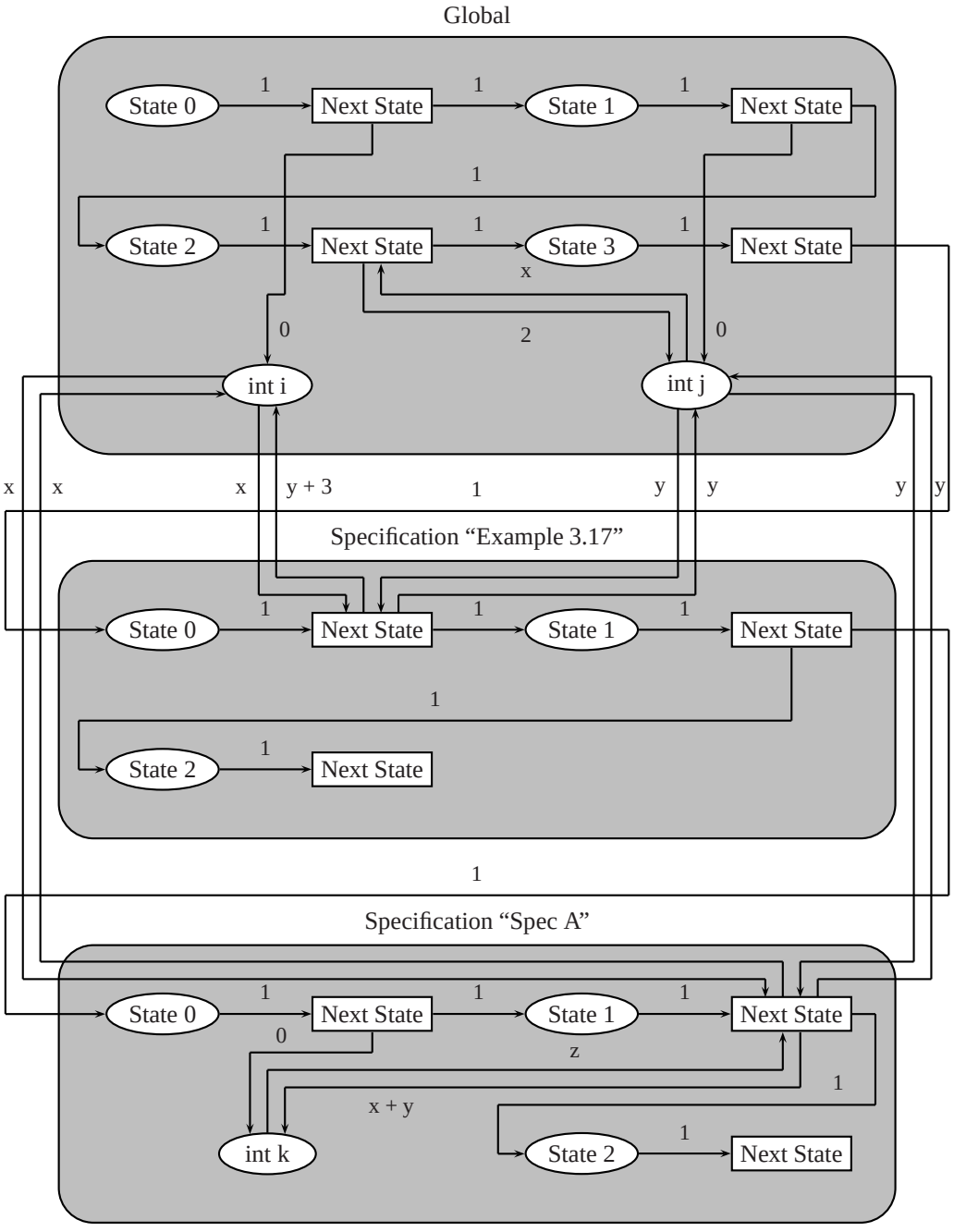

Figure 3.9: Colored Petri net example of arithmetic involving constants and variables.

and assigns the result to a global variable. The second adds two global variables and assigns the result to a local variable.

In the CP-net of Figure 3.9, the first arithmetic statement is represented by the transition between places *State 0* and *State 1* in the specification "Example 3.17" box. This transition has two arcs for each of the places representing the variables `i` and `j`. The arc with the *y* from place *int j* to the transition retrieves the token that represents the value of `j`. The arc with *y* from the transition to *int j* adds back to the place a token with the same value as the one removed. If this was not done, it would represent the value of `j` being erased after the first retrieval. The arc with *x* from *int i* removes the token representing the value of `i`, and the arc with *y + 3* to *int i* adds 3 to the representation of the value of `j`. The result is assigned to the place *int i*.

The transition between places *State 1* and *State 2* in the specification "Spec A" box represents the second arithmetic statement. This transition has two arcs for each of the places representing the `i`, `j`, and `k` variables where `k` is a local variable of the specification. The place representing `k` was initialized by the first transition in the specification "Spec A" box. The second arithmetic statement contains only variables, so the arcs from *int i* and *int j* to the transition retrieves the tokens that represent the values of the variables. The arcs from the transition to *int i* and *int j* insure that *int i* and *int j* are unchanged after the transition fires. The arc from *int k* to the transition removes the token representing the value of `k`, and the arc to *int k* from the transition adds together the values of the tokens retrieved from *int i* and *int j*. The result is stored in *int k*.

3.5.4.3 Comparison Operators

As with the arithmetic operators, the arc expressions of colored Petri nets support comparison operators. The main differences are that arithmetic operations are for integers and reals and the result of the operation is an integer or real while comparison operators are for any color and the result is a boolean. An example of a specification structure with a comparison operator (see §3.3.7) is:

```
integer i = 0
integer j = 2

start specification ''Example 3.18''
  boolean b = i + 1 < j
end specification
```

The conversion of the specification into a CP-net is shown in Figure 3.10:

In the global box, the variables `i` and `j` are initialized (see §3.5.4) and have the values 0 and 2 assigned to them. The first transition in the specification "Example 3.18" box, initializes the local boolean variable `b`. The comparison statement to be evaluated is `i + 1 < j`; however, this is too complex for the

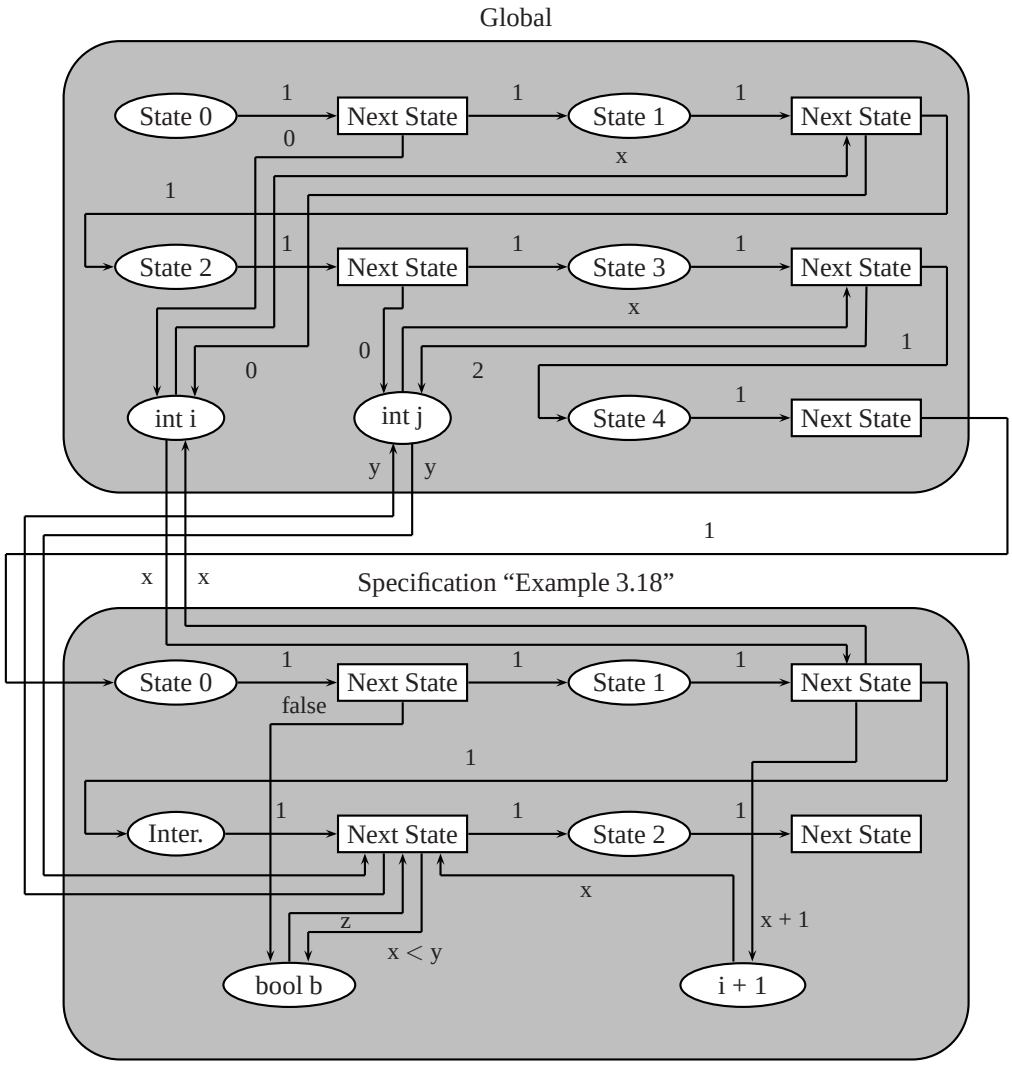

Figure 3.10: Colored Petri example involving a boolean comparison.

CP-net arc expressions; therefore, it is necessary to reduce `i + 1` into one value. This is the purpose of the transition between places *State 1* and *Inter.*. The place *i + 1* acts as an intermediate result. The transition between places *Inter.* and *State 2* compares the intermediate result of *i + 1* to the token stored in *int j* and saves the result in *bool b*. Note that when the token is retrieved from *i + 1* the place becomes empty. Intermediate results are only used one time.

3.5.4.4 Boolean Operators

The conversion of the boolean operators (see §3.3.8) for the CAPS specification language into colored Petri nets follows from the conversion of the arithmetic and comparison operators.

3.5.5 Block Statements

This section covers converting the block statements of the CAPS specification language into a colored Petri net. CAPSL includes three types of block statements: *if-else*, *when*, and *while*. The if-else statement is used for conditional execution. The when statement is used for concurrency, and the while statement is used for iteration. Conversion of the block statements into a CP-net is discussed below.

3.5.5.1 If Block

The if-else statement of CAPSL (see §3.3.9) is based upon the C/C++ if-else statement. Block statements of the specification language are similar to the specification and algorithm structure in that they can contain local variables along with other statements. An example of an if statement is the following:

```
start specification ‘‘Example 3.19’’
  integer i

  if i == 0
    i = i + 1
  end if
end specification
```

Figure 3.11 is the resulting colored Petri net:

In the figure, the first transition in the specification “Example 3.19” box initializes the place that represents the local variable `i`. The second transition moves a token to place *If 0* so that execution may begin in the if block. The first transition of the if block tests the condition of the if statement. If the condition is true, a token is moved to the place *Cond.*. If the condition is false, a token is moved to place *State 2*. Note that if the condition is true a token will be moved to *State 2* at the end of the if block. The second transition

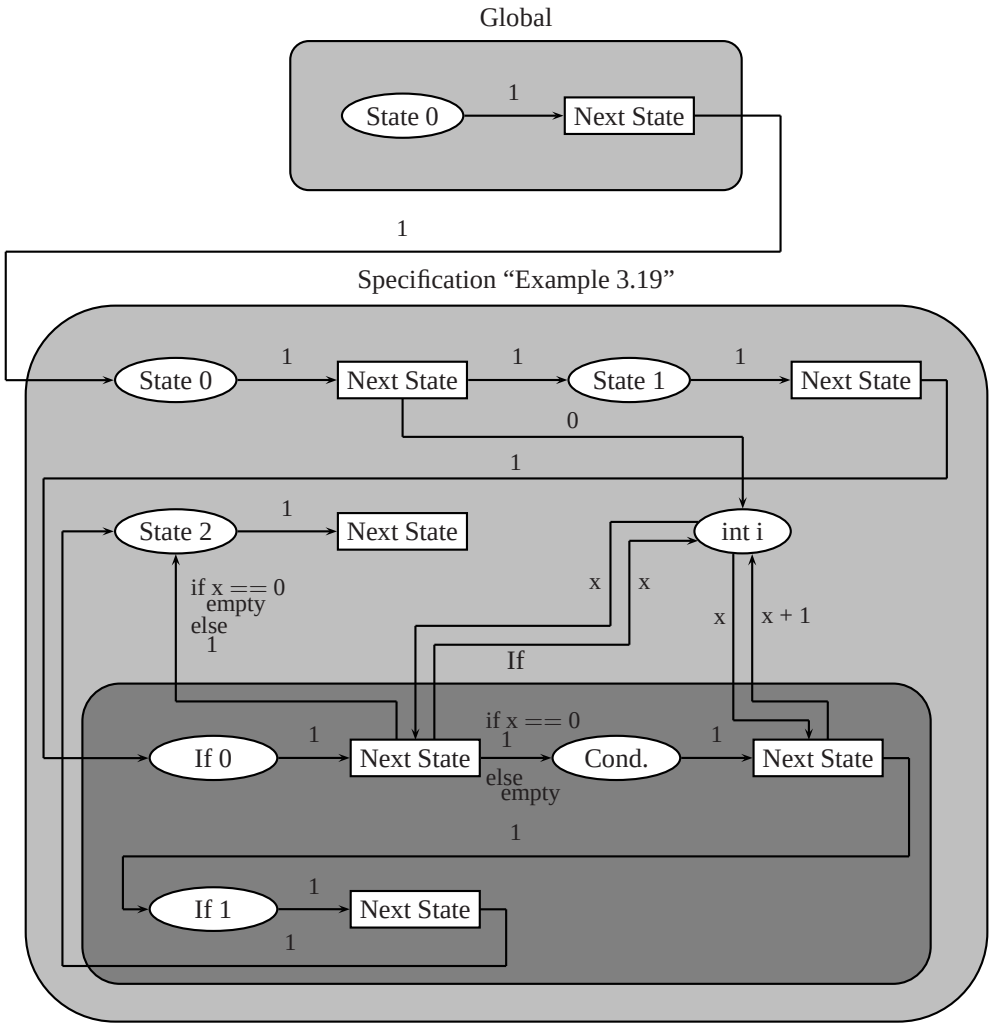

Figure 3.11: Colored Petri net example of an if statement.

in the if box represents the statement `i = i + 1`.

Let the places of the specification "Example 3.19" be represented by $P_s \in P_S$. Likewise, let T_s and A_s represent the transitions and arcs of the specification structure. Then the places, transitions, and arcs of the if statement block are defined as:

$$P_{If} \subset P_s \wedge T_{If} \subset T_s \wedge A_{If} \subset A_s \tag{3.18}$$

Every if statement block must contain at least one place and one transition with the place initially containing no colors:

$$\exists p_0 \in P_{If} \wedge |C(p_0)| = 0 \wedge \exists t_0 \in T_{If} \tag{3.19}$$

There is one arc from a transition in the specification structure to a place in the if statement block. When the transition fires, a color is moved to the if block representing execution of the if statement. The arc is defined as:

$$\exists a_1 = (t_{i-1}, p_0) \in A_s : Type(E(a_1)) = C(p_0) \wedge Type(Vars(E(a_1))) \subseteq \Sigma \tag{3.20}$$

For the transition t_{i-1}, $t_{i-1} \in T_s \wedge t_{i-1} \notin T_{If}$. The transition t_{i-1} represents the i^{th} statement of the specification and the place p_0 is the initial place of the if statement block. Additionally, there are two arcs from a transition of the if statement to a place in the specification structure:

$$\exists a_2 = (t_n, p_i) \in A_s : Type(E(a_2)) = C(p_i) \wedge Type(Vars(E(a_2))) \subseteq \Sigma \tag{3.21}$$

For the place p_i, $p_i \in P_s \wedge p_i \notin P_{If}$. The place p_i represents the i^{th} statement of the specification and $a = (t_{i-1}, p_i) \notin A_s$; therefore, there is no direct arc between transition t_{i-1} and place p_i. This ensures that execution must go through the if statement block. The transition t_n of the if block represents the final transition of the if statement. Additionally, the if block contains arcs representing the evaluation of the if condition. The arc representing a successful test of the condition is:

$$\exists a_3 = (t_{j-1}, p_j) \in A_{If} : Type(E(a_3)) = C(p_j) \wedge Type(Vars(E(a_3))) \subseteq \Sigma \tag{3.22}$$

The arc representing an unsuccessful test of the if statement condition is:

$$\exists a_4 = (t_{j-1}, p_i) \in A_s : Type(E(a_4)) = C(p_i) \wedge Type(Vars(E(a_4))) \subseteq \Sigma \tag{3.23}$$

In the equations, the place p_j represents the state *Cond.* of the if statement block. For these arcs, the expressions $E(a_3)$ and $E(a_4)$ are if-else expressions with the else expression of $E(a_3)$, and the if expression of $E(a_4)$, resulting in an empty token. As with C/C++, there may be an accompanying else block to an if statement. An example of an if-else statement is:

```
start specification ''Example 3.20''
  integer i

  if i == 0
    i = i + 1
  else
    i = i - 1
  end if
end specification
```

The colored Petri net from the conversion of the CAPSL code above is shown in Figure 3.12:

The first transition in the if block represents the condition of the if statement; however, in this case if the condition is false, a token is sent to place *Else 0*. This represents the start of execution in the else block. The last transition in the else block sends a token to place *State 2*, which represents the next state after the if-else block. The places, transitions, and arcs of the else statement block are be represented by:

$$P_{Else} \subset P_s \wedge T_{Else} \subset T_s \wedge A_{Else} \subset A_s \tag{3.24}$$

Every else block must contain at least one place and one transition with the place initially containing no colors:

$$\exists p_0 \in P_{Else} \wedge |C(p_0)| = 0 \wedge \exists t_0 \in T_{Else} \tag{3.25}$$

Equation 3.21 now applies to both the if and else statement blocks. This represents execution returning to the specification structure after the if-else statement has been executed. Additionally, Equation 3.23 becomes:

$$\exists a_4 = (t_{j-1}, p_0) \in A_s : Type(E(a_4)) = C(p_i) \wedge Type(Vars(E(a_4))) \subseteq \Sigma \tag{3.26}$$

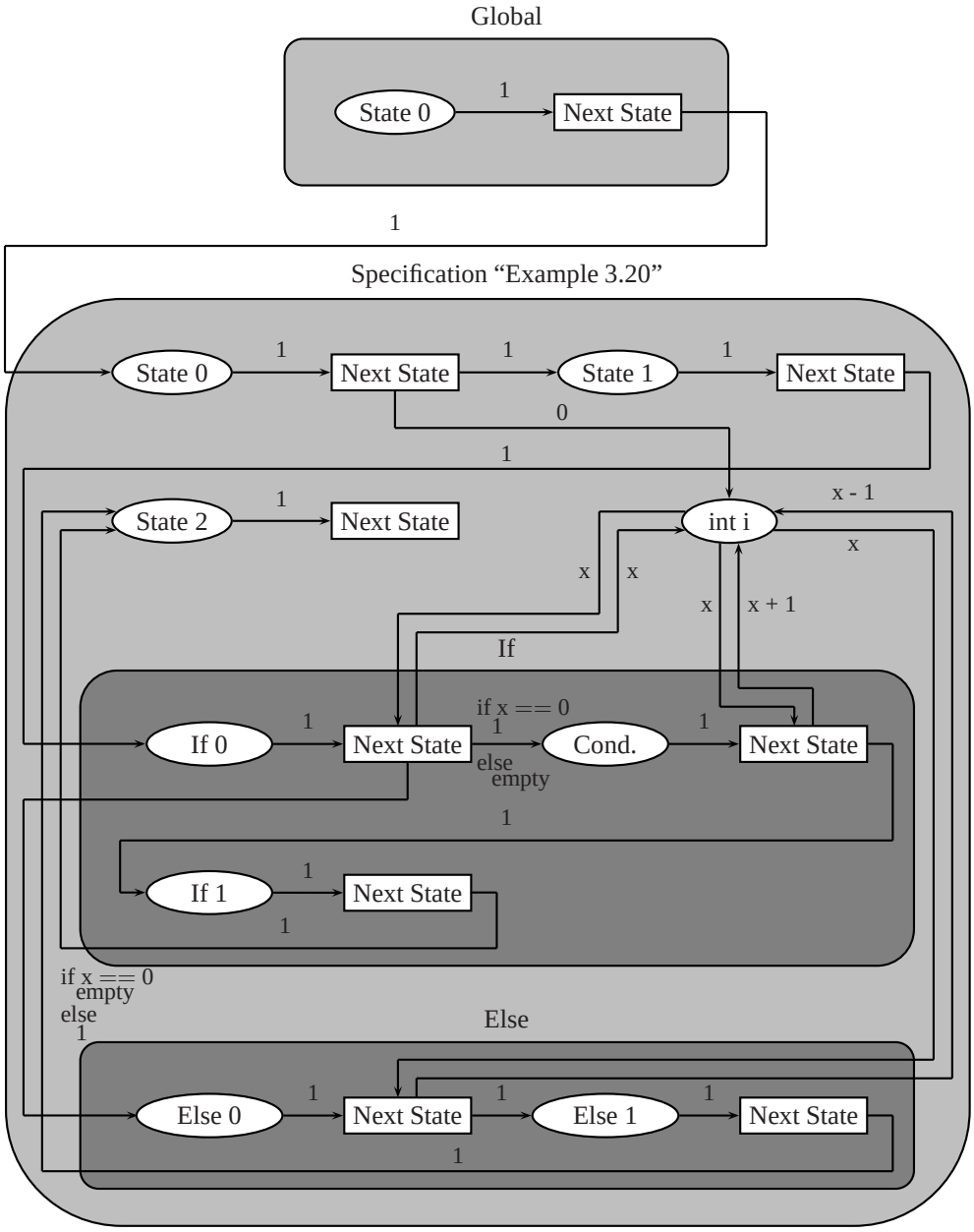

Figure 3.12: Colored Petri net example of an if-else statement.

Place p_0 represents the initial place of the else block.

3.5.5.2 When Block

The when statement (see §3.3.10) is similar to the if statement with the exception that the when statement will halt execution until its condition becomes true. An example of using a when statement is the following:

```
start specification ‘‘Example 3.21’’
  integer i

  when i == 0
    i = i + 1
  end when
end specification
```

Figure 3.13 is the resulting colored Petri net:

A when statement is designed for concurrency; therefore, there should be more than one specification structure during the use of a when statement (see §5.1.3). The difference in the conversion of an if and when statement is the condition. In the figure, the first transition of the when block represents the condition of the when statement. As with the if statement, if the condition is true, a token is moved to place *Cond.*. However, if the condition is false, a token is moved to place *When 0*. This results in the transition representing the condition being enabled until the condition becomes true. For the when statement, Equation 3.23 becomes:

$$\exists a_4 = (t_{j-1}, p_0) \in A_{When} : Type(E(a_4)) = C(p_0) \wedge Type(Vars(E(a_4))) \subseteq \Sigma \tag{3.27}$$

In the equation, the place p_0 represents the place *When 0* of the when statement block in Figure 3.13. The result of this arc is that if transition t_{j-1} fires and the expression of the arc evaluates to false, a token will be moved to place p_0.

3.5.5.3 While Block

The while statement block (see §3.3.11) is based on its equivalent in C/C++ and is the only iterative statement block in the CAPS specification language. The decision for this is based upon syntax. The syntax of the while statement matches the if and when statements. Other loop statements, such as for and do-until, have a different syntax and do not provide any extra functionality; therefore, to make the language easier to learn for novices, the other iterative blocks are not included. An example of a while statement is:

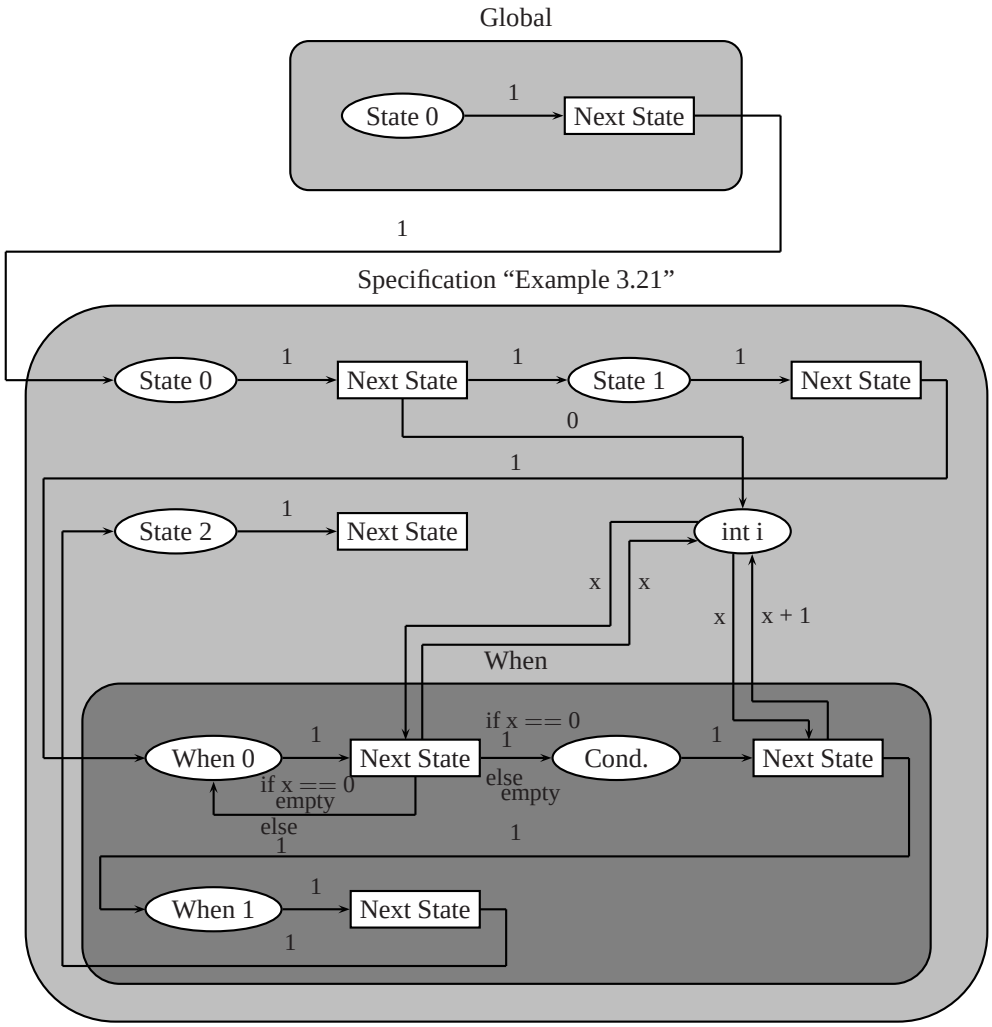

Figure 3.13: Colored Petri net example of a when statement.

```
start specification ‘‘Example 3.22’’
  integer i

  while i >= 0 and i < 10
    i = i + 1
  end while
end specification
```

Figure 3.14 is the converted colored Petri net of the specification:

The first transition in the specification “Example 3.22” box initializes the place that represents the variable `i`. The second transition sends a token to the place *While 0* in the while box. The while box represents the second statement of the specification. Testing of the condition for the while statement is the same as for the if statement. In the example, the condition is `i >= 0 and i < 10`. As was seen in §3.5.4, a condition of this type will require intermediate results for `i >= 0` and `i < 10`. The while box contains a place for each of these results and when the transitions before the *Inter.* places are fired, the intermediate calculations are made.

The condition of the while statement block is calculated by the transition before the place *Cond.*. If the condition is true, a token is sent to place *Cond.*. If the condition is false, a token is sent to place *State 2*, and execution of the the while statement block is finished. The transition in the while box after the place *Cond.* is responsible for calculating `i = i + 1`. The last transition of the while box sends a token back to place *While 0* so that the condition of the while loop can be retested. Equation 3.21 therefore becomes:

$$\exists a_2 = (t_n, p_0) \in A_{While} : Type(E(a_2)) = C(p_0) \wedge Type(Vars(E(a_2))) \subseteq \Sigma \tag{3.28}$$

The transition t_n represents the final transition, and p_0 represents the initial place, of the while statement block. Additionally, there can be break and continue statements. The break statement is represented by:

$$\exists a_5 = (t_{j-1}, p_i) \in A_s : Type(E(a_5)) = C(p_i) \wedge Type(Vars(E(a_5))) \subseteq \Sigma \tag{3.29}$$

In the equation, the transition t_{j-1} represents statement j in the while block where the break is encountered. The place p_i represents the ith statement of the specification; therefore, if transition t_{j-1} is fired, a token should be moved to the place p_i in the specification structure. A continue statement is

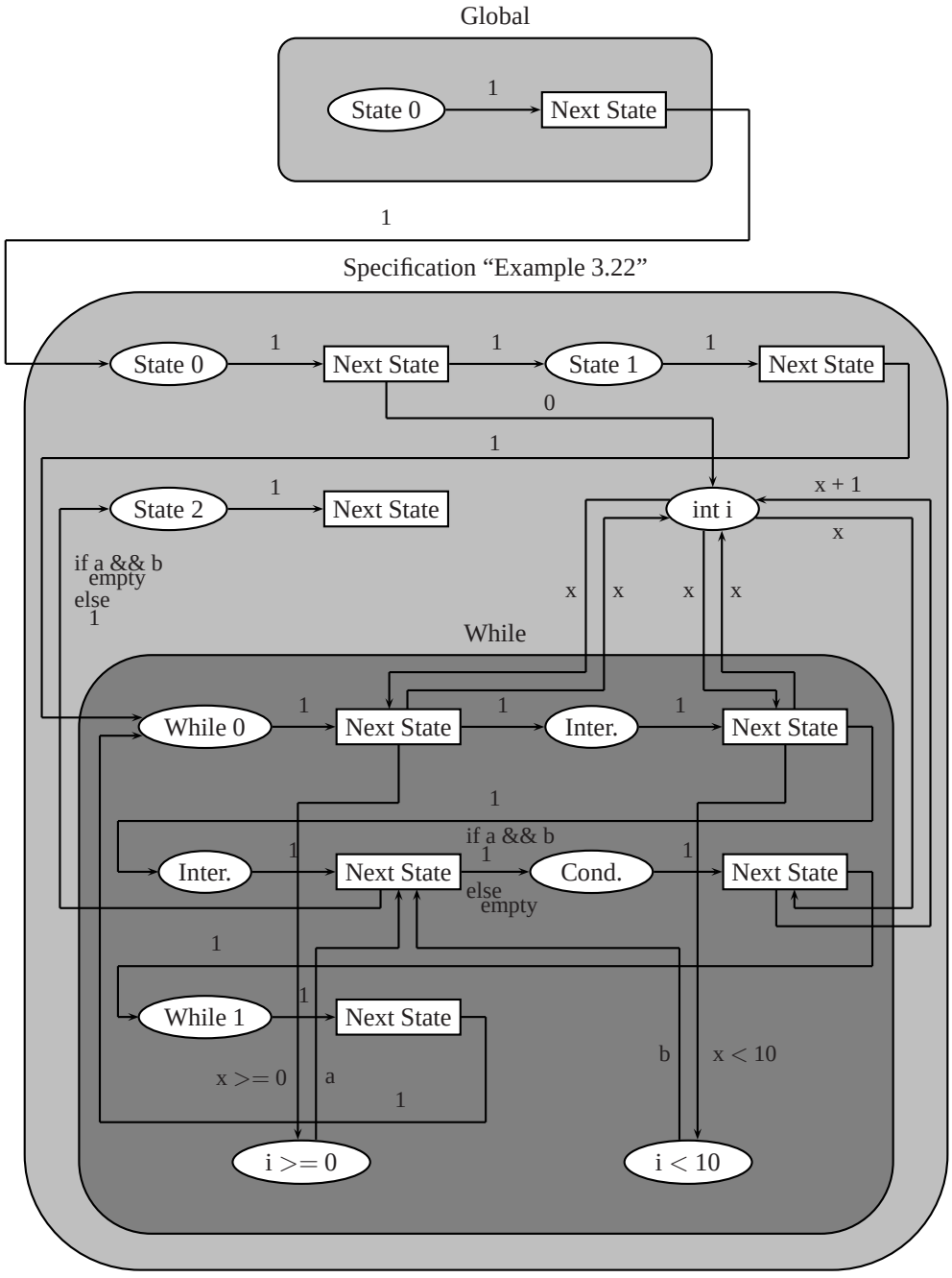

Figure 3.14: Colored Petri net example of a while statement.

represented by:

$$\exists a_6 = (t_{j-1}, p_0) \in A_{While} : Type(E(a_6)) = C(p_0) \wedge Type(Vars(E(a_6))) \subseteq \Sigma \tag{3.30}$$

This is similar to Equation 3.28 with the exception that transition t_{j-1} represents statement j of the while block where the continue statement was encountered.

3.6 Summary

CAPSL, the specification language of CAPS, is a very high-level language designed for concurrent specifications. It is one component of an automatic programming system; however, on its own, CAPSL provides a functional language that can be used to solve a variety of problems. When used as part of CAPS, the specifications can be converted into colored Petri nets and formally verified. This is discussed further in §3.3.14. Additionally, a natural language processing system is provided to convert English phrases into CAPSL structures and statements.

The CAPS specification language was designed with two primary goals: concurrency and formal verification. However, unlike the automatic programming systems discussed in §2.1.1, CAPS achieves formal verification through conversion into a colored Petri net. There are several advantages to this approach. One advantage is that CP-nets have a visual representation and this allows for an additional method of validation. CAPS CP-net models can fire transitions at specified intervals, the model can be paused, or the model can even be run in reverse. Another advantage over many of the systems listed is that the specifications can be directly executed without needing to convert them into a CP-net or other form.

A disadvantage of automatically converting CAPSL code into colored Petri nets is that it results in CP-nets larger than strictly necessary (see Chapter 5). However, the extra information allows for a mapping from the CP-nets to the specification and algorithm structures – something that is necessary if using the CP-nets for validation. In Chapter 4, formal verification of CAPSL code, and the converted colored Petri net, is discussed.

Chapter 4

Formal Verification of Specifications

The focus of this chapter is on the formal verification of CAPSL code. As can be seen from Figure 4.1, the specifications are first converted into a colored Petri net (see Section 3.5). Once in the form of a CP-net, formal verification can be performed on the specifications.

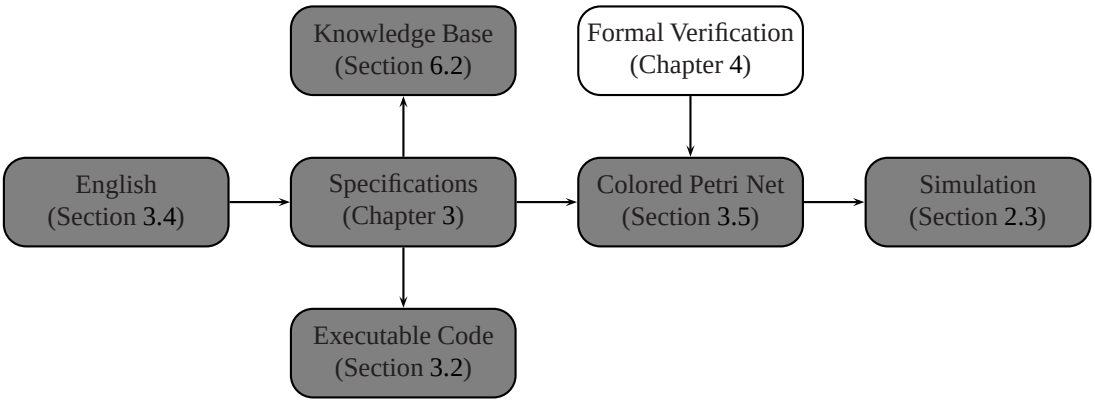

Figure 4.1: Graphical representation of CAPS focusing on formal verification of specifications.

In Section 4.1, some background information is provided about formal verification of specifications. Section 4.2 discusses the formal verification properties of similar systems, Section 4.3 details the formal verification properties of colored Petri nets, and Section 4.4 describes the formal verification properties of the CAPS specification language. A summary of the chapter is provided in Section 4.5.

4.1 Background

Historically, formal verification of specifications has been achieved through two methods: theorem proving and model checking. A theorem prover can either be used to construct a specification, known as constructive theorem proving [32], or to prove properties of a specification [30]. The input language of a theorem prover is typically first-order predicate calculus (FOPC); however, the issue of time is difficult to model in FOPC, and with timing being an important issue in many applications, temporal logic is often used in the place of FOPC. Temporal logic can be verified using a model checker [16], or in some cases, a theorem prover [79].

4.2 Formal Verification of Other Systems

In this section, the formal verification properties of two other systems are discussed: SpecWare and SMV. SpecWare is a deductive synthesis system and the formal verification of its specifications is described in §4.2.1. SMV is based on temporal logic and is used for modeling concurrent specifications. The formal verification of its specifications is discussed in §4.2.2.

4.2.1 SpecWare

SpecWare, developed by the Kestrel Institute, is a modern automatic programming system that falls within the class of deductive synthesis [53]. The input language of SpecWare is known as MetaSlang and is based on higher-order logics. Through specification transformation, a MetaSlang specification can be converted into a Lisp application or code.

The formal verification of MetaSlang specifications is achieved through external provers. SNARK (SRI's New Automated Reasoning Kit) is one of the possible external theorem provers and the one most closely integrated with SpecWare [79]. SNARK is capable of creating proofs for both first-order predicate calculus and temporal logic; however, the combination of SNARK with SpecWare is recent. Some simple theorems can not yet be proved without intervention from the user in terms of adding lemmas and hypotheses. An example of a call to the SNARK theorem prover from within a MetaSlang specification is:

```
prove prop1 in Example1
  with Snark
  using hypothesis1
  options "(use-paramodulation t)
           (use-resolution nil)
```

```
        (use-hyperresolution t)"
```

In the example above, the SNARK theorem prover is invoked to prove `prop1` using the hypothesis `hypothesis1`. The options specify the manner in which SNARK will try to prove `prop1`.

4.2.2 SMV

SMV is a tool for the verification of concurrent specifications [54]. Unlike SpecWare (see §4.2.1), SMV is not an automatic programming system. Its primary purpose is for the verification of hardware designs. The input language of SMV is based on the temporal logic CTL. Thus a user of SMV can reason about both synchronous and asynchronous systems.

Formal verification of an SMV specification is achieved through a model checker. The model checker examines all possible states of a model. Many model checkers include methods for reducing the state space; however, the verification of certain models may still remain intractable. SMV is capable of verifying a model with respect to the following properties: safety, liveness, fairness, and deadlock freedom. An example of using verification in SMV is:

```
forall(i = MEMORY)
{
  memoryProtection[i] :  assert G ((freeList[i].size +
     allocatedList[i].size) = 1);
}
prove memoryProtection;
```

In this example, the addresses of free and allocated memory lists are compared to test that memory is neither lost nor gained in the system.

4.3 Formal Verification of Colored Petri Nets

Five formal verification properties are examined in this section. These properties are *reachability* (§4.3.3), *boundedness* (§4.3.4), *home* (§4.3.5), *liveness* (§4.3.6), and *fairness* (§4.3.6). In order to prove these properties, an occurrence graph (§4.3.1) and a strongly connected component graph (SCC-graph) (§4.3.2) are used. The formal verification properties and graphs in this section follow the definitions given in [42] and [43].

In the simulation of a colored Petri net, if there is a set of enabled bindings, one binding is chosen at random. In the formal verification of a colored Petri net, each binding is chosen so that every possible marking that can be reached from the initial marking will be considered.

The properties described below are built into CAPS to allow for automatic formal verification of a CP-net. However, the complexity of many of the properties, including constructing the reachability set of an initial marking, have, in the worst case, an unbounded time complexity. This is due to some CP-nets having a reachability set that is not bounded. In general for CAPS, the reachability set of a colored Petri net will be exponential. CAPS allows for the user to set a limit on the size of the reachability set. This allows a subset of the true reachability set to be examined. In many cases, this can be useful even if the examined reachability set is not complete.

4.3.1 Occurrence Graph

An occurrence graph, also known as an O-graph or as a reachability set, is a directed graph such that the following properties from [43] hold:

$$V = [M_0 > \tag{4.1}$$

$$A = (M_1, b, M_2) \in VxBExV | M_1[b > M_2 \tag{4.2}$$

$$\forall a = (M_1, b, M_2) \in A : N(a) = (M_1, M_2) \tag{4.3}$$

The first property states that the set of vertices of the O-graph are the reachable states of the CP-net from its initial marking M_0. The second property defines the arcs of the O-graph. There is an arc for the O-graph between two vertices if for the markings that correspond to those vertices there is a binding element from the first marking to the second.

The algorithm below is for the creation of the occurrence graph and comes from [43].

The Waiting set in the algorithm contains the nodes, in actuality markings, that have not been processed. The Node function takes a marking and adds that marking to the Waiting set. The Next function takes a marking and calculates all of the markings that can be reached in one step from its argument, and the Arc function adds an arc to the O-graph between two markings.

Unfortunately, the algorithm can have an unbounded time and space complexity. This is due to CP-nets having, in the worst case, an unbounded reachability set. Improvements to this algorithm have been made for certain classes of graphs and similar algorithms for Petri nets have been developed to eliminate reacha-

Algorithm 4.1 Algorithm to create an occurrence graph.

```
Waiting := ∅
Node(M_0)
repeat
   select a node M_1 ∈ Waiting
   for all (b, M_2) ∈ Next(M_1) do
      Node(M_2)
      Arc(M_1, b, M_2)
   end for
   Waiting := Waiting - M_1
until Waiting = ∅
```

bility sets that are not bounded, but those algorithms result in a loss of information. Therefore, occurrence graphs for the CAPS system include limits on the amount of vertices the O-graph may contain. This would allow an occurrence graph to be constructed for n nodes where n is less than the size of the full occurrence graph.

4.3.2 Strongly Connected Component Graph

After the occurrence graph has been created, a strongly connected component graph (SCC-graph) is contructed. The SCC-graph is used in the formal verification of the colored Petri net. The formal verification properties that use the SCC-graph are home, liveness, and fairness. Reachability and boundedness use the occurrence graph. The strongly connected properties for the set $V^* \subseteq V$, from [43], are as follows:

$$\forall (v_1, v_2) \in V^* \times V^* : FDP(v_1, v_2) \neq \emptyset \tag{4.4}$$

$$\forall V' \subseteq V : (V' = strongly - connected \wedge V^* \subseteq V') \Rightarrow V^* = V' \tag{4.5}$$

In the first property, FDP stands for finite directed path. The property simply states that between every two vertices in a strongly connected component V^*, there must exist a finite directed path. Therefore, it is possible to get from any one vertex in a strongly connected component to every other vertex in that same component.

The second property states that if V^* is a strongly connected component and is a subset of another strongly connected component $V' \subseteq V$, then V^* must be the same as V'. Thus, a strongly connected component will be maximal containing all of the vertices in V such that it is strongly connected.

The SCC-graph is the set containing all of the strongly connected components of V. An algorithm

for constructing the SCC-graph is given in Algorithm 4.2.

Algorithm 4.2 Algorithm to create an SCC-graph.

```
while V ≠ ∅ do
  v ∈ V
  for all V_c ∈ SCC do
    for all v_x ∈ V_c do
      if FDP(v, v_x) ∧ FDP(v_x, v) then
        V_c := V_c ∪ v
        V := V − v
      end if
    end for
  end for
  if v ∈ V then
    SCC(v)
    V := V − v
  end if
end while
```

In the algorithm, the function FDP is used to determine if there is a finite directed path between two vertices. In a strongly connected component, there must be a finite path between each vertex of the component.

4.3.3 Reachability

Given a marking M' of a CP-net, the reachable markings from M' are $[M'>$. Reachability is concerned with the question of given another marking $M'' \in \mathbb{M}$, is $M'' \in [M'>$. Thus, the interest is if there is a finite number of steps such that $M'[*> M''$ where $*$ represents a set of steps of unknown length. In general, the concern is not only answer to this question but the set of steps as well if they exist.

For the formal verification, assume that M' is the initial marking M_0. Then for any marking M_1, $M_1 \in [M_0>$ if M_1 is a node in the occurrence graph. This follows from the occurrence graph property $V = [M_0>$ where V is the set of vertices of the occurrence graph. Then determine if $\exists M_1 : M_1 = M''$. During this search, the steps taken through the occurrence graph can be recorded and returned if a match is found.

4.3.4 Boundedness

Boundedness is used to determine upper and lower bounds on tokens. The bounds can be on either a set of possible tokens over all places or on the tokens of a specific place. Let us first consider bounds on

a set of possible tokens covering all of the places in a CP-net. TE is the set of possible tokens. Given a set $W \subseteq TE$, there is an upper bound for W iff:

$$\forall M \in [M_0>: |(M|W)| \leq n \tag{4.6}$$

where $n \in \mathbb{N}$. If an upper bound exists, W is bounded. W has a lower bound iff:

$$\forall M \in [M_0>: |(M|W)| \geq n \tag{4.7}$$

By the definition of places in a CP-net, all sets will have a lower bound. The simplest bound, and one which is of little use, of course being 0. The definitions for the bounds of specific places in the CP-net differ from that given in [42]. The reason is that CAPS is not using hierarchical CP-nets and thus the definition needed to be changed. The bounds for a place $p \in P$ are therefore:

$$\forall M \in [M_0>: |(M(p)|W)| \leq n \tag{4.8}$$

for the upper bound and

$$\forall M \in [M_0>: |(M(p)|W)| \geq n \tag{4.9}$$

for the lower bound. The calculation of the bounds can be done during the construction of the occurrence graph. Thus a separate pass through the occurrence or SCC-graph is not necessary.

4.3.5 Home

A marking $M \in \mathbb{M}$ is considered to be a home marking if from any reachable marking, one can always return to M. A marking M is considered to be a home marking iff:

$$\forall M' \in [M_0>: M \in [M'> \tag{4.10}$$

In other words, from the initial marking M_0, the marking M must always be reachable from the current marking M'. However, it is not necessary that a home marking will ever be reached since a CP-net is non-deterministic.

A set of markings $X \subseteq \mathbb{M}$ may also be considered. Such a set is said to be a home space iff:

$$\forall M' \in [M_0>: X \cap [M'> \neq \emptyset \tag{4.11}$$

Instead of being able to always reach one home marking from the current marking M', at least one of the markings in the home space must be reachable. As with a home marking, it is possible that a marking in the home space may never be reached.

In order to determine if M is a home marking or X is a home space, the components of the SCC-graph are searched. Iff every component contains M, then is M a home marking. Likewise, iff every component contains a marking $M'' \in X$, then is X a home space.

4.3.6 Liveness

Given a marking $M \in \mathbb{M}$, and a set $Y \subseteq BE$ of binding elements, the question of liveness concerns the possibility of binding elements becoming enabled. Much as home markings, liveness does not guarantee that any member of a set Y of binding elements will ever be enabled, only that they can be enabled.

A marking M is said to be dead iff:

$$\forall y \in BE : \neg M[y> \tag{4.12}$$

This means that given the marking M, there does not exist an enabled binding. When a CP-net reaches this point during a simulation, it must come to a halt.

It can be determined if a marking M is dead from the occurrence graph. If the node representing M in the occurrence graph is a terminal node, that is, it does not have any output arcs, then the marking M is dead.

The set Y is said to be dead in the marking M iff:

$$\forall M' \in [M> \forall y \in Y : \neg M'[y> \tag{4.13}$$

Notice that Y is only dead iff for all of the reachable markings from M, none of those markings have an enabled binding element contained in the set Y.

In terms regarding the SCC-graph, the set of binding elements Y is dead in a marking M iff from the connected component containing M in the SCC-graph the components that are accessible from that

component have no markings in which a binding element on one of their output arcs is from the set Y.

Y is said to be live iff:

$$\forall M' \in [M_0> \exists M'' \in [M'> \exists y \in Y : M''[y> \tag{4.14}$$

This statement is interpreted as meaning that for all of the reachable markings in the CP-net, there will always exist a reachable marking that has an enabled binding element from the set Y.

The terminal components in the SCC-graph are used to determine liveness. The set of binding elements Y is live iff every terminal component in the SCC-graph has a marking M such that an output arc of M has as its binding element an element from the set Y.

4.3.7 Fairness

Fairness is used to determine the number of times that a binding occurs. Let set $Z \subseteq BE$. Then set Z is said to be impartial iff:

$$OC_Z(\sigma) = \infty \tag{4.15}$$

In the equation above, $OC_Z(\sigma)$ represents the number of occurrences of Z for the sequence σ. The sequence is defined as:

$$\sigma = M_1[Y_1>M_2[Y_2>M_3... \tag{4.16}$$

The sequence σ is infinite; therefore, set Z must occur an infinite number of times to be impartial. Set Z is said to be fair iff:

$$EN_Z(\sigma) = \infty \Rightarrow OC_Z(\sigma) = \infty \tag{4.17}$$

Thus, if there are infinite enablings, there are infinite occurrences. And last, set Z is said to be just iff:

$$\forall i \in \mathbb{N}_+ : [EN_{Z,i}(\sigma) \neq 0 \Rightarrow \exists k \geq i : [EN_{Z,k}(\sigma) = 0 \vee OC_{Z,k}(\sigma) \neq 0]] \tag{4.18}$$

The statement is interpreted to mean that if there is a persistent enabling, then there is an occurrence.

This ensures that the binding cannot only occur but will occur.

4.4 Formal Verification in CAPS

Section 3.5 discussed converting CAPSL code into a colored Petri net (CP-net). Once the conversion has occurred, the CP-net can be formally verified. CAPSL includes several statements that are used by CAPS to construct markings and bindings (see §3.3.14). The formal verification properties of CAPS include reachability (§4.4.1) and boundedness (§4.4.2). Properties to be added later include home, liveness, and fairness (§4.4.3).

4.4.1 Reachability

Reachability in colored Petri nets is a matter of searching the occurrence graph for a particular marking M'' (see §4.3.3). However, the reachability set may be exponential or unbounded. If $M'' \in [M'>$ where M' is the initial marking of the colored Petri net, the marking M'' is said to be reachable. CAPSL includes a `reachable` statement that is used by CAPS to automatically construct the marking M''. An example of using the statement is:

```
integer i = 0

start specification ‘‘Example 4.1’’
  i = 2

  reachable i == 2
end specification
```

Figure 4.2 shows the colored Petri net representation of the specification:

The occurrence graph of the colored Petri net in Figure 4.2 contains seven nodes. Three nodes are for each of the states in the global scope. *State 0* is the initial state of the global scope, *State 1* represents the declaration of `integer i`, and *State 2* represents `i = 0`. As execution moves through the colored Petri net, each of these states will contain the color 1. Thus there are three occurrence nodes for the global scope. Likewise, there are there are three occurrence nodes for the specification “Example 4.1”. *State 0* is the initial state of the specification, *State 1* represents `i = 2`, and *State 2* represents the statement `reachable i == 2`. The seventh occurrence node represents the end of the execution for the colored Petri net.

The colored Petri net engine of CAPS has been modified to find a reachability marking. The marking M'' that is automatically constructed is used to test places that represent variables of a CAPSL specification.

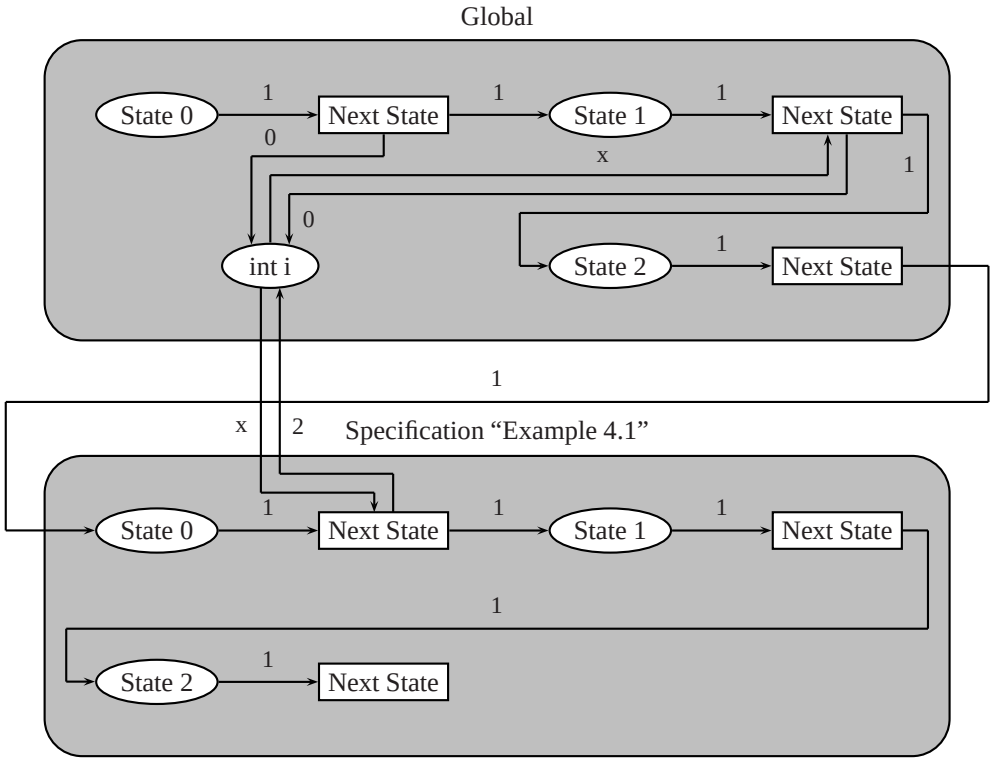

Figure 4.2: Colored Petri net of specification with a reachable statement.

Therefore, the places that do not represent variables, such as *State 0* through *State n*, are not used in determining if a marking is reachable. In the example, the occurrence nodes are all tested to determine if the place *int i* has the color 2 in any of the markings. In this case, the marking M'' is reachable.

4.4.2 Boundedness

Boundedness is used to determine the upper and lower bounds for a set of tokens over either a particular place or over all places (see §4.3.4). The colored Petri net engine of CAPS supports both methods of boundedness; however, the `boundedness` statement of CAPSL tests only the bounds for a specific place. In this case, the place represents a variable of the specification – typically a sequence or set. An example of the `boundedness` statement is:

```
sequence x

start specification ‘‘Example 4.2’’
  add x 7

  boundedness x 7
end specification
```

The colored Petri net representation of the specification is shown in Figure 4.3:

There are six nodes in the occurrence graph of the colored Petri net in Figure 4.3. Two nodes are for the states in the global scope, three nodes are for the states in the specification, and one node that represents the end of the execution for the specification. Boundedness in the example will examine the colors of the place *seq x* for all of the nodes of the occurrence graph to determine the minimum and maximum number of colors of value 7 that are in the place. As the sequence is initially empty, the lower bound will be 0. One seven is later added to the sequence; therefore, the upper bound will be 1.

As with reachability, boundedness for the colored Petri net engine has been modified. In this case, the modification allows for the search of elements that may be in a place representing a collection.

4.4.3 Home, Liveness, and Fairness

The formal verification properties reachability and boundedness utilize the occurrence graph of a colored Petri net. As mentioned in §4.4.1, some modifications had to be made to find a reachability marking of the occurrence graph. The verification properties home (see §4.3.5), liveness (see §4.3.6), and fairness (see §4.3.7) all utilize the strongly connected component graph (SCC-graph). As discussed in §4.3.2, the SCC-graph is constructed from the occurrence graph. An example of using the home property is shown in

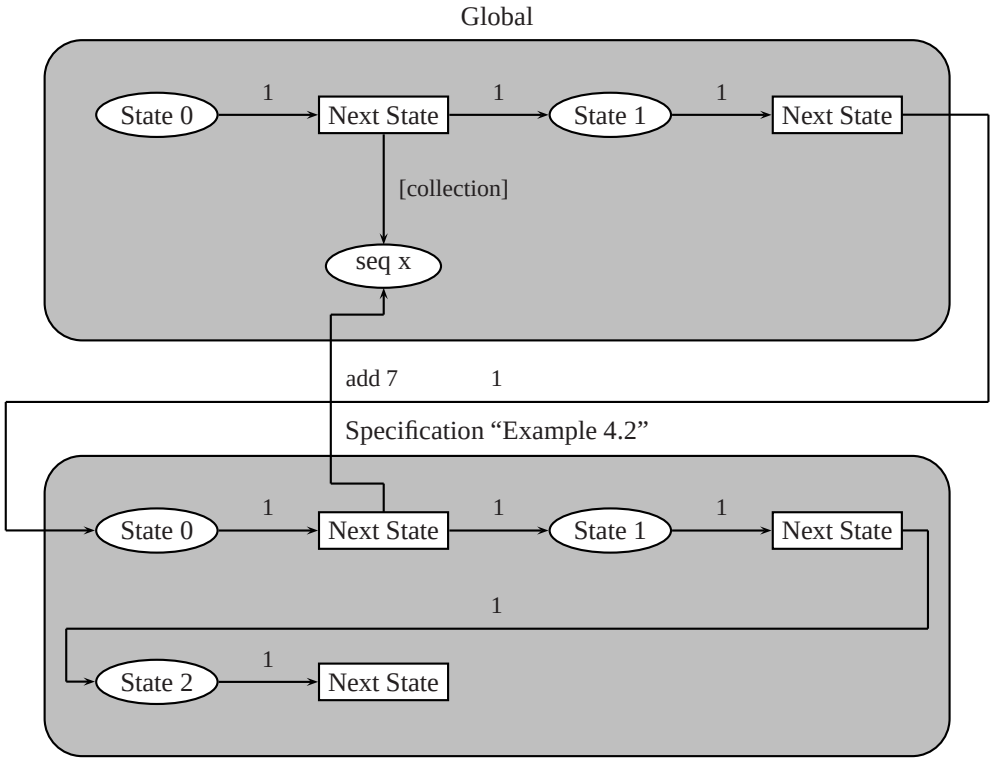

Figure 4.3: Colored Petri net of specification with a boundedness statement.

the following specification:

```
integer i = 0

start specification ‘‘Example 4.3’’
  i = 2

  home i == 2
end specification
```

Specification “Example 4.3” follows the form of specification “Example 4.1”, and as with the former specification, there are seven occurrence nodes; however, there are also seven SCC nodes. The issue is with the places representing *State 0* through *State n*. Other than places representing states in a while loop, each place representing a state will result in a new strongly connected component of size 1. The result is that `i == 2` is not considered a home marking. Therefore, in order to use the home, liveness and fairness verification properties, a modification will need to be made in the construction of the SCC-graph. This is discussed further in §6.2.2.

4.5 Summary

Formal verification of concurrent specifications is a primary goal of CAPS. For this purpose, formal verification statements have been built into CAPSL, and the colored Petri net engine is capable of constructing occurrence and SCC-graphs. At this point, the formal verification properties are limited to reachability and boundedness; however, the system can later be extended to support home, liveness, and fairness. The SCC-graph is constructed from the occurrence graph, so much of the work has already been done.

Chapter 5 provides case studies for the producer-consumer and domination number problems. With the producer-consumer problem, the concurrency of CAPS is examined, while the collections are examined with the domination number.

Chapter 5

Case Studies

This chapter presents two case studies for CAPS. The first study is presented in Section 5.1 and is concerned with the classical producer-consumer problem. The purpose of this study is to examine the concurrency of CAPS. The second study is given in Section 5.2 and examines the domination number of a tree T. The collection data types of the CAPS specification language are examined in this study. A comparison of CAPS to three other languages is given in Section 5.3, and a summary of the chapter is given in Section 5.4.

5.1 Producer-Consumer Problem

The producer-consumer problem is a classical operating system problem in which there are i producers, j buffers, and k consumers. The producer(s) add items to the buffer(s), and the consumer(s) remove items from the buffer(s).

Background information about the importance of the producer-consumer problem is given in §5.1.1. A parallel algorithm for a two producer, two buffer, and two consumer problem is given in §5.1.2. §5.1.3 contains executable programs of the algorithm presented in §5.1.2 for C (5.1.3.1) and CAPS (5.1.3.2). §5.1.4 contains the results of the study.

5.1.1 Background

The producer-consumer problem is an example of *concurrent computing*. In concurrent computing, there are multiple tasks executing and interacting with one another. More detail concerning concurrent computation is given in Section 3.2. With the producer-consumer problem, there are at least two tasks running

concurrently. The first task produces some item and places it into a buffer. The second task removes the items from the buffer.

There are many variations of the producer-consumer problem. For example, one variation may have three tasks producing items and placing them into two buffers while two other tasks remove the items from those buffers.

C. A. Petri invented Petri nets in his dissertation [64]. Initially, there was a great deal of interest in Petri nets because it was suspected by many that Petri nets might have more modeling power than Turing machines. However, a simple variation of the producer-consumer problem showed that Petri nets were actually less powerful than Turing machines. Hack showed that for a Petri net to be Turing-complete, all that was required was the addition of an *inhibitor arc* to a Place/Transition Petri net [35].

Figure 5.1 shows a Place/Transition Petri net that models a producer-consumer problem that requires an inhibitor arc. The producer-consumer variation has two producers, two buffers, and two consumers.

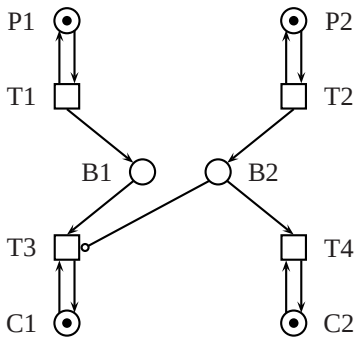

Figure 5.1: Producer-Consumer Place/Transition net using an inhibitor arc.

At the top of the figure, the places *P1* and *P2* are the producers. When transition *T1* fires, producer *P1* adds a token to buffer *B1*. Likewise, when transition *T2* fires, producer *P2* adds a token to buffer *B2*. The consumers are the places *C1* and *C2*. When transition *T4* fires, a token is removed from buffer *B2* and added to the consumer *C2*. The only constraint on the consumer *C2* is that there must be a token in buffer *B2* before transition *T4* will fire.

When transition *T3* fires, a token is removed from buffer *B1* and added to consumer *C1*. One of the constraints on this happening is that there must be a token in buffer *B1*. An additional constraint, and one that can not be modeled by the original Petri net, is that the buffer *B2* must be empty. Note that there is an inhibitor arc (has a circle on the end) from buffer *B2* to transition *T3*.

5.1.2 Algorithm

A parallel algorithm for the producer-consumer problem shown in Figure 5.1 is given in Algorithm 5.1. Items are moved in parallel from producers to buffers and from buffers to consumers.

Algorithm 5.1 Producer-consumer algorithm for two producers, two buffers, and two consumers in which one of the consumers requires that the buffer for the other consumer is empty.

```
Producer1 := 5000
Producer2 := 5000
for 1 ≤ i ≤ 4 do in parallel
  if i = 1 then
    while |Producer1| > 0 do
      move item from Producer1 to Buffer1
    end while
  end if
  if i = 2 then
    while |Producer2| > 0 do
      move item from Producer2 to Buffer2
    end while
  end if
  if i = 3 then
    while |Producer1| + |Buffer1| > 0 do
      if |Buffer1| ≠ 0 ∧ |Buffer2| = 0 then
        move item from Buffer1 to Consumer1
      end if
    end while
  end if
  if i = 4 then
    while |Producer2| + |Buffer2| > 0 do
      if |Buffer2| ≠ 0 then
        move item from Buffer2 to Consumer2
      end if
    end while
  end if
end for
```

5.1.3 Solutions

In this section, executable programs for the producer-consumer algorithm that was presented in §5.1.2 are given for C and CAPSL. Interesting portions of the code are shown and discussed while the full programs can be found in Appendix B.1. Parts of the colored Petri net from the conversion of the CAPSL solution are also shown.

5.1.3.1 C Solution

The C solution of the producer-consumer problem utilizes the POSIX (Portable Operating System Interface) thread library for multi-threading. Threads are used because several parts of the program will need to operate in parallel, and a thread uses less overhead than creating a new process. POSIX threads (pthreads) are used because the IEEE POSIX 1003.1c standard is available on, and consistent across, most Unix systems.

Four threads must be created from the initial process. Two of the threads move items from the producers to the buffers. The two other threads move items from the buffers to the consumers. The code to create the threads is:

```
pthread_create(&ptThreadID[0], NULL, &consumer1, NULL);
pthread_create(&ptThreadID[1], NULL, &consumer2, NULL);
pthread_create(&ptThreadID[2], NULL, &producer1, NULL);
pthread_create(&ptThreadID[3], NULL, &producer2, NULL);
```

The third argument of the `pthread_create` function is a function. These function arguments are used for moving the items from the producers to the buffers and from the buffers to the consumers.

Mutexes and wait/signal are used for thread synchronization to ensure that race conditions are handled, deadlock is not avoided, and cycles are not wasted. For example, in the `producer1` function, the buffer variable is locked so that an item can be added to the buffer. If the buffer were not locked, then the `consumer1` function might be removing an item from the buffer at the same time. This would result in a race condition. The code for locking buffer 1 so an item can be added is:

```
pthread_mutex_lock(&pmtMutexB1);
iBuffer1++;
pthread_mutex_unlock(&pmtMutexB1);
```

It is possible that the consumers will be forced to wait before they can remove an item from their respective buffers. Neither consumer 1 nor consumer 2 can remove an item if there is no item in their buffer. Additionally, consumer 1 can not remove an item from its buffer if there are any items in buffer 2. One way to handle this would be to use a busy wait. However, this method wastes cycles, so instead, wait/signal is used. An example of consumer 1 using a wait because its buffer is empty is:

```
pthread_mutex_lock(&pmtMutexCondB1);
pthread_cond_wait(&pctConditionB1, &pmtMutexCondB1);
pthread_mutex_unlock(&pmtMutexCondB1);
```

Once a thread waits, it must be signaled for execution to resume. Therefore, buffer 1 and 2 will signal when an item is added. Additionally consumer 2 will signal when it removes all of the items from buffer 2 (in case consumer 1 is waiting for buffer 2 to become empty). Below is code from buffer 1 signaling that it is no longer empty:

```
pthread_mutex_lock(&pmtMutexCondB1);
pthread_cond_signal(&pctConditionB1);
pthread_mutex_unlock(&pmtMutexCondB1);
```

The full C program can be found in §B.1.2. In all, the code is 212 lines.

5.1.3.2 CAPS Solution

The complexity of the C solution of the producer-consumer problem is due, in part, to the use of the POSIX library. The POSIX library works very well for the creation of multi-threaded applications; however, its use requires a level expertise that a casual programmer will not possess. The CAPS solution for the producer-consumer problem uses the built-in concurrency of CAPS (see Section 3.2). While CAPS is implemented in C++ and achieves concurrency through the use of the POSIX library, the implementation details are hidden from the user.

In the C program when a separate thread is created for execution, the code looks like the following:

```
pthread_create(&ptThreadID[0], NULL, &consumer1, NULL);
```

The first argument of the function is the `pthread`, the second is the attributes of the pthread, the third is the C function where the execution will begin, and the fourth is the arguments for the function. If there is more than one argument, the arguments will need to be placed into a data structure such as a `struct`. The function where the execution will begin will receive the arguments as a `void *` pointer and must know how to cast the pointer to retrieve the arguments. In CAPS, the code to create separate threads simply involves calling *specifications*, such as:

```
specification "Producer 1"
```

In the above statement, `specification` indicates that a specification object is to be named for execution. Following that is the name of the specification – in this case it is “Producer 1”. In the producer-consumer problem, buffers are accessed by multiple processes. Variables are used for the representation of the buffers. This sharing of memory requires that mutexes are used to prevent two threads from accessing a variable at the same time. A case of using mutexes in C to protect access to a variable is:

```
pthread_mutex_lock(&pmtMutexB1);
iBuffer1++;
pthread_mutex_unlock(&pmtMutexB1);
```

In this example, the variable `iBuffer1` is to be updated indicating that an item has been moved from consumer 1; however, it is possible that more than one thread may be accessing this variable. For example, consumer 1 may be checking to see if buffer 1 contains any items. Therefore a mutex is locked before, and

unlocked after, the variable is updated. CAPS automatically handles the locking and unlocking of mutexes, so the equivalent CAPS code is simply:

```
iBuffer1 = iBuffer1 + 1
```

Other than thread creation and mutexes, the other multi-threading issue that is seen in the C code is the use of wait/signal. Wait/signal is used when some portions of the code may need to wait before they can execute. In the producer-consumer problem, if buffer 1 is empty, or buffer 2 is not empty, then consumer 1 can not remove an item from buffer 1 and must until those conditions are not true. To wait in C, the code looks like the following:

```
pthread_mutex_lock(&pmtMutexCondB1);
pthread_cond_wait(&pctConditionB1, &pmtMutexCondB1);
pthread_mutex_unlock(&pmtMutexCondB1);
```

In addition to this code, the process will need to determine if it should wait. This requires checking the states of variables and therefore locking and unlocking mutexes other than the mutex `pmtMutexCondB1`, which is used in this case only for waiting and signaling.

CAPS uses the *when* statement (see §3.3.10) for waiting on a condition. The entire code for having consumer 1 check buffers 1 and 2 and remove an item from buffer 1 if that buffer is not empty and buffer 2 is empty is:

```
when iBuffer1 > 0 and iBuffer2 == 0
  iBuffer1 = iBuffer1 - 1
end when
```

In C, after a wait has been issued, another thread must perform a signal before the thread that is waiting resumes execution. The code for this looks like the following:

```
pthread_mutex_lock(&pmtMutexCondB1);
pthread_cond_signal(&pctConditionB1);
pthread_mutex_unlock(&pmtMutexCondB1);
```

The programmer must also determine where the signaling code should be placed and when it should be run. CAPS has signaling built into its system. CAPS records what variables are waiting on a condition, and when those variables are updated elsewhere, the signaling is handled automatically.

The full CAPS solution is in §B.1.2. The CAPSL code was written to mirror the C code; however, the CAPSL code is 68 lines – 32% the size of the C code.

5.1.3.3 CAPS Colored Petri Net Solution

Section 3.5 discusses how to convert CAPSL statements into a colored Petri net. Below, colored Petri nets are shown for portions of the CAPS solution to the producer-consumer problem with the solution given in §B.1.2. The prime specification of the solution is:

```
start specification "Producer-Consumer"
  specification "Producer 1"
  specification "Producer 2"
  specification "Consumer 1"
  specification "Consumer 2"
end specification
```

This specification calls other specifications that will move items from producers to buffers and from buffers to consumers. Each specification call results in the creation of a new thread that allows for the specifications to be run concurrently (see §3.2.1). Figure 5.2 shows the CP-net that represents the "Producer-Consumer" specification:

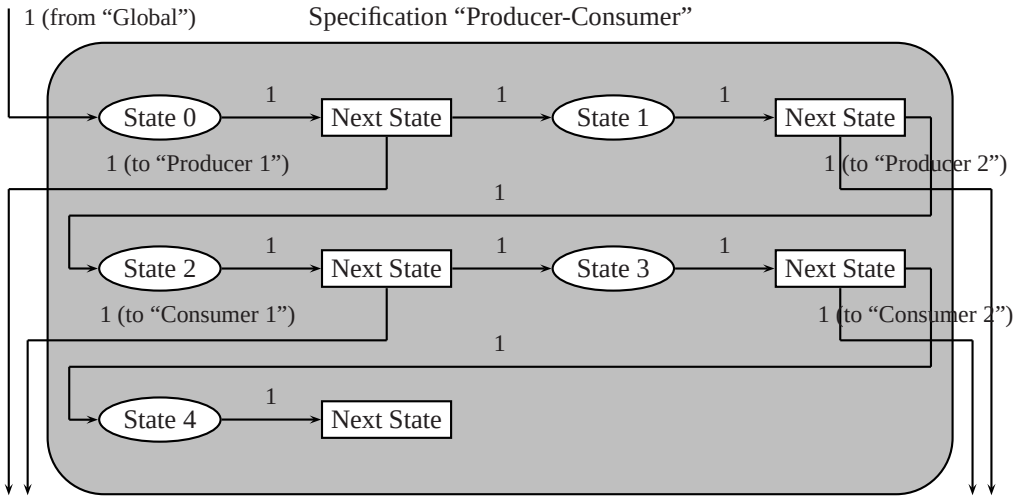

Figure 5.2: Colored Petri net of the "Producer-Consumer" specification.

Note that the CP-net representing the "Producer-Consumer" specification is nearly the same size as Figure 5.1 that models the entire producer-consumer problem. The colored Petri nets created when converting the CAPSL statements are designed to capture the structure of the specifications in addition to their behavior.

The "Producer 1" specification is the next specification to be modeled and is as follows:

```
start specification "Producer 1"
  while iProducer1 > 0
    iBuffer1 = iBuffer1 + 1
    iProducer1 = iProducer1 - 1
```

```
  end while
end specification
```

This specification is responsible for moving all of the items from the first producer to the first buffer. The colored Petri net of the specification is shown in Figure 5.3:

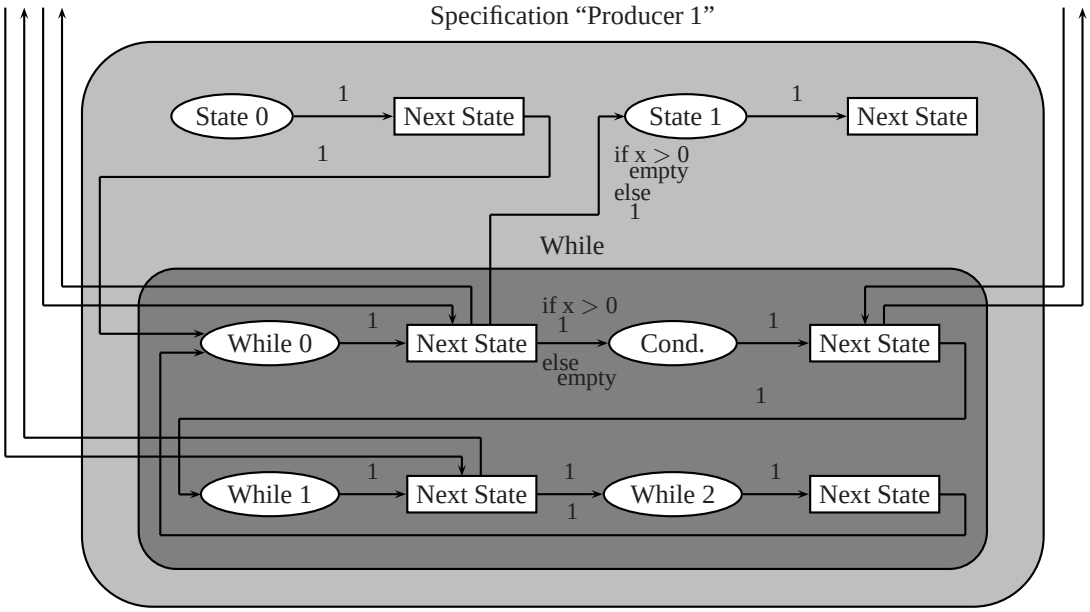

Figure 5.3: Colored Petri net of the “Producer 1” specification.

In order to keep the figure clean, some arc expressions have been omitted. The unlabeled arcs for the first and third transitions of the while box go to/from the place in the global box representing the variable `iProducer1`, and the arcs for the second transition in the while box go to/from the place representing the variable `iBuffer1`. The “Consumer 2” specification is the last specification to examine and is:

```
start specification "Consumer 2"
  while iConsumer2 < PRODUCER2_INITIAL
    when iBuffer2 > 0
      iBuffer2 = iBuffer2 - 1
    end when
    iConsumer2 = iConsumer2 + 1
  end while
end specification
```

In this specification, items are moved from the second buffer to the second consumer; however, at the same time, items are being moved by the “Producer 2” specification to the buffer from the second producer. The when statement makes certain that the buffer is not empty before an item is removed. If the buffer is empty, the “Consumer 2” specification will wait until an item is added to the buffer. Figure 5.4 contains the

CP-net representing the specification:

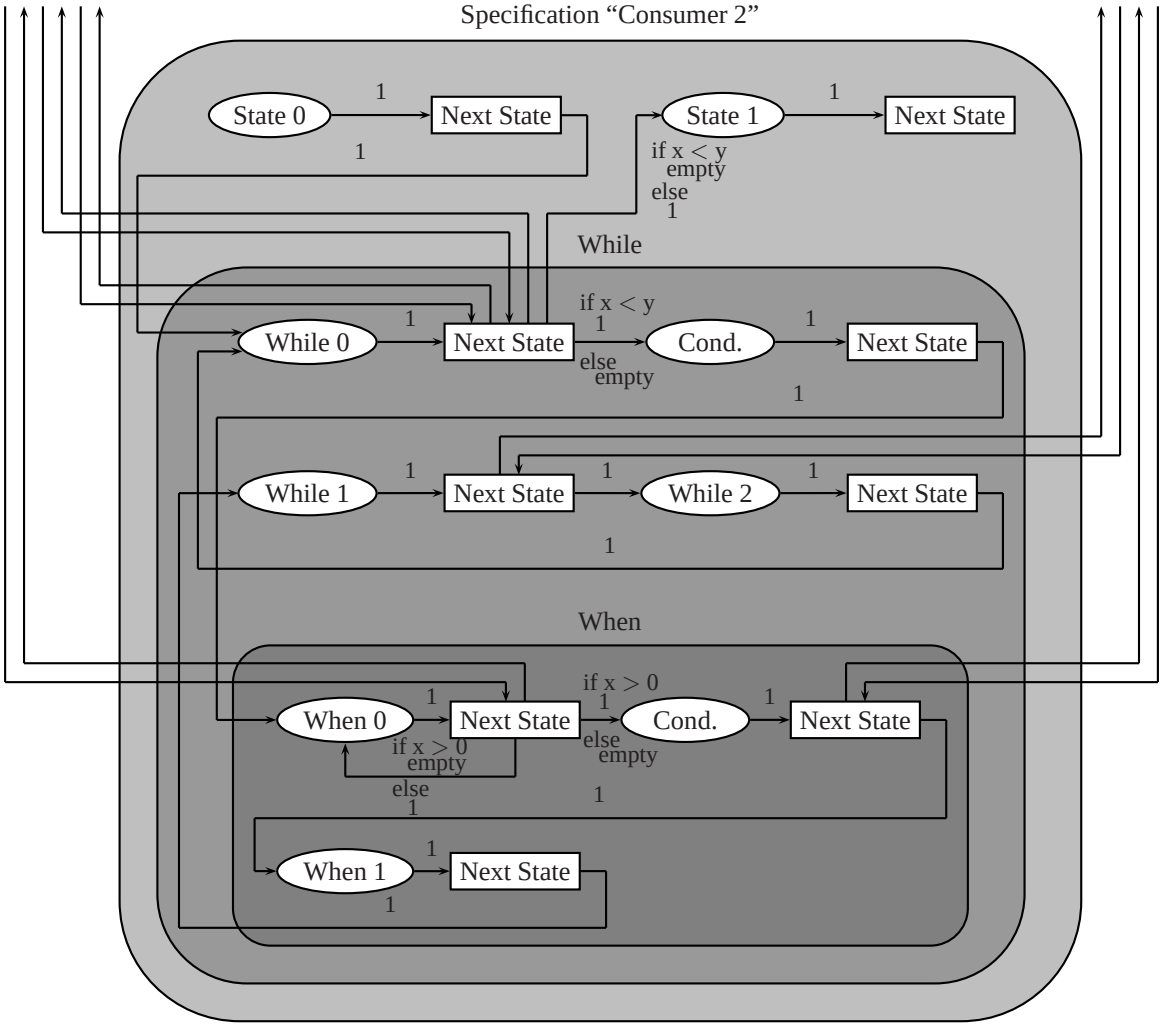

Figure 5.4: Colored Petri net of the “Consumer 2” specification.

As with the previous figure, certain arc expressions have been left off to avoid cluttering the image. In the figure, there is a while box inside of the specification box and a when box in the while box. This corresponds to the structure of the specification.

5.1.4 Results

There are a wide variety of automatic programming systems, but one goal they share in common is to reduce the amount of code that a programmer must write. Figure 5.5 compares the solutions of the

producer-consumer problem for C and CAPS:

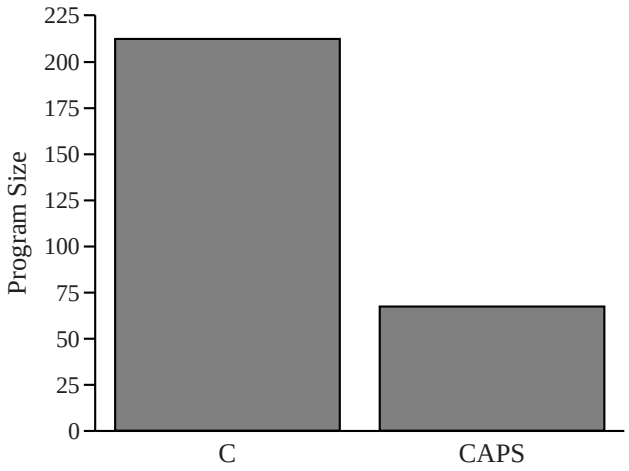

Figure 5.5: Comparison of the program sizes for the solutions to the producer-consumer problem.

The CAPS solution is approximately one-third the size of the C solution. Additionally, the CAPS solution can be automatically converted into a colored Petri net; thus allowing formal verification of the specifications.

5.2 Domination Number

Calculation of the *domination number* of a graph G is an NP-Complete problem; however, there are several classes of graphs for which the domination number can be calculated in polynomial time. This chapter shall focus on an algorithm to compute the domination number of a tree T.

Background information on domination, and the calculation of the domination number, is presented in §5.2.1. An algorithm for the domination number of a tree T is given in §5.2.2. §5.2.3 contains solutions for the algorithm presented in §5.2.2 for C and CAPS. The results of the study are given in §5.2.4.

5.2.1 Background

The *domination number* of a graph G shall be explained using an example. Take the graph shown in Figure 5.6, which consists of a set of vertices $V = (A, B, C, D, E, F, G)$ and a set of edges E connecting these vertices.

Find a *dominating set* of the graph G. A set S is a dominating set if all of the vertices that are

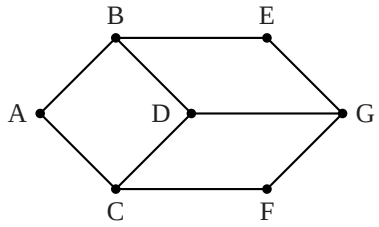

Figure 5.6: A graph G consisting of seven vertices.

not in S are neighbors of at least one vertex in S. Figure 5.7 shows the graph G with a dominating set $S = (A, D, E, F)$.

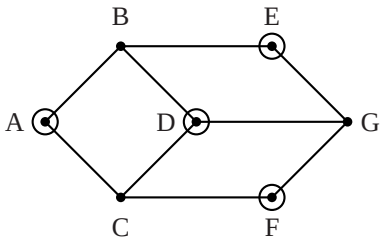

Figure 5.7: A dominating set of the graph G. In this example, the dominating set consists of 4 vertices.

In Figure 5.7, each vertex not in the dominating set S is adjacent to at least one vertex in S. For example, vertex B is adjacent to three dominating vertices: A, D, and E. If there was a vertex that was not in S and was not adjacent to any vertices in S, then S would not be a dominating set.

The domination number of a graph G, denoted $\gamma(G)$, equals the minimum cardinality of a dominating set $S \subseteq V(G)$. Figure 5.8 shows graph G with a minimum dominating set $S = (A, G)$.

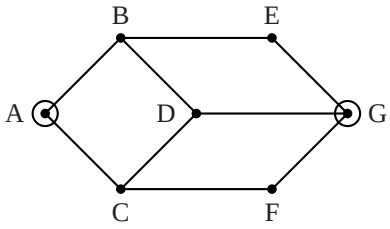

Figure 5.8: A minimum size dominating set of the graph G.

In the example graph GGG, a smallest dominating set has 2 vertices; therefore, $\gamma(G) = 2$. For a general graph G, determination of the domination number is an NP-Complete problem. For more information

on domination in graphs, the reader is encouraged to consult [37].

5.2.2 Algorithm

A linear algorithm for computing the domination number, $\gamma(T)$, of a tree T was presented in [18]. The algorithm from Cockayne, Goodman, and Hedetniemi is shown in Algorithm 5.2:

Algorithm 5.2 Linear algorithm by Cockayne, Goodman, and Hedetniemi for determining $\gamma(T)$ of a tree T.

```
DominatingSet := ∅
∀v ∈ T : v is BOUND
while |T| > 1 do
  v := LeafOf(T)
  u := NeighborOf(v)
  if v is BOUND then
    vertex u becomes REQUIRED
  else if v is REQUIRED then
    DominatingSet := DominatingSet ∪ v
    if u is BOUND then
      vertex u becomes FREE
    end if
  end if
  T := T − v
end while
if |T| == 1 then
  if v is !FREE then
    DominatingSet := DominatingSet ∪ v
  end if
  T := T − v
end if
```

The algorithm begins with an empty dominating set and a tree T. Every vertex of the tree has one of 3 labels: *bound*, *required*, or *free*. At each stage of the algorithm, a leaf is removed from the tree. If the leaf is labeled as *free*, the leaf can be removed and does not need to be added to the dominating set. If the leaf is labeled *required*, then another vertex in the tree depends upon the leaf for domination, therefore, the leaf must be added to the dominating set. If the leaf is labeled *bound* and has a neighbor, then the neighbor's label is changed to *required*. If the leaf that was labeled as *bound* does not have a neighbor, the leaf is added to the dominating set. At the end of the algorithm, there is one remaining vertex. If the vertex is not labeled as *free*, the vertex is added to the dominating set.

At the completion of the algorithm, the dominating set contains the minimum cardinality of vertices that dominate the tree T. At each stage of the algorithm, the amount of work done is O(1), and there are O(n) steps, so the algorithm is linear.

5.2.3 Solutions

In this section, solutions are given for the domination algorithm presented in §5.2.2. The programs are written in C and CAPSL. Interesting portions of the code are shown and discussed while the full programs can be found in Appendix B.2. Interesting portions of the colored Petri net conversion from the CAPSL solution is also shown.

5.2.3.1 C Solution

The C solution contains code for both the domination algorithm described in §5.2.2 and the creation and management of a tree data structure, which is handled by a combination of structs, enums, and dynamic arrays. Dynamic arrays are not necessary if the size of the tree is known in advance; however, this was done in order to give the C solution the same functionality as the CAPSL solution. Unlike the producer-consumer solution, the code executes in a single thread.

The C language, unlike most modern languages, does not provide direct support for dynamic arrays, but through the use of pointers and memory allocation, the behavior of dynamic arrays can be mimicked. For example, below is the struct representing a tree.

```
struct Tree
{
  struct Node **nNodes;
  int iNodes;
  int iNodeCapacity;
};
```

The struct for the tree contains a pointer to a struct for a node that points to an array of Node struct pointers. The other variables in the struct tell how many elements are in the array and the capacity of the array. The struct for the nodes is shown below:

```
struct Node
{
  enum NodeType ntType;
  struct Node **nNeighbors;
  int iNeighbors;
  int iNeighborCapacity;
};
```

Each node has a type that is an enumeration. The value of the enumeration is either free, bound, or required. A node also has a pointer pointing to an array of Node struct pointers. This array holds references to the neighbors of the node.

The solution contains functions for adding, removing, finding, and retrieving neighbors of a partic-

ular node or nodes of a tree. In addition, there is a function to return a leaf node of a tree. The domination_algorithm function of the solution closely resembles 5.2 with the only difference being that the C code needs to deal with memory management - freeing used memory if a node is deleted from the tree sent in to the algorithm. Below is a portion of the C code:

```
// Is there a remaining node?
if((tCurrent->nNodes != NULL) && (tCurrent->iNodes == 1) &&
   (tCurrent->nNodes[0] != NULL))
{
  iDelete = 1;
  // See what type the node is
  if(tCurrent->nNodes[0]->ntType != FREE)
  {
    // Node is not free. Add to the dominating set
    add_node(tDominationSet, tCurrent->nNodes[0]);
    iDelete = 0;
  }

  // Remove the remaining node
  remove_node(tCurrent, 0, iDelete);
}
```

In the code, there may be a node of the tree that remains to be processed. If the remaining node is not labeled free, in is included in the domination set and then removed from the tree. The full program listing can be found in Appendix B.2.1 and consists of 485 lines of code.

5.2.3.2 CAPS Solution

The C solution uses arrays and memory allocation to mimic the behavior of dynamic arrays. This results in the code becoming more complex and longer. The CAPS solution uses collections to represent a tree and the nodes of the tree. The types of collections in CAPS include sequences and sets and are able to grow dynamically. More information concerning collections may be found in §3.3.4.

The CAPS code has been designed to follow the structure of Algorithm 5.2. The following code appears after the `end while`:

```
if |sT| == 1
  sV = sT[0]
  % should the node be added?
  if sV[1] != FREE
    add sDomination sV
  end if
  % the node has been processed
  removeat sT 0
end if
```

In the code above, the algorithm has processed all but one node of the tree. If the remaining node is

labeled bound or required, that node must be included in the dominating set; otherwise, a minimum dominating set of the tree has already been found. In the CAPS code, the following adds the node `sV`, represented by a sequence, to the set `sDomination`:

```
add sDomination sV
```

As discussed earlier, the equivalent C code uses two structs to represent nodes and trees. The tree struct contains an array of nodes. When a node is added to a tree, there is a possibility that the capacity of the array will be exceeded. In that case, the capacity of the array must be increased. A small segment of the code responsible for that operation is shown below:

```
tCurrent->iNodeCapacity = 2;
tCurrent->nNodes =
   (struct Node **)malloc(sizeof(struct Node *) *
   tCurrent->iNodeCapacity);
tCurrent->iNodes = 0;
```

In the CAPS code below, the element at index zero of the sequence `sT` is removed:

```
removeat sT 0
```

A C function mimics this behavior. A small portion of the code is shown below in which the elements of the array of nodes are shifted:

```
int i;
for(i = iIndex; i < tCurrent->iNodes - 1; i++)
{
  tCurrent->nNodes[i] = tCurrent->nNodes[i + 1];
}
```

The complete CAPS solution can be found in Appendix B.2.2. The CAPS code is 75 lines – 15% the size of the C code; though, most of this difference is a result of the use of collections.

5.2.3.3 CAPS Colored Petri Net Solution

Earlier, conversion of CAPSL code for the producer-consumer problem into a colored Petri net was examined. In the CAPS solution for the domination number, the following code was shown (the complete solution can be found in Appendix B.2.2):

```
if |sT| == 1
  sV = sT[0]
  % should the node be added?
  if sV[1] != FREE
    add sDomination sV
  end if
  % the node has been processed
  removeat sT 0
end if
```

In the code, all but one node of the tree has been processed, and the if statements are used to determine if the node needs to be added to the domination set. In Section 3.5, the conversion of CAPSL statements into a colored Petri net was discussed. Figure 5.9 shows the colored Petri net representation of the CAPSL code:

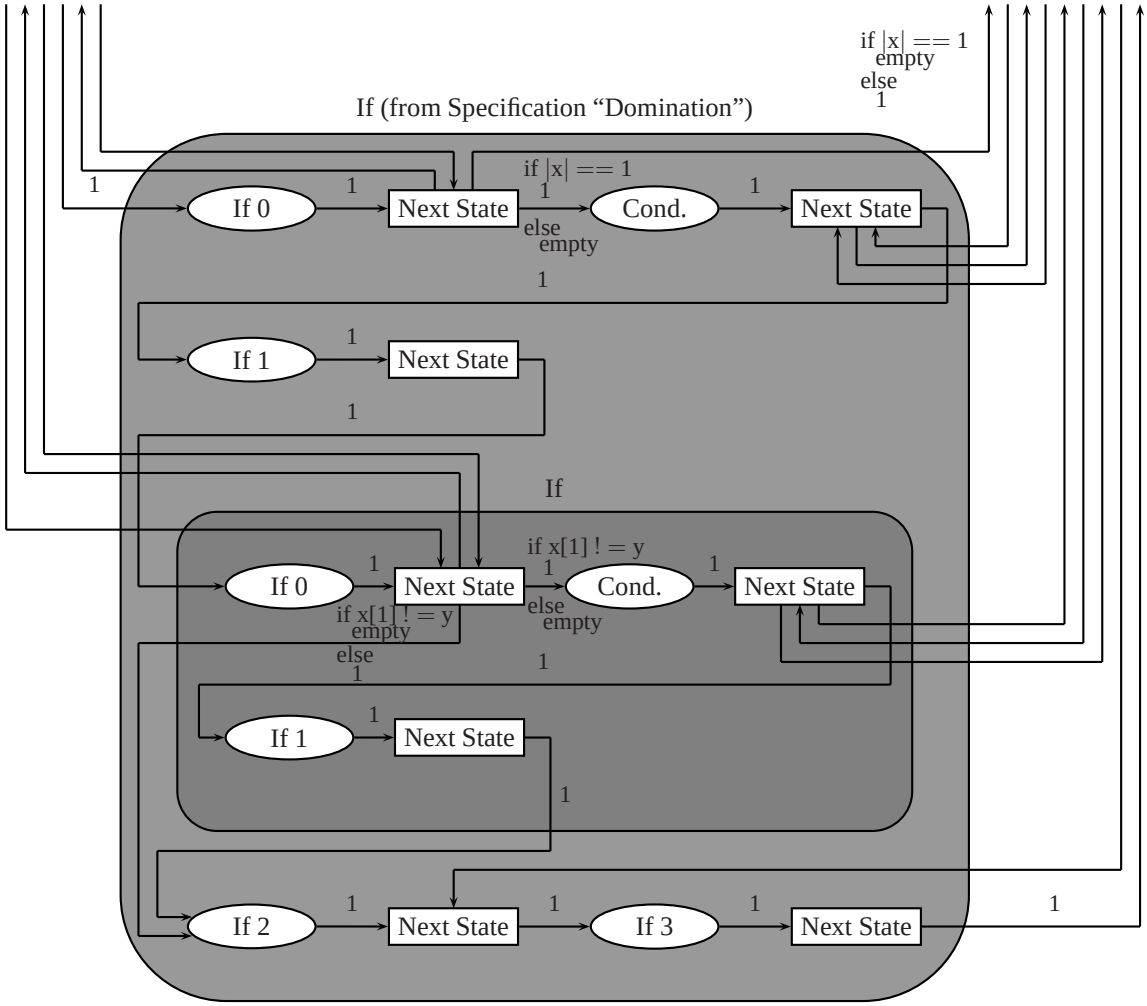

Figure 5.9: Portion of the colored Petri net from the "Domination" specification.

In the figure, the transition between places *If 0* and *Cond.* in the outer if block has five arcs. The two unlabeled arcs are used to retrieve, and place back, the sequence `sT`. In the arcs with the if expressions, the sequence is represented by the variable *x*. More information concerning if blocks can be found in §3.5.5. The transition after the place *Cond.* in the outer if block has three unlabeled arcs. One of these arcs is used to retrieve the element at index 0 of sequence `sT`. Another arc removes the elements of sequence `sV`, and the

third arc sets the sequence `sV` to the element that was retrieved from `sT`.

In the inner if block, the transition between places *If 0* and *Cond.* has three unlabeled arcs. Two of the arcs are used to retrieve, and place back, the value of the variable `FREE`. The other retrieves the element at index 1 of the sequence `sV`. The transition in the inner if block after the place *Cond.* also has three unlabeled arcs. Two of the arcs are used to retrieve, and place back, all of the elements of sequence `sV`. The other arc adds `sV` to the set `sDomination`.

The transition between places *If 2* and *If 3* in the outer if block has one unlabeled arc. That arc removes the element at index zero of the sequence `sT`.

5.2.4 Results

§5.2.3 discussed solutions to the domination number algorithm for C and CAPS. Figure 5.10 shows a comparison of the program sizes for the solutions:

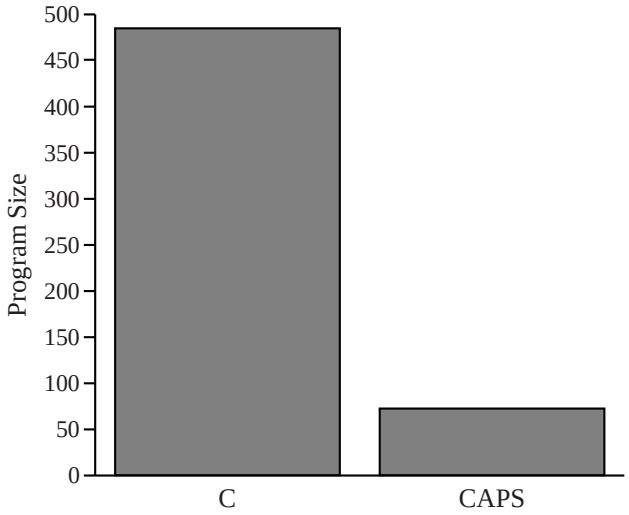

Figure 5.10: Comparison of the program sizes for the solutions to the domination algorithm.

The primary difference between the solutions are the collections (see §3.3.4) that are built into CAPS. Had C++ and the standard template library been used, or another language such as Java that contains collections, the difference in the solution sizes would be smaller.

5.3 Comparison to Other Systems

In this chapter, the specification language of CAPS has been compared to the programming language C for the producer-consumer and domination number problems. The conclusions of these comparisons are given in §5.3.1. Additionally, the formal verification properties of CAPS are compared to those of SpecWare and SMV in Chapter 4 and reviewed in §5.3.2 and §5.3.3 respectively.

5.3.1 C

C is a popular high-level programming language that has been used for programs ranging from operating systems to video games. CAPSL, the specification language of CAPS, is a very high-level language that focuses on concurrency. As a very high-level language, users of CAPSL do not need to worry about issues such as memory management or casting of types. Additionally, the case studies showed that CAPSL code may be more succinct than the equivalent C code; however, with regard to libraries and execution speed, C has a clear advantage over CAPSL. The specification language of CAPS is not optimized for performance and this is an area of future work (see §6.2.1).

The primary advantage of using CAPS over C, or any similar language such as Java, is the formal verification built into CAPS (see Chapter 4).

5.3.2 SpecWare

SpecWare is among the state-of-the-art automatic programming systems that is capable of proving and transforming specifications into Lisp and Java code [53]. The specification language of SpecWare is known as MetaSlang and is derived from higher-order logic. A MetaSlang specification can be proven through the use of a theorem prover (see §4.2.1).

However, SpecWare is not currently capable of creating and verifying concurrent specifications. Additionally, the syntax and semantics of MetaSlang are sufficiently different from standard programming languages that developers need to spend a significant amount of time learning the language. CAPSL, the specification language of CAPS, is based on C, Java, and Basic, which should reduce the learning curve required to use the system.

5.3.3 SMV

SMV is a model checker capable of formally verifying concurrent specifications [54]; however, unlike SpecWare, SMV is not an automatic programming system. SMV specifications are typically of hardware designs. The verification of a SMV specification is done by checking all possible execution paths of a program. In this respect, the formal verification of a CAPS specification more closely resembles the verification of SMV rather than SpecWare. The input language of SMV is derived from CTL – a form of temporal logic.

The issues of SMV are two-fold. The first is that developers, in general, are not familiar with temporal logic. As mentioned previously, the syntax and semantics of CAPSL are based off of popular programming languages in order to try to minimize the learning curve of new users. The second issue is that a specification written for SMV must be translated to another form after formal verification has been performed. This could be either to a programming language or a hardware design. CAPS was designed as a complete system so that the specifications can be both verified and executed.

5.4 Summary

The case studies of this chapter have shown that the specification language of CAPS is capable of modeling systems written in other languages such as C. Additionally, for the problems presented in this chapter, the specification language was able to model the system with significantly less code. Another benefit of using CAPS is that the code can automatically be converted into a colored Petri net. This allows for formal verification of the specifications. The conclusions of the dissertation are given in Chapter 6.

Chapter 6

Conclusions

Concurrent computing has become an important issue with the introduction of multiple core processors; however, the creation of a concurrent application is a difficult task. In a concurrent application, multiple threads are interacting in ways that may be unforeseen by the programmer. CAPS (Concurrent Automatic Programming System) was developed as a system capable of describing both concurrency and formal verification, with the formal verification based on colored Petri nets. The specification language was developed such that a mapping could be created for converting each structure and statement into a CP-net. Once the specifications are represented by a colored Petri net, a simulation can be run or formal verification of the specifications can be performed.

CAPS handles many of the details of concurrency for the user and allows formal verification of the specifications in which all possible states are examined. This allows the programmer to be able to determine if the specifications ever reach a set of desired, or undesired, states. Summaries of the contributions made by this research and the conclusions of the research are presented in Section 6.1. Areas of future work are discussed in Section 6.2.

6.1 Summary of Contributions and Conclusions

The contributions of this research are discussed in the list below:

- The primary contribution of this research is a concurrent automatic programming system. The state-of-the-art automatic programming systems discussed in Chapter 2 are all sequential. Specifications are becoming more complex, and automatic programming is one method of handling the additional com-

plexity; however, with the recent trend of adding multiple cores to processors, automatic programming systems must be able to create concurrent specifications. This research shows one method of adding concurrency to an automatic programming system.

- The specification language CAPSL is a very high-level programming language designed for concurrency that uses a different approach to concurrent computing than other specification and programming languages. Many programmers understand how to create sequential applications but do not understand the issues of concurrency. The specification language was designed to handle those details so that a user can quickly create a concurrent application without needing to understand mutexes, race conditions, and other issues involved with concurrency.

- An extensive mapping has been created from the specification language to colored Petri nets. This extends work that has been done in other areas of software engineering [9] [63]. Those methods converted a subset of a language into Petri nets. The specification language and the colored Petri nets of this research were developed simultaneously to allow each structure and statement of the specification language to be converted automatically into a colored Petri net. The colored Petri nets retain the same semantics as the specifications that were transformed; therefore, if the colored Petri nets are correct, the specifications are correct. This is a critical point for the formal verification of the specifications.

- The colored Petri nets of this research were developed in order represent the structures and statements of the specification language and are a new type of hierarchical CP-net. Several modifications have been made to the formal verification of the colored Petri nets. Some of these modifications are due to the differences between the formal verification of a colored Petri net and a specification. In a specification, the state changes when the value of a variable changes. In a CP-net, the state changes when the colors of a place change. The formal verification has therefore been modified to examine subsets of places and colors. These subsets are based on the specification language.

The formal verification of concurrent specifications was one of the primary goals of this research. There already exist several specification and programming languages that are designed for concurrent computing, but there are few that are able to handle both concurrent computing and formal verification. In order to aid the user in formal verification, CAPSL includes several formal verification statements. This allows the user to quickly create markings and bindings to verify properties of the specifications. The use of colored Petri nets in CAPS is primarily for the formal verification of specifications; however, an additional benefit is

that colored Petri nets have a graphical representation. A simulator has been developed for the colored Petri net engine of CAPS. The user is able to control the flow of execution as well as change the state of the colored Petri net. Several automatic programming systems have been designed to create simulation applications, but those systems are not able to simulate the specifications themselves.

A natural language processing (NLP) system is used as a front-end for the specification language. Other automatic programming systems that have used an NLP interface include [32] and [33]; however, few recent automatic programming systems have incorporated natural language. One of the goals of an automatic programming system is to raise the level of the input language such that the system can be used by individuals other than programmers. A natural language interface would therefore appear to be one of the best methods to achieve this goal.

A goal of this research was to create an automatic programming system to aid users in the creation of concurrent specifications. As was seen in the case study of the producer-consumer problem, CAPS is able to produce equivalent code that is both smaller and more readable. This is achieved by CAPS handling many of the issues of concurrency for the user.

Current automatic programming systems typically can scale to no more than a few thousand lines of specification. This is the case with CAPS as well. The specification language needs to be optimized before larger specifications are used. The formal verification of a specification is more dependent upon the state space of the specification than the length of the specification. The system has been verified for state spaces that contain thousands of occurrence graph nodes, and there are methods that can be implemented to reduce the size of the state space for certain classes of specifications.

The natural processing system does relax some constraints on syntax for the user. This allow the user to focus more on the design of specifications; however, the user must also be trained for the natural language processing system.

6.2 Future Work

CAPS can currently be used for the creation, execution, and formal verification of concurrent specifications, while there is further development to be done on the various systems of CAPS. In §6.2.1, future work on the specification language is discussed, §6.2.2 describes further work to be done on the colored Petri net engine, and §6.2.3 mentions possible enhancements to the natural language processing engine.

6.2.1 Specification Language

CAPSL, the specification language of CAPS, is a very high-level language designed for the creation of concurrent specifications (see Chapter 3). CAPSL includes features such as concurrency, memory management, collection data types, and automatic conversion of data types. An interpreter is provided that allows for the direct execution of CAPSL code. This removes the need to convert CAPSL code into an executable form – such as Lisp or Java code. The interpreter is not optimized and thus will not perform at the same level as a conventional programming language; therefore, one possible area of future work is the execution of the specifications.

Additionally, several standard features are not present in the language. CAPSL has only partial support for `return` statements, and there is currently no support for libraries; this may be supplemented through the use of a knowledge base as discussed below.

6.2.1.1 Knowledge Base

A knowledge base is essential for the use of CAPS in specific fields. Figure 6.1 is adapted from Rich and Waters' [70] and shows domain knowledge relative to programming knowledge for different users of an automatic programming system:

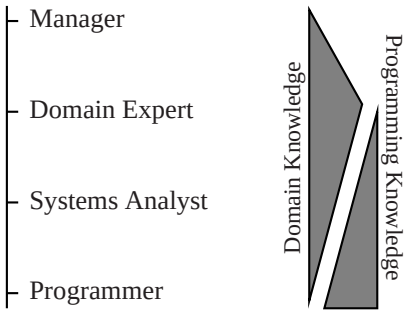

Figure 6.1: Comparison of domain knowledge to programming knowledge.

The knowledge base would include either CAPSL code or elements that could automatically be converted into CAPSL code. Information in the knowledge base may include domain information, such as graph isomorphism algorithms or networking components.

6.2.2 Colored Petri Nets

The primary goal of CAPS is to aid the user in the creation and verification of concurrent specifications, two areas in which Petri nets perform well. Colored Petri nets are high-level Petri nets that permit a more efficient representation of complex expressions than with standard Petri nets. Figure 6.2 from Chapter 5 shows the colored Petri net representation of a specification.

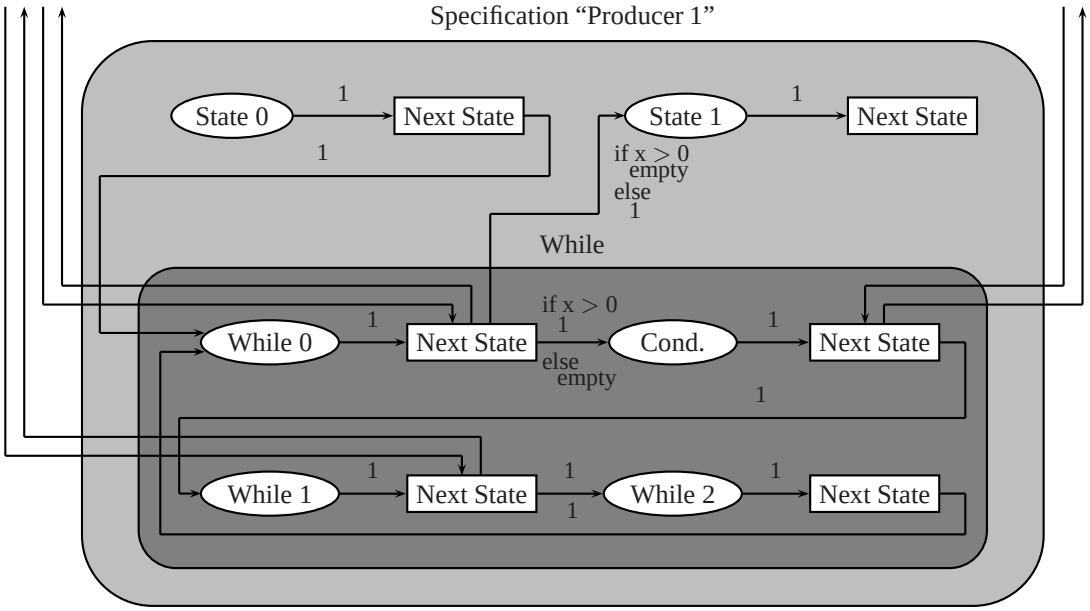

Figure 6.2: Colored Petri net of the “Producer 1” specification.

The colored Petri net engine of CAPS has been designed both as one component of CAPS and as an independent component that can be used in other systems [45]. The interface of the colored Petri net engine allows for the visual inspection of, and control over, an executing specification; however, the colored Petri nets created from the conversion of CAPSL code can be difficult for a human to parse. Support should be added such that the places and transitions associated with a particular CAPSL statement can be highlighted.

6.2.2.1 Formal Verification

Formal verification of a colored Petri net is a difficult process and requires significant amounts of time and memory. There are methods that can be used to reduce the time and memory required for certain classes of colored Petri nets. One method is to use OE-graphs, occurrence graphs with equivalence classes. OE-graphs should be incorporated into the formal verification of CAPS.

CAPS currently supports the reachability and boundedness formal verification properties (see Chapter 4). Support should be extended for home, liveness, and fairness. This requires a modification in the construction of the SCC-graph. Additionally, CAPS should be able to detect deadlocks that may arise from the use of concurrency. One example would be a `when` statement whose condition can never become true.

6.2.3 Natural Language Processing

A goal of automatic programming systems is to allow users that are not software developers to create specifications. In order to do this, the system must be able to hide much of the syntax and semantics of the specification language from the user. One example is Amphion, in which the user interacts with the system through a graphical interface[51]. A more common approach is to use a natural language processing system as a front end, first done by QA3 [20]. However, natural language processing systems have not been used on recent automatic programming systems.

CAPS does include a basic natural language processing system in order to relax the constraints of syntax for users of the system (see Chapter 3), but this interface should be extended to natural language processing in general. The natural language processing engine should be expanded to handle a greater subset of English and CAPSL commands.

6.2.3.1 Automatic Speech Recognition

The user currently interacts with the natural language processing engine of CAPS via the keyboard; however, the dictionary of CAPS incorporates CMUDict. CMUDict contains over 125,000 words and the phonemes associated with those words. Those phonemes can then be used in an automatic speech recognition system that would allow users of CAPS to interact with the system via speech.

Appendix A

Colored Petri Nets

Colored Petri nets (CP-nets) were discussed in Chapter 2. In this appendix, we examine a few elements of the CP-net in more detail. In Section A.1, we look at the operators and functions available for use on the different colors.

A.1 Color Operators and Functions

This section details the operators and functions specific to the colors of the CP-net kernel. Every color in the kernel has at least seven operators. The operators common to all of the colors are:

- =
- ==
- !=
- <
- >
- <=
- >=

As in C and C++, the first operator is assignment. The remaining six operators for the unit and string colors are based off of the labels of those colors. For the boolean, integer, and real colors, the last six

operators are based off of the numerical value of the colors in which False is assumed to be 0 and True to be everything else for the boolean color.

The remaining operators and functions for the unit colors are in §A.1.1, boolean colors are examined in §A.1.2, integers in §A.1.3, reals in §A.1.4, and the string colors in §A.1.5.

A.1.1 Unit

A unit is the simplest color type. The default name, and value, of a unit color is (); however, this name/value can be changed upon initialization of the CP-net.

A.1.2 Boolean

There are three other operators for the boolean color. The boolean operators are equivalent to their C/C++ counterparts and are as follows:

- $\sim$
- &&
- ||

The $\sim$ operator is unary, converting False to True and True to False. The && operator is binary and is a logical *and*. The || operator is binary and is a logical *or*.

A.1.3 Integer

The integer color has an additional six operators and 3 functions. The operators and functions are shown below:

- $\sim$
- $+$
- $-$
- $*$
- $/$

- %
- abs
- min
- max

The $\sim$ operator is unary and is equivalent to multiplying the argument by -1. The arithmetic operators are binary and self explanatory. The % operator, as in C/C++, is modulo.

The *abs* function takes one argument and returns the absolute value of that argument while the *min* and *max* functions take two arguments and return the respective smaller or larger value.

A.1.4 Real

The real color has five additional operators and 15 additional functions. These additional operators and functions are displayed in the list below:

- $\sim$
- $+$
- $-$
- $*$
- $/$
- squareRoot
- abs
- min
- max
- floorInt
- ceilingInt
- naturalLog

- logarithm
- exponential
- sine
- arcSine
- cosine
- arcCosine
- tangent
- arcTangent

The $\sim$ operator is a unary operator that effectively multiplies its argument by -1. The last four operators are binary and are used for arithmetic.

There are 15 functions for the real colors. Of these functions, all except the *min* and *max* functions take one argument. Those two functions take two arguments. The *floorInt* and *ceilingInt* functions take a real number and return, respectively, the next lowest or highest integer.

A.1.5 String

The string color has one additional operator and two additional functions. The operator and functions are:

- $\wedge$
- substring
- size

The $\wedge$ operator is binary and is used for the concatenation of two strings.

The *substring* function takes a string and two integers. The first integer denotes the position and the second integer the length of the substring to return from the string argument. The *size* function takes a string and returns an integer number representing the number of characters in the string.

Appendix B

Code of Case Studies: C and CAPS

This appendix contains C and CAPS source code for the producer-consumer and domination number problems.

B.1 Producer-Consumer

This section contains the C and CAPS source code for the solutions to the producer-consumer problem.

B.1.1 C

A full listing of the C code for the problem is listed below:

```
001 #include <stdio.h>
002 #include <pthread.h>
003
004 // The initial number of items for producers 1 and 2
005 #define PRODUCER1_INITIAL 5000
006 #define PRODUCER2_INITIAL 5000
007
008 // The mutexes for the buffers so a producer and consumer
009 //    will not access them at the same time
010 pthread_mutex_t pmtMutexB1       = PTHREAD_MUTEX_INITIALIZER;
011 pthread_mutex_t pmtMutexB2       = PTHREAD_MUTEX_INITIALIZER;
```

```
012 pthread_mutex_t pmtMutexCondB2   = PTHREAD_MUTEX_INITIALIZER;
013 pthread_cond_t  pctConditionB2   = PTHREAD_COND_INITIALIZER;
014 pthread_mutex_t pmtMutexCondB1   = PTHREAD_MUTEX_INITIALIZER;
015 pthread_cond_t  pctConditionB1   = PTHREAD_COND_INITIALIZER;
016 pthread_mutex_t pmtMutexCondC1B2 = PTHREAD_MUTEX_INITIALIZER;
017 pthread_cond_t  pctConditionC1B2 = PTHREAD_COND_INITIALIZER;
018
019 // The producer, consumer, and buffer counters
020 int iProducer1 = PRODUCER1_INITIAL;
021 int iProducer2 = PRODUCER2_INITIAL;
022 int iConsumer1 = 0;
023 int iConsumer2 = 0;
024 int iBuffer1   = 0;
025 int iBuffer2   = 0;
026
027 // -- Function that moves items from producer 1 to buffer 1 --
028 void *producer1()
029 {
030   // Add all of the items from the producer to the buffer
031   while(iProducer1-- > 0)
032   {
033     // Need a lock for the buffer
034     pthread_mutex_lock(&pmtMutexB1);
035     // Do not worry about overfilling the buffer
036     iBuffer1++;
037     pthread_mutex_unlock(&pmtMutexB1);
038
039     // Indicate to consumer 1 that an item added to buffer
040     pthread_mutex_lock(&pmtMutexCondB1);
041     pthread_cond_signal(&pctConditionB1);
042     pthread_mutex_unlock(&pmtMutexCondB1);
043   }
044 }
```

```
045
046 // -- Function that moves items from producer 2 to buffer 2 --
047 void *producer2()
048 {
049   // Add all of the items from the producer to the buffer
050   while(iProducer2-- > 0)
051   {
052     // Need a lock for the buffer
053     pthread_mutex_lock(&pmtMutexB2);
054     // Do not worry about overfilling the buffer
055     iBuffer2++;
056     pthread_mutex_unlock(&pmtMutexB2);
057
058     // Indicate to consumer 2 that an item added to buffer
059     pthread_mutex_lock(&pmtMutexCondB2);
060     pthread_cond_signal(&pctConditionB2);
061     pthread_mutex_unlock(&pmtMutexCondB2);
062   }
063 }
064
065 // -- Function locks buffer 1 when it is not empty --
066 void consumer1_lock_b1()
067 {
068   // Continue trying to lock the buffer when it is not empty
069   while(1)
070   {
071     // We need to access the buffer 1 counter
072     pthread_mutex_lock(&pmtMutexB1);
073     // Is there anything in the buffer
074     if(iBuffer1 < 1)
075     {
076       // The buffer is empty, so release the mutex
077       pthread_mutex_unlock(&pmtMutexB1);
```

```
078
079       // We need to wait on a signal indicating when the
080       //    buffer is not empty
081       pthread_mutex_lock(&pmtMutexCondB1);
082       pthread_cond_wait(&pctConditionB1, &pmtMutexCondB1);
083       pthread_mutex_unlock(&pmtMutexCondB1);
084     }
085     else
086     {
087       // The buffer is not empty and we have it locked
088       break;
089     }
090   }
091 }
092
093 // -- Function locks buffer 2 when it is empty --
094 void consumer1_lock_b2()
095 {
096   // Continue trying to lock the buffer when it is empty
097   while(1)
098   {
099     // We need to access the buffer 2 counter
100     pthread_mutex_lock(&pmtMutexB2);
101     // Is there anything in the buffer
102     if(iBuffer2 > 0)
103     {
104       // The buffer is not empty, so release the mutex
105       pthread_mutex_unlock(&pmtMutexB2);
106
107       // We need to wait on a signal indicating when the
108       //    buffer is empty. Once we have the signal, need
109       //    to lock the buffer again and see if it is still
110       //    empty
```

```
111       pthread_mutex_lock(&pmtMutexCondC1B2);
112       pthread_cond_wait(&pctConditionC1B2, &pmtMutexCondC1B2);
113       pthread_mutex_unlock(&pmtMutexCondC1B2);
114     }
115     else
116     {
117       // The buffer is empty and we have it locked
118       break;
119     }
120   }
121 }
122
123 // -- Function that moves items from buffer 1 to consumer 1 --
124 void *consumer1()
125 {
126   // Move all of the items from the buffer to the consumer
127   while(iConsumer1++ < PRODUCER1_INITIAL)
128   {
129     // Lock the first buffer when it is not empty
130     consumer1_lock_b1();
131
132     // Lock the second buffer when it is empty
133     consumer1_lock_b2();
134
135     // Remove the item from the buffer
136     iBuffer1--;
137     pthread_mutex_unlock(&pmtMutexB2);
138     pthread_mutex_unlock(&pmtMutexB1);
139   }
140 }
141
142 // -- Function locks buffer 2 when it is not empty --
143 void consumer2_lock_b2()
```

```
144 {
145   // Continue trying to lock the buffer when it is not empty
146   while(1)
147   {
148     // We need to access the buffer 2 counter
149     pthread_mutex_lock(&pmtMutexB2);
150     // Is there anything in the buffer
151     if(iBuffer2 < 1)
152     {
153       // The buffer is empty, so release the mutex
154       pthread_mutex_unlock(&pmtMutexB2);
155
156       // We need to wait on a signal indicating when the
157       //    buffer is not empty
158       pthread_mutex_lock(&pmtMutexCondB2);
159       pthread_cond_wait(&pctConditionB2, &pmtMutexCondB2);
160       pthread_mutex_unlock(&pmtMutexCondB2);
161     }
162     else
163     {
164       // The buffer is not empty and we have it locked
165       break;
166     }
167   }
168 }
169
170 // -- Function that moves items from buffer 2 to consumer 2 --
171 void *consumer2()
172 {
173   // Move all of the items from the buffer to the consumer
174   while(iConsumer2++ < PRODUCER2_INITIAL)
175   {
176     // Lock the second buffer when it is not empty
```

```
177     consumer2_lock_b2();
178
179     // Remove the item from the buffer
180     iBuffer2--;
181
182     // If the buffer is empty, send a signal indicating so.
183     //    The consumer 1 may be waiting for the signal
184     if(iBuffer2 == 0)
185     {
186       pthread_mutex_lock(&pmtMutexCondC1B2);
187       pthread_cond_signal(&pctConditionC1B2);
188       pthread_mutex_unlock(&pmtMutexCondC1B2);
189     }
190     pthread_mutex_unlock(&pmtMutexB2);
191   }
192 }
193
194 int main()
195 {
196    int i;
197    pthread_t ptThreadID[4];
198
199    // Create the producer and consumer threads
200    pthread_create(&ptThreadID[0], NULL, &consumer1, NULL);
201    pthread_create(&ptThreadID[1], NULL, &consumer2, NULL);
202    pthread_create(&ptThreadID[2], NULL, &producer1, NULL);
203    pthread_create(&ptThreadID[3], NULL, &producer2, NULL);
204
205    // Wait for all of the threads to finish
206    for(i = 0; i < 4; i++)
207    {
208      pthread_join(ptThreadID[i], NULL);
209    }
```

```
210
211    return 0;
212 }
```

B.1.2 CAPS

A full listing of the CAPS code for the problem is listed below:

```
001 % the initial number of items for producers 1 and 2
002 integer PRODUCER1_INITIAL = 5000
003 integer PRODUCER2_INITIAL = 5000
004
005 % the producer, consumer, and buffer counters
006 integer iProducer1 = PRODUCER1_INITIAL
007 integer iProducer2 = PRODUCER2_INITIAL
008 integer iConsumer1 = 0
009 integer iConsumer2 = 0
010 integer iBuffer1 = 0
011 integer iBuffer2 = 0
012
013 % -- Prime specification that initializes buffers --
014 start specification "Producer-Consumer"
015   % run the producer and consumer specifications
016   specification "Producer 1"
017   specification "Producer 2"
018   specification "Consumer 1"
019   specification "Consumer 2"
020 end specification
021
022 % -- Specification moves items from producer 1 to buffer 1 --
023 start specification "Producer 1"
024   % Add all of the items from the producer to the buffer
025   while iProducer1 > 0
026     % Do not worry about overfilling the buffer
027     iBuffer1 = iBuffer1 + 1
```

```
028     iProducer1 = iProducer1 - 1
029   end while
030 end specification
031
032 % -- Specification moves items from producer 2 to buffer 2 --
033 start specification "Producer 2"
034   % Add all of the items from the producer to the buffer
035   while iProducer2 > 0
036     % Do not worry about overfilling the buffer
037     iBuffer2 = iBuffer2 + 1
038     iProducer2 = iProducer2 - 1
039   end while
040 end specification
041
042 % -- Specification moves items from buffer 1 to consumer 1 --
043 start specification "Consumer 1"
044   % Move all of the items from the buffer to the consumer
045   while iConsumer1 < PRODUCER1_INITIAL
046     % Remove an item from the buffer when it is not empty and
047     %    the other buffer is empty
048     when iBuffer1 > 0 and iBuffer2 == 0
049       % Remove the item from the buffer
050       iBuffer1 = iBuffer1 - 1
051     end when
052     iConsumer1 = iConsumer1 + 1
053   end while
054 end specification
055
056 % -- Specification moves items from buffer 2 to consumer 2 --
057 start specification "Consumer 2"
058   % Move all of the items from the buffer to the consumer
059   while iConsumer2 < PRODUCER2_INITIAL
060     % Remove an item from the buffer when it is not empty and
```

```
061     %    the other buffer is empty
062     when iBuffer2 > 0
063       % Remove the item from the buffer
064       iBuffer2 = iBuffer2 - 1
065     end when
066     iConsumer2 = iConsumer2 + 1
067   end while
068 end specification
```

B.2 Domination

This section contains the C and CAPS source code for the solutions to the domination number problem.

B.2.1 C

A full listing of the C code for the problem is listed below:

```
001 #include <stdio.h>
002 #include <stdlib.h>
003
004 // A label of a node
005 enum NodeType
006 {
007   FREE,
008   BOUND,
009   REQUIRED
010 };
011
012 // A node of the tree
013 struct Node
014 {
015   enum NodeType ntType;
016   struct Node **nNeighbors;
017   int iNeighbors;
```

```
018   int iNeighborCapacity;
019 };
020
021 // A tree containing nodes
022 struct Tree
023 {
024   struct Node **nNodes;
025   int iNodes;
026   int iNodeCapacity;
027 };
028
029 // -- Function to add a neighbor to a node --
030 int add_neighbor(struct Node *nCurrent,
031    struct Node *nNeighbor)
032 {
033   // Make sure the arguments are valid
034   if((nCurrent == NULL) || (nNeighbor == NULL))
035   {
036     // A problem adding the neighbor
037     return 1;
038   }
039
040   // Does a list of neighbors need to be created?
041   if(nCurrent->nNeighbors == NULL)
042   {
043     // Create the list of neighbors
044     nCurrent->iNeighborCapacity = 2;
045     nCurrent->nNeighbors =
046        (struct Node **)malloc(sizeof(struct Node *) *
047        nCurrent->iNeighborCapacity);
048     nCurrent->iNeighbors = 0;
049
050     // NULL out the members of the array
```

```
051     int i;
052     for(i = 0; i < nCurrent->iNeighborCapacity; i++)
053     {
054       nCurrent->nNeighbors[i] = NULL;
055     }
056   }
057
058   // Make sure that the node is not already a neighbor
059   int i;
060   for(i = 0; i < nCurrent->iNeighbors; i++)
061   {
062     if(nCurrent->nNeighbors[i] == nNeighbor)
063     {
064       // Node is already a neighbor
065       return 1;
066     }
067   }
068
069   // Does the list of neighbors need to be expanded?
070   if(nCurrent->iNeighbors == nCurrent->iNeighborCapacity)
071   {
072     // Create a new array for the neighbors
073     nCurrent->iNeighborCapacity =
074        (nCurrent->iNeighborCapacity * 2) + 1;
075     struct Node **nNewArray =
076        (struct Node **)malloc(sizeof(struct Node *) *
077        nCurrent->iNeighborCapacity);
078
079     // Copy over the contents of the array
080     for(i = 0; i < nCurrent->iNeighbors; i++)
081     {
082       nNewArray[i] = nCurrent->nNeighbors[i];
083     }
```

```
084
085     // NULL out the other members of the array
086     for(i = nCurrent->iNeighbors;
087        i < nCurrent->iNeighborCapacity; i++)
088     {
089       nNewArray[i] = NULL;
090     }
091
092     // Set the reference to the new array and delete the old
093     free(nCurrent->nNeighbors);
094     nCurrent->nNeighbors = nNewArray;
095   }
096
097   // Add the neighbor
098   nCurrent->nNeighbors[nCurrent->iNeighbors] = nNeighbor;
099   nCurrent->iNeighbors++;
100
101   // No problems were encountered
102   return 0;
103 }
104
105 // -- Function to get a neighbor of a node --
106 struct Node* get_neighbor(struct Node *nCurrent, int iIndex)
107 {
108   // Make sure the arguments are valid
109   if((nCurrent == NULL) || (nCurrent->nNeighbors == NULL) ||
110      (iIndex < 0) || (iIndex >= nCurrent->iNeighbors))
111   {
112     // A problem getting the neighbor
113     return NULL;
114   }
115
116   // Return the neighbor
```

```
117   return nCurrent->nNeighbors[iIndex];
118 }
119
120 // -- Function to find the index of a neighbor --
121 int find_neighbor(struct Node *nCurrent,
122    struct Node *nNeighbor)
123 {
124   // Make sure the arguments are valid
125   if((nCurrent == NULL) || (nCurrent->nNeighbors == NULL) ||
126      (nNeighbor == NULL))
127   {
128     // A problem finding the neighbor
129     return -1;
130   }
131
132   // Search through the neighbors
133   int i;
134   for(i = 0; i < nCurrent->iNeighbors; i++)
135   {
136     // Check to see if this is the neighbor
137     if(nCurrent->nNeighbors[i] == nNeighbor)
138     {
139       return i;
140     }
141   }
142
143   // The neighbor was not found
144   return -1;
145 }
146
147 // -- Function to remove a neighbor of a node --
148 int remove_neighbor(struct Node *nCurrent, int iIndex)
149 {
```

```
150   // Make sure the arguments are valid
151   if((nCurrent == NULL) || (nCurrent->nNeighbors == NULL) ||
152      (iIndex < 0) || (iIndex >= nCurrent->iNeighbors))
153   {
154     // A problem removing the neighbor
155     return 1;
156   }
157
158   // Save a reference to the neighbor
159   struct Node *nNeighbor = nCurrent->nNeighbors[iIndex];
160
161   // Do the neighbors need to be shifted?
162   int i;
163   for(i = iIndex; i < nCurrent->iNeighbors - 1; i++)
164   {
165     nCurrent->nNeighbors[i] = nCurrent->nNeighbors[i + 1];
166   }
167
168   // Show that a neighbor was removed
169   nCurrent->iNeighbors--;
170   nCurrent->nNeighbors[nCurrent->iNeighbors] = NULL;
171
172   // Remove the current node from the list of its nieghbor
173   int iNeighbor = find_neighbor(nNeighbor, nCurrent);
174   if((nNeighbor != NULL) && (iNeighbor >= 0))
175   {
176     remove_neighbor(nNeighbor, iNeighbor);
177   }
178
179   return 0;
180 }
181
182 // -- Function to remove all of the neighbors of a node --
```

```
183 int remove_neighbors(struct Node *nCurrent)
184 {
185   // Make sure the argument is valid
186   if((nCurrent == NULL) || (nCurrent->nNeighbors == NULL))
187   {
188     // A problem removing the neighbors
189     return 1;
190   }
191
192   // Remove the neighbors
193   while(nCurrent->iNeighbors > 0)
194   {
195     remove_neighbor(nCurrent, 0);
196   }
197
198   // Delete the neighbors array
199   free(nCurrent->nNeighbors);
200   nCurrent->nNeighbors = NULL;
201
202   return 0;
203 }
204
205 // -- Function to add a node to a tree --
206 int add_node(struct Tree *tCurrent, struct Node *nNode)
207 {
208   // Make sure the arguments are valid
209   if((tCurrent == NULL) || (nNode == NULL))
210   {
211     // A problem adding the node
212     return 1;
213   }
214
215   // Does a list of nodes need to be created?
```

```
216   if(tCurrent->nNodes == NULL)
217   {
218     // Create the list of nodes
219     tCurrent->iNodeCapacity = 2;
220     tCurrent->nNodes =
221        (struct Node **)malloc(sizeof(struct Node *) *
222        tCurrent->iNodeCapacity);
223     tCurrent->iNodes = 0;
224
225     // NULL out the members of the array
226     int i;
227     for(i = 0; i < tCurrent->iNodeCapacity; i++)
228     {
229       tCurrent->nNodes[i] = NULL;
230     }
231   }
232
233   // Make sure that the node is not already in the tree
234   int i;
235   for(i = 0; i < tCurrent->iNodes; i++)
236   {
237     if(tCurrent->nNodes[i] == nNode)
238     {
239       // Node is already in the tree
240       return 1;
241     }
242   }
243
244   // Does the list of nodes need to be expanded?
245   if(tCurrent->iNodes == tCurrent->iNodeCapacity)
246   {
247     // Create a new array for the nodes
248     tCurrent->iNodeCapacity = (tCurrent->iNodeCapacity * 2) + 1;
```

```
249     struct Node **nNewArray =
250        (struct Node **)malloc(sizeof(struct Node *) *
251        tCurrent->iNodeCapacity);
252
253     // Copy over the contents of the array
254     for(i = 0; i < tCurrent->iNodes; i++)
255     {
256       nNewArray[i] = tCurrent->nNodes[i];
257     }
258
259     // NULL out the other members of the array
260     for(i = tCurrent->iNodes; i < tCurrent->iNodeCapacity; i++)
261     {
262       nNewArray[i] = NULL;
263     }
264
265     // Set the reference to the new array and delete the old
266     free(tCurrent->nNodes);
267     tCurrent->nNodes = nNewArray;
268   }
269
270   // Add the node
271   tCurrent->nNodes[tCurrent->iNodes] = nNode;
272   tCurrent->iNodes++;
273
274   // No problems were encountered
275   return 0;
276 }
277
278 // -- Function to get a node of a tree --
279 struct Node* get_node(struct Tree *tCurrent, int iIndex)
280 {
281   // Make sure the arguments are valid
```

```
282   if((tCurrent == NULL) || (tCurrent->nNodes == NULL) ||
283      (iIndex < 0) || (iIndex >= tCurrent->iNodes))
284   {
285     // A problem getting the node
286     return NULL;
287   }
288
289   // Return the node
290   return tCurrent->nNodes[iIndex];
291 }
292
293 // -- Function to find the index of a node --
294 int find_node(struct Tree *tCurrent, struct Node *nNode)
295 {
296   // Make sure the arguments are valid
297   if((tCurrent == NULL) || (tCurrent->nNodes == NULL) ||
298      (nNode == NULL))
299   {
300     // A problem finding the node
301     return -1;
302   }
303
304   // Search through the nodes
305   int i;
306   for(i = 0; i < tCurrent->iNodes; i++)
307   {
308     // Check to see if this is the node
309     if(tCurrent->nNodes[i] == nNode)
310     {
311       return i;
312     }
313   }
314
```

```
315   // The node was not found
316   return -1;
317 }
318
319 // -- Function to remove a node of a tree --
320 int remove_node(struct Tree *tCurrent, int iIndex,
321    int iDelete)
322 {
323   // Make sure the arguments are valid
324   if((tCurrent == NULL) || (tCurrent->nNodes == NULL) ||
325      (iIndex < 0) || (iIndex >= tCurrent->iNodes))
326   {
327     // A problem removing the node
328     return 1;
329   }
330
331   // Remove the node
332   if(tCurrent->nNodes[iIndex] != NULL)
333   {
334     // Remove the neighbors of the node
335     remove_neighbors(tCurrent->nNodes[iIndex]);
336
337     // Free the node if it is to be deleted
338     if(iDelete == 1)
339     {
340       free(tCurrent->nNodes[iIndex]);
341     }
342   }
343
344   // Do the nodes need to be shifted?
345   int i;
346   for(i = iIndex; i < tCurrent->iNodes - 1; i++)
347   {
```

```
348     tCurrent->nNodes[i] = tCurrent->nNodes[i + 1];
349   }
350
351   // Show that a node was removed
352   tCurrent->iNodes--;
353   tCurrent->nNodes[tCurrent->iNodes] = NULL;
354
355   return 0;
356 }
357
358 // -- Function to remove all of the nodes of a tree --
359 int remove_nodes(struct Tree *tCurrent)
360 {
361   // Make sure the argument is valid
362   if((tCurrent == NULL) || (tCurrent->nNodes == NULL))
363   {
364     // A problem removing the nodes
365     return 1;
366   }
367
368   // Remove the nodes
369   while(tCurrent->iNodes > 0)
370   {
371     remove_node(tCurrent, 0, 1);
372   }
373
374   // Delete the nodes array
375   free(tCurrent->nNodes);
376   tCurrent->nNodes = NULL;
377
378   return 0;
379 }
380
```

```
381 // -- Function to get a leaf of a tree --
382 int get_leaf(struct Tree *tCurrent, struct Node **nLeaf)
383 {
384   // Make sure the argument is valid
385   if((tCurrent == NULL) || (tCurrent->nNodes == NULL))
386   {
387     // A problem getting the leaf
388     return -1;
389   }
390
391   *nLeaf = NULL;
392
393   // Set the current node to a leaf of the tree
394   int i;
395   for(i = tCurrent->iNodes - 1; i >= 0; i--)
396   {
397     // Is the node at index i a leaf?
398     if((tCurrent->nNodes[i] != NULL) &&
399        (tCurrent->nNodes[i]->nNeighbors != NULL) &&
400        (tCurrent->nNodes[i]->iNeighbors == 1))
401     {
402       *nLeaf = tCurrent->nNodes[i];
403       break;
404     }
405   }
406
407   return i;
408 }
409
410 // -- Function to find a dominating set of a tree --
411 struct Tree* domination_algorithm(struct Tree *tCurrent)
412 {
413   // Domination set will be represented by the tree structure
```

```
414   struct Tree *tDominationSet =
415      (struct Tree *)malloc(sizeof(struct Tree));
416
417   // The current node and its neighbor
418   struct Node *nCurrent = NULL;
419   struct Node *nNeighbor = NULL;
420
421   // Remove nodes from the tree one at a time
422   int iDelete = 1;
423   int iIndex;
424   while(tCurrent->iNodes > 1)
425   {
426     // Reset the current node, its neighbor, and if it
427     //    should be deleted
428     nCurrent = NULL;
429     nNeighbor = NULL;
430     iDelete = 1;
431
432     // Get a leaf of the tree
433     iIndex = get_leaf(tCurrent, &nCurrent);
434
435     // Get a neighbor of the current node
436     nNeighbor = get_neighbor(nCurrent, 0);
437
438     // Make sure the nodes are valid
439     if((nCurrent != NULL) && (nNeighbor != NULL))
440     {
441       // See what type the node is
442       if(nCurrent->ntType == BOUND)
443       {
444         // Make the neighbor required
445         nNeighbor->ntType = REQUIRED;
446       }
```

```
447       else if(nCurrent->ntType == REQUIRED)
448       {
449         // Node is not free. Add to the dominating set
450         add_node(tDominationSet, nCurrent);
451         iDelete = 0;
452
453         // See what type the neighbor is
454         if(nNeighbor->ntType == BOUND)
455         {
456           // Make the neighbor free
457           nNeighbor->ntType = FREE;
458         }
459       }
460     }
461
462     // Remove the current node
463     remove_node(tCurrent, iIndex, iDelete);
464   }
465
466   // Is there a remaining node?
467   if((tCurrent->nNodes != NULL) && (tCurrent->iNodes == 1) &&
468      (tCurrent->nNodes[0] != NULL))
469   {
470     iDelete = 1;
471     // See what type the node is
472     if(tCurrent->nNodes[0]->ntType != FREE)
473     {
474       // Add the node to the domination set
475       add_node(tDominationSet, tCurrent->nNodes[0]);
476       iDelete = 0;
477     }
478
479     // Remove the remaining node
```

```
480     remove_node(tCurrent, 0, iDelete);
481   }
482
483   // Return the domination set
484   return tDominationSet;
485 }
```

B.2.2 CAPS

A full listing of the CAPS code for the problem is listed below:

```
001 % a label of a node
002 integer FREE = 0
003 integer BOUND = 1
004 integer REQUIRED = 2
005
006 % -- Find a dominating set of a tree --
007 start specification "Domination"
008   % tree to hold the nodes
009   sequence sT
010   set sDomination
011
012   % node format: index, type, neighbor1, neighbor2, etc
013   sequence sU
014   sequence sV
015   sequence sW
016   integer iV
017
018   % remove nodes from the tree one at a time
019   while |sT| > 1
020     % find a leaf of the tree
021     iV = 0
022     while iV < |sT|
023       sV = sT[iV]
024       % a leaf has only three members. two children and a parent
```

```
025       if |sV| == 3
026         break
027       end if
028       iV = iV + 1
029     end while
030
031     % get the neighbor of the leaf
032     if |sV| >= 3
033       sU = sV[2]
034     end if
035
036     % is the node required?
037     if sV[1] == BOUND
038       sU[1] = REQUIRED
039     else
040       % should the node be added?
041       if sV[1] == REQUIRED
042         add sDomination sV
043         % can the neighbor be freed?
044         if sU[1] == BOUND
045           sU[1] = FREE
046         end if
047       end if
048     end if
049
050     % the node has been processed
051     removeat sT iV
052
053     % remove the node from its neighbor
054     iV = 2
055     while iV < |sU|
056       % is this a leaf?
057       sW = sU[iV]
```

```
058       if sW[0] == sV[0]
059         removeat sU iV
060       end if
061       iV = iV + 1
062     end while
063   end while
064
065   % is there an unprocessed node?
066   if |sT| == 1
067     sV = sT[0]
068     % should the node be added?
069     if sV[1] != FREE
070       add sDomination sV
071     end if
072     % the node has been processed
073     removeat sT 0
074   end if
075 end specification
```

Bibliography

[1] AHO, A. V., KERNIGHAN, B. W., AND WEINBERGER, P. J. *The AWK Prgramming Language*. Addison-Wesley, Reading, Massachusetts, 1988.

[2] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.

[3] ALLEN, J. *Natural Language Understanding*. Addison-Wesley, Reading, Massachusetts, 1995.

[4] ANDERSON, M., ELMSTRØM, R., LASSEN, P. B., AND LARSEN, P. G. Making specifications executable – using iptes meta-iv. In *Euromicro* (1992).

[5] BACKUS, J. W. Preliminary report, specifications for the ibm mathematical formula translating system, fortran. Tech. rep., IBM Corp., New York, November 1954.

[6] BALZER, R. A 15 year perspective on automatic programming. *IEEE Transactions on Software Engineering SE-11*, 11 (November 1985), 1257–1268.

[7] BARSTOW, D. An experiment in knowledge-based automatic programming. *Artificial Intelligence 12*, 1 (August 1979), 73–119.

[8] BARSTOW, D. A perspective on automatic programming. *AI Magazine 5*, 1 (1984), 5–27.

[9] BASTIDE, R. Approaches in unifying Petri nets and the object-oriented approach. In *Proceedings of the Application and Theory of Petri Nets 1995 - Workshop on Object-Oriented Programming and Models of Concurrency* (1995).

[10] BECK, K., AND ANDRES, C. *Extreme Programming Explained: Embrace Change*, 2nd ed. Addison-Wesley, Reading, Massachusetts, 2004.

[11] BHANSALI, S. Architecture-driven reuse of code in kase. In *Proceedings of the 5th International Conference on Software Engineering and Knowledge Engineering* (San Francisco, California, 1993).

[12] BLAINE, L., GILHAM, L., AND SMITH, D. R. Planware: Domain-specific synthesis of high-performance schedulers. In *Proceedings of the 13th Automated Software Engineering Conference* (Los Alamitos, California, 1998), pp. 270–280.

[13] BRATKO, I. *Prolog Programming for Artificial Intelligence*, 3rd ed. Addison-Wesley, Harlow, England, 2001.

[14] BRILL, E. Transformation-based error-driven learning and natural language processing: A case study in part-of-speech tagging. *Computational Linguistics 21*, 4 (1995), 543–566.

[15] CHANDY, K. M., AND MISRA, J. Proofs of distributed algorithms: An exercise. In *Developments in Concurrency and Communication* (1990), Addison-Wesley.

[16] CLARKE, JR., E. M., GRUMBERG, O., AND PELED, D. A. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[17] COAD, P., AND NICOLA, J. *Object-Oriented Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1993.

[18] COCKAYNE, E., GOODMAN, S., AND HEDETNIEMI, S. A linear algorithm for the domination number of a tree. *Information Processing Letters 4*, 2 (1975), 41–44.

[19] COHEN, J. A view of the origins and development of prolog. *Communications of the ACM 31*, 1 (January 1988), 26–36.

[20] COLES, S. An on-line question-answering systems with natural language and pictorial input. In *Proceedings of the 23rd ACM National Conference* (1968), pp. 157–167.

[21] CONSTABLE, R. L. Constructive mathematics and automatic program writers. In *Proceedings of the IFIP Congress* (1971), pp. 229–233.

[22] COPI, I. M., AND COHEN, C. *Essentials of Logic*. Prentice Hall, Upper Saddle River, New Jersey, 2003.

[23] DIJKSTRA, E. W. Go to statement considered harmful. *Communications of the ACM 11*, 2 (1968), 147–148.

[24] FISCHER, B., AND SCHUMANN, J. Autobayes: A system for generating data analysis programs from statistical models. *Journal of Functional Programming 13*, 3 (2003), 483–508.

[25] FLENER, P. *Login Program Synthesis from Incomplete Information*. Kluwer Academic Publishers, Boston, Massachusetts, 1995.

[26] FRIEDBERG, R. M. A learning machine: Part i. *IBM Journal of Research and Development 2* (1958), 2–15.

[27] GELERNTER, D., CARRIERO, N., CHANDRAN, S., AND CHANG, S. Parallel programming in linda. In *International Conference on Parallel Processing* (August 1985), pp. 255–263.

[28] GENESERETH, M. R., AND NILSSON, N. J. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann, Los Altos, California, 1987.

[29] GOLD, E. M. Language identification in the limit. *Information and Control 10*, 5 (1967), 447–474.

[30] GORDON, M. J. C., AND MELHAM, T. F., Eds. *Introduction to HOL*. Cambridge University Press, 1993.

[31] GREEN, C. Theorem-Proving by Resolution as a Basis for Question-Answering Systems. *Machine Intelligence 4* (1969), 183–205.

[32] GREEN, C. C. Application of theorem proving to problem solving. In *Proceedings of the International Joint Conference of Artificial Intelligence* (Washington, DC, May 1969), pp. 219–239.

[33] GREEN, C. C. Results in knowledge based program synthesis. In *Proceedings of the 6th International Joint Conference of Artificial Intelligence* (1979), pp. 342–344.

[34] GREEN, C. C., AND RAPHAEL, B. The use of theorem-proving techniques in question-answering systems. In *Proceedings of the 23rd ACM National Conference* (1968), pp. 169–181.

[35] HACK, M. *Decidability Questions for Petri Nets*. PhD thesis, Massachusetts Institute of Technology, 1975.

[36] HARRIS, Z. S. *String Analysis of Sentence Structure*. The Hague, Mouton, 1962.

[37] HAYNES, T. W., HEDETNIEMI, S. T., AND SLATER, P. J. *Fundamentals of Domination in Graphs*. Marcel Dekker, Inc., 1998.

[38] HOARE, C. A. R. Communicating sequential processes. *Communications of the ACM 21*, 8 (1978), 666–677.

[39] HOARE, C. A. R., BROOKES, S. D., AND ROSCOE, A. W. A theory of communicating sequential processes. *Journal of the ACM 31*, 3 (1984), 560–599.

[40] HOLLAND, J. H. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, Michigan, 1975.

[41] JENSEN, K. Coloured petri nets and the invariant method. *Theoretical Computer Science 14* (1981), 317–336.

[42] JENSEN, K. *Coloured Petri Nets: Basic Concepts, Analysis Methods, and Practical Use*, vol. 1. Springer-Verlag, New York, 1992.

[43] JENSEN, K. *Coloured Petri Nets: Basic Concepts, Analysis Methods, and Practical Use*, vol. 2. Springer-Verlag, New York, 1997.

[44] KANT, E. On the efficient synthesis of efficient programs. *Artificial Intelligence 20*, 3 (May 1983), 253–305.

[45] KENNEDY, K. E. Lattice: An environment for modeling and simulation. In *Spring Simulation Conference* (San Diego, April 2005).

[46] KESTREL INSTITUTE. Specware 4.0 language manual, October 2007. [Online; accessed 29-October-2007].

[47] KOZA, J. Genetic-programming.org, April 2006. [Online; accessed 20-April-2006].

[48] KOZA, J. R. *Genetic Programming: On the Programming of Computers by Natural Selection*. The MIT Press, Cambridge, Massachusetts, 1992.

[49] LAVRAC, N., AND FLACH, P. An extended transformation approach to inductive logic programming. *ACM Transactions on Computational Logic 2*, 4 (October 2001), 458–494.

[50] LIU, H. Montylingua v.2.1, October 2007. [Online; accessed 31-October-2007].

[51] LOWRY, M., PHILPOT, A., PRESSBURGER, T., AND UNDERWOOD, I. Amphion: Automatic programming for subroutine libraries. In *Proceedings of the 9th Knowledge-Based Software Engineering Conference* (Monterey, CA, 1994), p. 2.

[52] MANNA, Z., AND WALDINGER, R. Synthesis: Dreams $\Rightarrow$ programs. *IEEE Transactions on Software Engineering 5*, 4 (July 1979), 294–328.

[53] MCDONALD, J., AND ANTON, J. Specware - producing software correct by construction. Tech. Rep. KES.U.01.3, Kestrel Institute, Palo Alto, California, March 2001.

[54] MCMILLAN, K. L. *The SMV System*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1992.

[55] MITCHELL, T. M. *Machine Learning*. WCB McGraw Hill, Boston, Massachusetts, 1997.

[56] MUGGLETON, S. Duce, an oracle based approach to constructive induction. In *Proceedings of the IJCAI* (Milan, 1987), Morgan Kaufmann, pp. 287–292.

[57] MUGGLETON, S. Inductive logic programming. *New Generation Computing 8*, 4 (1991), 295–318.

[58] MUGGLETON, S. Stochastic logic programs. In *Proceedings of the 5th International Workshop on Inductive Logic Programming* (1995), L. De Raedt, Ed., Department of Computer Science, Katholieke Universiteit Leuven, p. 29.

[59] MUGGLETON, S. Learning stochastic logic programs. In *Proceedings of the AAAI2000 Workshop on Learning Statistical Models from Relational Data* (2000).

[60] MUGGLETON, S., AND BUNTINE, W. Machine invention of first-order predicates by inverting resolution. In *Proceedings of the Fifth International Machine Learning Conference* (Ann Arbor, Michigan, 1988), Morgan Kaufmann, pp. 339–352.

[61] MUGGLETON, S., AND FENG, C. Efficient induction of logic programs. In *Proc. of the 1st Conference on Algorithmic Learning Theory* (Tokyo, Japan, 1990), Ohmsha Publications, pp. 368–381.

[62] NEWELL, A., AND SIMON, H. A. Gps: A program that simulates human thought. 109–124.

[63] NEWMAN, A., SHATZ, S. M., AND XIE, X. An approach to object system modeling by state-based object petri nets. 1–21.

[64] PETRI, C. A. *Kommunikation mit Automaten*. PhD thesis, Schriften des Institutes für Instrumentelle Mathematik, 1962.

[65] PITT, L., AND VALIANT, L. G. Computational limits on learning from examples. *Journal of the ACM 35*, 4 (October 1988), 965–984.

[66] PLOTKIN, G. D. A note on inductive generaliztion. *Machine Intelligence 5* (1970), 153–163.

[67] PLOTKIN, G. D. A further note on inductive generaliztion. *Machine Intelligence 6* (1971), 101–124.

[68] QUINLAN, J. R. Learning logical definitions from relations. *Machine Learning 5*, 3 (August 1990), 239–266.

[69] REYNOLDS, J. C. Transformational systems and the algebraic structure of atomic formulas. *Machine Intelligence 5* (1970), 135–151.

[70] RICH, C., AND WATERS, R. C. Automatic programming: Myths and prospects. *IEEE Computer 21*, 8 (August 1988), 40–51.

[71] RUSSELL, S. J., AND NORVIG, P. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, New Jersey, 1995.

[72] SAMMUT, C. A. *Learning Concepts by Performing Experiments*. PhD thesis, Department of Computer Science, University of New South Wales, 1981.

[73] SAMMUT, C. A., AND BANERJI, R. B. Learning concepts by asking questions. In *Machine Learning: An Artificial Intelligence Approach*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Eds. Morgan Kauffmann, 1986, pp. 167–192.

[74] SHAPIRO, E. Y. *Algorithmic Program Debugging*. The MIT Press, Cambridge, Massachusetts, 1983.

[75] SIMON, H. A. Experiments with a heuristic compiler. *Journal of the ACM 10*, 4 (October 1963), 493–506.

[76] SLAGLE, J. R. Experiments with a deductive question-answering program. *Communications of the ACM 8*, 12 (December 1965), 792–798.

[77] SMITH, D. R. Constructing specification morphisms. *Journal of Symbolic Computation 15*, 5/6 (1993), 571–606.

[78] SOLOMONOFF, R. J. A formal theory of inductive inference. *Information and Control 7* (1964), 1–22,224–254.

[79] STICKEL, M., WALDINGER, R., LOWRY, M., PRESSBURGER, T., AND UNDERWOOD, I. Deductive composition of astronomical software from subroutine libraries. In *Proceedings of the Twelfth International Conference on Automated Deduction (CADE-12)* (Nancy, France, June 1994), pp. 341–355.

[80] STOLZ, W. S., TANNENBAUM, P. H., AND CARSTENSEN, F. V. A stochastic approach to the grammatical coding of english. *Communications of the ACM 8*, 6 (1965), 399–405.

[81] SUTHERLAND, R. Online plain text english dictionary, 2005. [Online; accessed 14-October-2005].

[82] TURING, A. M. Intelligent machinery. Can be found in Machine Intelligence 4, 1948.

[83] TURING, A. M. Computing machinery and intelligence. *Mind 59*, 236 (October 1950), 433–460.

[84] UNIVERSITY, C. M. Cmu pronouncing dictionary, 2005. [Online; accessed 14-October-2005].

[85] VALIANT, L. G. A theory of the learnable. *Communications of the ACM 27*, 11 (November 1984), 1134–1142.

[86] VOUTILAINEN, A. Morphological disambiguation. In *Constraint Grammar: A Language-Independent System for Parsing Unrestricted Text*, F. Karlsson, A. Voutilainen, J. Heikkilä, and A. Anttila, Eds. Mouton de Gruyter, 1995, pp. 165–284.

[87] WAHLS, T., LEAVENS, G. T., AND BAKER, A. L. Executing formal specifications with constraint programming. Tech. Rep. TR97-12a, Department of Computer Science, Iowa State University, Iowa, 1998.

[88] WALDINGER, R. J., AND LEE, R. C. T. Prow: A step toward automatic program writing. In *Proc. of the 1st IJCAI* (Washington, DC, 1969), pp. 241–252.

[89] WARREN, D. An abstract Prolog instruction set. Technical Note 309, SRI International, October 1983.

[90] WOODCOCK, J., AND DAVIES, J. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall, London, Engalnd, 1996.

[91] ZELLE, J. M., AND MOONEY, R. J. Learning to parse database queries using inductive logic programming. In *Proceedings of the Thirteenth National Conference on Aritificial Intelligence* (Portland, OR, August 1996), pp. 1050–1055.