

eCGE: A MULTI-PLATFORM PETRI NET EDITOR

By

DAVID DUGAN

A thesis submitted in partial fulfillment of
the requirements for the degree of

MASTER OF SCIENCE

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

MAY 2005

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of
DAVID DUGAN find it satisfactory and recommend that it be accepted.

Chair

ACKNOWLEDGEMENT

The eCGE project has greatly benefited from the support and advice of a number of individuals, most notably my advisor, Dr. Frederick Sheldon, and my family (including the cat). Dr Sheldon was my undergraduate professor at University of Colorado at Colorado Springs in 1998 for a computer architecture class. He nominated me to work as a graduate assistant at WSU. During my time at WSU, he has been a constant source of stimulation and encouragement for which I am forever grateful. Dr Sheldon moved to Oak Ridge National Laboratories and is unable to fulfill the administrative role in regard to the evaluation of the thesis. However, during his time at Oak Ridge he continued to provide guidance for this Project and made many useful suggestions to improve the quality of this project. I am deeply indebted to Dr Anneliese Andrews for volunteering for that administrative role and for her support of this project.

eCGE: A MULTI-PLATFORM PETRI NET EDITOR

ABSTRACT

by David Dugan, M.S.
Washington State University
May 2005

Chair: Frederick T. Sheldon

This thesis describes the design and application of the enhanced Petri Net Graphical Editor (eCGE) application. This project implements the core features of a Stochastic Petri net modeling tool and is implemented in Java for portability. It provides an interactive environment for developing and visualizing a Petri net model. The user interface is graphical with highly interactive layout and editing features, and is designed to detect syntactic errors in a Petri net model as it is developed. This tool contains a number of improvements over the original application, such as an improved design, which provides the ability to read and write a file in the C-based Stochastic Petri net Language (CSPL) format, and features to help organize a model.

LIST OF PUBLICATIONS

Frederick T. Sheldon and David Dugan. “Stochastic Petri Nets and Discrete Event Simulation: A Comparative Study of Two Formal Description Methods,” *IEEE 3rd Workshop on Formal Descriptions and Software Reliability*, San Jose, CA, October 7, 2000.

TABLE OF CONTENTS

ACKNOWLEDGEMENT	III
ABSTRACT	IV
LIST OF PUBLICATIONS.....	V
TABLE OF CONTENTS.....	VI
LIST OF FIGURES	XII
LIST OF TABLES	XV
CHAPTER ONE	1
INTRODUCTION	1
1.1 PROBLEM DEFINITION	1
1.2 MOTIVATION / GRAPHICAL EDITOR FOR PETRI NETS	1
1.3 SCOPE	3
1.4 ORGANIZATION	4
CHAPTER TWO	5
METHODS OF MODELING SYSTEM BEHAVIOR.....	5
2.1 INTRODUCTION.....	5
2.2 BACKGROUND ON FORMALLY MODELING SYSTEM BEHAVIOR.....	5
2.3 PERFORMANCE ANALYSIS AND FORMAL METHODS	6
2.4 DEFINING THE SYNTAX OF A MODELING LANGUAGE	7
2.5 ELEMENTS OF A LANGUAGE DEFINITION	8

2.6 SYNTAX	9
2.7 GRAPHS	9
CHAPTER THREE	11
BACKGROUND ON PETRI NETS.....	11
3.1 IN THE BEGINNING	11
3.2 TIME IN PETRI NETS	12
3.3 PETRI NETS AND PERFORMANCE MODELING	13
3.4 NUMERICAL ANALYSIS OF A STOCHASTIC PETRI NET MODEL	15
3.5 SURVEY OF RELATED WORK.....	16
3.5.1 <i>Petri</i>	17
3.5.2 <i>DSPNexpress2000</i>	17
3.5.3 <i>CPN2000</i>	17
3.5.4 <i>UltraSAN</i>	17
3.5.5 <i>Doodle</i>	17
CHAPTER FOUR.....	19
CONTRIBUTION OF THIS WORK.....	19
4.1 STARTING POINT FOR THE PROJECT.....	19
4.2 IMPROVEMENTS IN READING AND SAVING A FILE	20
4.2.1 <i>Background</i>	20
4.2.2 <i>File Format Compatibility</i>	21
4.2.3 <i>CSPL Conversion Example</i>	22
4.3 BASIS FOR THE ENHANCED CGE	22

4.4	PAST AND PRESENT CONTRIBUTIONS TO CGE	24
4.5	DESIGN OF THE ENHANCED CGE	25
4.5.1	<i>Design Approach</i>	26
4.5.2	<i>Levels of Abstraction</i>	27
CHAPTER FIVE		31
ECGE TOOL OVERVIEW AND IMPLEMENTATION DETAILS.....		31
5.1	INTRODUCTION.....	31
5.2	USER INTERFACE LEVEL	31
5.2.1	<i>Interaction Techniques</i>	33
5.2.2	<i>Drawing Window</i>	33
5.3	DESIGN LEVEL	34
5.4	ALGORITHMIC LEVEL	35
5.4.1	<i>AVL Tree</i>	36
5.4.2	<i>CSPL Parser</i>	36
5.4.3	<i>Mouse Handler</i>	36
5.4.4	<i>Multiple Document Prototype</i>	38
5.4.5	<i>Arc Line Calculations</i>	38
5.5	CASE STUDY 1: USE OF A PRE-EXISTING MODEL	38
5.5.1	<i>Importing the model into eCGE</i>	39
5.5.2	<i>Description of the Model</i>	41
5.5.3	<i>Steps used to develop the model</i>	41
5.6	CASE STUDY 2: LAYOUT OF A MODEL	43
5.6.1	<i>Conflict in a Petri net</i>	44

5.6.2	<i>Confusion in a Petri net</i>	44
5.6.3	<i>Detecting the structure of the model through the layout</i>	45
CHAPTER SIX		47
THE PETRI NET LANGUAGE PRIMITIVES		47
6.1	ELEMENTS OF A PETRI NET	47
6.2	PLACES	47
6.3	TRANSITIONS	47
6.3.1	<i>Selectively Disabling a Transition</i>	48
6.4	ARCS	49
6.5	INTRODUCING SPNP’S C-BASED STOCHASTIC PETRI NET LANGUAGE	49
6.5.1	<i>Specification of “options”</i>	50
6.5.2	<i>Specification of “net”</i>	51
6.5.3	<i>Assert Function</i>	52
6.5.4	<i>Other Functions: ac_init, ac_reach, and ac_final</i>	53
CHAPTER SEVEN		54
CONCLUSIONS		54
7.1	SUMMARY	54
7.2	FUTURE PLANS	54
BIBLIOGRAPHY		56
APPENDIX A		65
LAYOUT ALGORITHMS		65

A.1 INTRODUCTION.....	66
A.2 GENERAL PRINCIPLES OF A GRAPH LAYOUT ALGORITHM	66
<i>A.2.1 Spring Algorithm</i>	67
<i>A.2.2 Tree Algorithm</i>	67
<i>A.2.3 Random Algorithm</i>	67
A.3 INTERFACE FOR DESIGNING A GRAPH LAYOUT ALGORITHM	67
APPENDIX B.....	69
DESIGN DOCUMENTATION	69
B.1 INTRODUCTION.....	70
B.2 ARCHITECTURAL MODEL OF THE PROJECT	70
B.3 MOUSE HANDLER TIMING DIAGRAMS	73
<i>B.3.1 Single Click</i>	74
<i>B.3.2 Double Click</i>	74
<i>B.3.3 Mouse Drag</i>	75
B.4 AVL TREE DOCUMENTATION	75
<i>B.4.1 AVL Tree Data Structures</i>	75
<i>B.4.2 Insertion</i>	76
<i>B.4.3 Deletion</i>	80
<i>B.4.4 Detailed description of the list traversal functions</i>	87
APPENDIX C.....	90
USER'S MANUAL	90
C.1 INTRODUCTION.....	91

C.2	TOOLBAR	91
C.3	MENU BARS	91
C.3.1	<i>File Menu</i>	92
C.3.2	<i>Edit Menu</i>	92
C.3.3	<i>Functions Menu</i>	92
C.3.4	<i>Algorithms Menu</i>	93
C.4	PLACES	93
C.5	TRANSITIONS	93
C.5.1	<i>Transition Properties</i>	94
C.5.2	<i>Firing Rate</i>	94
C.5.3	<i>Guard Functions</i>	95
C.6	ARCS	96
C.6.1	<i>Cardinality</i>	96
C.7	PARAMETERS	97
C.7.1	<i>General Parameters</i>	97
C.7.2	<i>Output File Parameters</i>	99
C.7.3	<i>Markov Chain Options</i>	101
C.7.4	<i>Simulation Options</i>	103

LIST OF FIGURES

FIGURE 1: DATA STREAM.....	8
FIGURE 2: CSPL FILE CONVERSION	22
FIGURE 3: CGE FIRST ITERATION	24
FIGURE 4: CGE SECOND ITERATION.....	24
FIGURE 5: CGE THIRD ITERATION	24
FIGURE 6: CGE FOURTH ITERATION.....	25
FIGURE 7: DESIGN APPROACH.....	26
FIGURE 8: DESIGN LEVEL COMPONENTS.....	29
FIGURE 9: SAMPLE ERROR DIALOG BOX	32
FIGURE 10: INITIAL ARRANGEMENT OF CASE STUDY MODEL	39
FIGURE 11: RANDOM ARRANGEMENT OF CASE STUDY MODEL.....	40
FIGURE 12: ORGANIZED LAYOUT OF CASE STUDY MODEL	40
FIGURE 13: CASE STUDY MODEL IN CSPL FORMAT	43
FIGURE 14: CONFLICT IN A PETRI NET	44
FIGURE 15: CONFUSION IN A PETRI NET	44
FIGURE 16: EXAMPLE OF AMBIGUITY INTRODUCED BY CONFUSION.....	44
FIGURE 17: CONFUSION IN A PETRI NET – CSPL VERSION.....	46
FIGURE 18: PN PRIMITIVES.....	47
FIGURE 19: CLASS ARCHITECTURE FOR eCGE.....	71
FIGURE 20: CLASS ARCHITECTURE FOR MOUSE HANDLER	72
FIGURE 21: CLASS ARCHITECTURE FOR A DOCUMENT	73

FIGURE 22: MOUSE HANDLER FLOW CHART	74
FIGURE 23: PARSING A SINGLE CLICK	75
FIGURE 24: PARSING A DOUBLE CLICK.....	75
FIGURE 25: PARSING A MOUSE DRAG.....	76
FIGURE 26: RIGHT ROTATION	79
FIGURE 27: DOUBLE ROTATION, LEFT SIDE	79
FIGURE 28: LEFT ROTATION.....	80
FIGURE 29: DOUBLE ROTATION, RIGHT SIDE	80
FIGURE 30: AVL DELETE, CASE 1	82
FIGURE 31: AVL DELETE, CASE 2	83
FIGURE 32: AVL DELETE, CASE 3	83
FIGURE 33: RE-BALANCE LEFT SIDE, CASE 1	85
FIGURE 34: RE-BALANCE LEFT SIDE, CASE 2	86
FIGURE 35: RE-BALANCE RIGHT SIDE, CASE 1	87
FIGURE 36: RE-BALANCE RIGHT SIDE, CASE 2	87
FIGURE 37: FUNCTION NEXT CASE	90
FIGURE 38: TOOLBAR	92
FIGURE 39: PLACE ATTRIBUTES DIALOG BOX	94
FIGURE 40: TRANSITION ATTRIBUTE DIALOG BOX	96
FIGURE 41: GUARD FUNCTION DIALOG BOX	96
FIGURE 42: ARC ATTRIBUTES DIALOG BOX.....	97
FIGURE 43: GENERAL PARAMETERS	99
FIGURE 44: REACHABILITY GRAPH OPTIONS	100

FIGURE 45: OTHER OUTPUT OPTIONS	101
FIGURE 46: MARKOV PARAMETERS.....	103
Figure 47: Simulation Parameters	105

LIST OF TABLES

TABLE 1: COMPONENTS OF THE ENHANCED CGE.....	23
--	----

CHAPTER ONE

INTRODUCTION

1.1 Problem Definition

Petri Nets are an established means of visualizing certain types of systems. The aim of this project is to implement a graphic editor for developing and editing Petri Net models. This graphic editor should have the ability to create and modify the graphs of the nets and, ideally, it should demonstrate the progress of the tokens through the net. It can also assist the user in the layout of a model.

A software-based modeling tool can provide many benefits in developing a Petri net model. First, a model can be created quickly from a palette of pre-defined elements. Then as a model evolves, it can be quickly modified to reflect these changes. The design and interface of these tools have reflected the state of the art in computer software design. For example, the modeling language of many early tools was textual. More recently developed tools have used graphical means to represent a Petri net model.

However, modeling tools should go beyond an electronic drawing board to provide features to help organize a model. To this end, the eCGE application provides an interface for developing graph layout algorithms. The Spring and Tree Algorithms have been implemented to provide an example of how to use this interface [1].

1.2 Motivation / Graphical Editor for Petri Nets

The motivation for this project is to develop a tool to generate a Stochastic Petri net model graphically rather than textually. This is accomplished by implementing an easy to use graphical editor, which provides a set of tools to develop and visualize a Stochastic Petri Net model. At

the user interface level, the goal of the project is to minimize any unnecessary effort in translating an actual system into the Petri net formalism. Extensive consistency checking enforces the semantic correctness of a model as it is developed. In addition, it is possible to import a Petri Net model written in the C-based Stochastic Petri net Language (CSPL) [2].

Representing knowledge via a graph is not as trivial a task as it seems. This is especially true when the information to represent is inherently abstract, such as many of the constructs in systems analysis and design. For example, a *process* is an intangible concept that cannot be touched or adequately visualized in terms of its inherent structure or form. Petri nets can artificially represent the abstract concept of processes through the movement of tokens in a model. The use of such artificial substitutes is inevitable because a direct isomorphism does not exist between the real-world concept and its Petri net representation.

In order to represent abstract information, a decision must be made on how to partition the real-world knowledge into the various Petri net constructs and represent the various elements onto the drawing window so that it is intuitive. Conceptually, these basic principles for representing abstract information fall into two main categories: *information distribution* and *spatial organization*. These principles are used by the eCGE application in creating a visual grammar of the Petri net modeling formalism.

Information distribution refers to the level of partitioning of information into the elementary Petri net constructs (arcs, places, transitions, and guards). The various types of information may be represented at different levels of granularity. For example, the main window of a Petri net model shows the overall graphical structure of a model, while dialog boxes can be opened to show the attributes of an individual element. Another example is hierarchical Petri nets, where a transition can represent another Petri net model.

Spatial organization refers to the way meaning is conveyed through the layout within the presentation space. For example, graph layout algorithms can be used to help organize the elements of a model. The eCGE application currently implements two such algorithms. These algorithms provide differing representations of the Petri net elements. Even though these representations are isomorphic (or informationally equivalent), they will not necessarily be equally effective in conveying the underlying meaning of a particular model [3].

The layout of a Petri net model can be useful to communicate the underlying meaning of the net. The eCGE application provides a framework for developing a variety of graph layout algorithms for organizing the elements of a Petri net model. It may be useful to be able to select a particular layout based on the model at hand. The results of each algorithm provide different perceptual cues that affect the readability of a model. Some models may be easier to develop or less error-prone with a particular layout.

1.3 Scope

The eCGE tool presented in this paper is based on and extends the work done by Gravelle [4]. The original CGE was a proof of concept demonstration that a graphical interface could be built to generate the CSPL coding language constructs. The basic idea was to make the original CGE tool easier to maintain and extend in a number of areas. This project extends the original application in a number of areas:

- The application is written in Java to enhance portability and extensibility.
- The design is based on the concepts of object-oriented analysis/design, rather than procedural methodology.
- The application has the ability to read and write a Petri net model in two formats: CSPL and an application-specific format.

- An object-oriented interface was developed for creating a graph layout algorithm.

1.4 Organization

This document is organized as follows. Chapter 2 provides a brief introduction in the use of formal methods in modeling a system. Chapter 3 presents some background in the Petri nets and describes a number of related tools. Chapter 4 presents the methodology used for developing this project and a comparison with the previous version of CGE. Chapter 5 gives an overview of the actual design of the application. Chapter 6 describes the Petri net elements included in this project. Chapter 7 concludes this thesis with a plan for future studies on this topic. Appendix A provides background on the graph layout algorithms. Appendix B contains the design documentation for the eCGE project. Appendix C provides the user's manual for the eCGE application.

CHAPTER TWO

METHODS OF MODELING SYSTEM BEHAVIOR

2.1 Introduction

The design and the correct implementation of software are difficult tasks, for which many formal methods have been proposed and employed. The aim of such models is to give precise, unambiguous, and complete specifications, which are of value in implementing the software system and in proving correctness properties. Much research has gone into developing formal methods that can rigorously demonstrate that an implementation is consistent with the requirements. A number of these is based on existing modeling paradigms, such as state machines, Petri nets, or process algebras.

2.2 Background on Formally Modeling System Behavior

The formal methods model encompasses a set of activities that lead to mathematical specification of an actual system. Formal methods enable a user to specify, develop, and verify a computer-based model by applying a rigorous mathematical notation. When formal methods are used during design, they serve as a basis for program verification and therefore enable the user to discover and correct errors that might otherwise go undetected.

As the size and complexity of software systems has increased, it has become increasingly more difficult for software designers to produce a quality product that meets the end user's requirements in a timely manner. The problem is often a symptom of the informal nature of the design process. Requirements informally agreed upon are often misunderstood or misinterpreted, leading to deficiencies in the final software product. In addition, these deficiencies are often not discovered until very late in the development process or after

implementation, causing delays in delivery. In order to reduce the cost of developing reliable systems that provide the desired functionality, it is necessary to provide a means to test the formal specification early in the software life cycle.

As a result, researchers have begun to restructure the design process, introducing approaches and tools, such as formal specification and verification, which enable a designer to rigorously demonstrate that an implementation is consistent with its requirements. Demonstrating that code is consistent with its critical requirements is a difficult process. However, the process can be made traceable by verifying the design at every step.

2.3 Performance Analysis and Formal Methods

Traditionally, formal correctness verification and performance analysis have been two separate fields. Whereas the validation and verification processes are based on formal techniques, the classical approach to performance is based on human ingenuity and experience, and consists of devising abstract models that can be analyzed by simulation or by applying stochastic process theory. A key problem of the traditional performance evaluation approach lies in the credibility of the model: the functional equivalence between a model and the actual system is almost impossible to prove. This is not the case for the model used for verification, since the model is the specification of the actual system. This consideration suggests that formal methods should be used for performance analysis as well as for formal verification. Combining performance analysis with formal verification provides a number of other advantages. For example, it makes performance prediction possible in the early design phases, thus avoiding costly redesign, and facilitates the automation of the performance analysis process [5].

As already noted in [6], the use of a formal description language as a paradigm for performance modeling requires the extension of the language with temporal and probabilistic

specifications. The temporal specifications are necessary to describe the time lapse between consecutive events. The probabilistic specifications are necessary to describe the selection among different possible behaviors.

Many researchers [7]-[16] have considered the two types of extensions separately, the timing extension for formal verification of time-critical systems, and the probabilistic extension for probabilistic verification or testing, when an exhaustive validation is impossible. Each extension gives a reasonable insight into the related problems, but merging probabilistic and timing information for performance modeling involves new aspects [17]. Pioneering work in this respect was done in the area of Petri nets, with the formulation of some well-known timed and probabilistic extensions such as stochastic Petri Nets [18]-[20] and their offspring [21]-[26], and Timed Petri Nets [27]-[29].

2.4 Defining the Syntax of a Modeling Language

The timed and probabilistic extensions to the Petri net formalism present a challenge for the definition of a Petri net modeling language. Much research has been done to clarify some of the notions arising in defining a modeling language for the Petri net constructs. The timed and probabilistic extensions to the Petri net formalism have led to a number of variations in the modeling language. The differences between these extensions can lead to confusion.

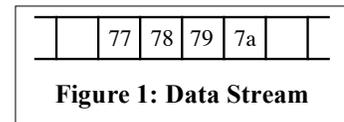
The same basic syntax is used in each case, but the underlying meaning, or semantics, differs between variants. One area of interest is distinguishing a language's notation, or *syntax*, from its meaning, or *semantics*, as well as recognizing the differences between variants of syntax and semantics in their nature, purpose, style, and use [30].

2.5 Elements of a Language Definition

Much has been said about the distinction between the purist notation of information and its semantic representation as data. Extracting or understanding the information encoded in data requires an interpretation - a mapping that assigns a meaning to each (legal) piece of data.

Understanding the difference between syntax and semantics helps avoid confusion. A major source of confusion is the mixing of the data and information notations. In one case, two pieces of data might encode the same information, for example, “June 20, 2001” and “The last day of the first spring in the third millennium.”

In another case [31], the same piece of data might have several meanings and therefore denote different information for different



people or applications. An example is the sequence of hexadecimal numbers shown in Figure 1. Taken as a raw data stream, this data has no meaning. Its meaning depends on how it is interpreted. The following list shows a number of possible interpretations based on common data types.

- A sequence of 1-byte characters: ‘wxyz’
- A 2-byte integer: $7778_{16} = 30,584_{10}$
- A 4-byte integer: $7778797a_{16} = 2,004,384,122_{10}$
- A 4-byte floating-point number: 2.00

Just as people use natural language to communicate with each other, machines use machine-readable languages for communication. Both kinds of language - whether they are natural, artificial, or hardware description languages - contain a great variety of meaningful language elements. Communication stakeholders must therefore agree on the language, which in turn fixes the data set that they can communicate.

Accordingly, a language consists of a syntactic notation, which is possibly an infinite set of legal elements, together with the meaning of those elements, typically expressed by relating the syntax to a semantic domain. Thus, any language definition must consist of the set of syntactic elements, the semantic domain and a mapping from the syntactic elements to the semantic domain [30].

2.6 Syntax

Textual languages are symbolic in spirit, and their basic syntactic expressions are put together in linear character sequences. In contrast, the basic expressions in iconic languages are small pictorial signs that depict elements. An iconic language can be more intuitive than a textual language, but can be more difficult to design. Iconic and diagrammatic languages are proving extremely helpful in software and systems development. In a theoretical sense, textual languages and visual or diagrammatic ones have no principle difference, but when rigor and formality are called for, properly defining diagrams seems much harder [30].

It is possible for a human reader to guess the meaning of most terms, since a good language designer probably chooses keywords and special symbols with a meaning similar to some accepted norm, but a computer cannot act on such assumptions. To be useful in the computing arena, any language - whether it is textual or visual or used for programming, requirements, specification, or design - must come complete with rigid rules that clearly state allowable syntactic expressions and give a rigid description of their meaning.

2.7 Graphs

Visualizing information, especially information of complex and intricate nature, has for many years been the subject of considerable work by many people. The interesting information

for this thesis is not quantitative, but rather of a structural, set-theoretical, and relational nature. Consequently, the focus is on diagrammatic paradigms that are essentially topological in nature rather than geometric.

A graph, in its most basic form, is simply a set of points, or nodes, connected by edges or arcs. Its role is to represent a (single) set of elements S and some binary relation R on them. The precise meaning of the relation R is part of the application and has little to do with the mathematical properties of the graph itself. Certain restrictions on the relation R yield special classes of graphs that are of particular interest, such as ones that are connected, directed, acyclic, planar, or bipartite.

Graphs are used extensively in virtually all branches of computer science. The elements represented by the nodes in these applications range from the most concrete (i.e., physical gates in a circuit diagram) to the most abstract (i.e., complexity classes in a classification scheme). The edges have been used to represent almost any conceivable kind of relation, including ones of temporal, causal, functional, or epistemological nature. Graphs can be modified to support a number of different kinds of elements and relationships [32].

CHAPTER THREE

BACKGROUND ON PETRI NETS

3.1 In the Beginning

A somewhat less widely used extension of graphs is the formalism of Petri nets. The Petri net, first described in Carl Adam Petri's dissertation in 1962 [33], is a tool for describing and studying systems that are characterized as being concurrent, asynchronous, distributed, parallel, non-deterministic, and/or stochastic. In general, they can be applied to any system that can be described graphically. As a graphical tool, Petri Net can be used as a visual-communication aid similar to flow charts, block diagrams, and networks. In addition, tokens are used in these nets to simulate the dynamic and concurrent activities of systems. As a mathematical tool, it is possible to set up state equations, algebraic equations, and other mathematical models governing the behavior of systems. However, in modeling a large system the benefits of using Petri nets tend to be lost due to an excessive amount of work needed for analysis.

Petri nets have proved to be of great value in many applications over the years. Some examples of the successful use of Petri nets can be found in the areas of performance evaluation and communication protocols. Many promising areas have been explored to include distributed and concurrent systems (hardware, software, and logistical) and the analysis of complex state systems (languages, etc.). In addition, some very far reaching ideas have been explored, including local area networks, legal systems, human factors, neural networks, digital filters, and decision models [17].

3.2 Time in Petri Nets

Petri nets were originally conceived to describe software behavior and functionalities in a time-independent fashion. A model described a system in terms of a sequence of possible states. The amount of time spent in each state was not considered. This is adequate when dealing with non-time-critical applications, for example, when timing does not affect correctness, but only performance. If this is not the case, formal methods including quantitative temporal specifications are necessary in order to verify self-consistency, as well as the desired functional properties. As a result, Petri nets have been extended to include the concept of time [34]-[35].

The addition of time to the Petri net formalism is accomplished by associating a *firing time* to each transition. This firing time specifies the time that the transition has to be enabled before it can actually fire. Different types of transitions can be distinguished depending on their associated delay, for instance:

- Deterministic – The firing rate is a fixed time. This model has the concept of an underlying clock that ticks off time and can have synchronous transition firings [36]. This approach is useful for synchronous transition firings that are predictable and happen at regular time intervals. A special case is an immediate transition, which fires after a zero time delay.
- Stochastic - The Stochastic Petri net model has the underlying notion of time as a random variable where events are asynchronous. The firing time is based on a random distribution function. The most common distribution is exponential, represented with a timed transition. In theory, it is possible to use other distributions, as is done in specifying regenerative simulation models. (i.e., relaxing the exponential assumption on the transition firing rate). Stochastic Petri nets are useful for asynchronous events

that happen at non-predictable or random timing - for example, studying the performance and dependability issues of a system.

- Hierarchical (or high-level Petri nets) - The firing time is based on the analysis of another Petri net model, sometimes called a sub-net [37]. Support for hierarchical Petri nets are not built into the current version of the CGE application.

There have been several modifications of these basic models, in addition to the various semantic interpretations of firing rules. First, the Generalized Stochastic Petri Nets extend Stochastic Petri nets by adding the concept of instantaneous transitions [38]. Second, the Extended Stochastic Petri net add features to relax the exponential assumption on the transition firing rates [24]. Third, the Generalized Timed Petri net adds a selection probability to groups of transitions but maintains the discrete time concept [28]. Fourth, Stochastic Petri nets with ‘new better than used’ distributions for firing times are being used to specify regenerative simulation models [39]. All of these models have had analysis software written to support their application.

The exponential distribution models the time between independent events or a process time which is memoryless (knowing how much time has passed gives no information about how much additional time will pass before the process is complete); for example, the times between the arrival of a large number of customers acting independently of each other. The exponential is a highly variable distribution and is sometimes overused because it often leads to mathematically tractable models. If the time between events is exponentially distributed, then the number of events in a fixed period of time is Poisson.

3.3 Petri Nets and Performance Modeling

System reliability, maintainability, availability, and performance analyses are important and complex. Several modeling methods, such as fault trees, Markov Chains, and Stochastic Petri

Nets are used for such analyses. Fault trees are easy to use in developing system models, but are not suitable to describe systems with repairable components. The Markov chain model becomes too complicated to specify by hand for any real-life system.

Petri nets provide a more versatile environment than fault trees and Markov Chains. Petri nets can be used to show the dynamic behavior as well as the static aspects of a system. For example, the addition of time to Petri nets provides a paradigm for the construction of performance models or protocols [14].

Markov analysis provides a means of analyzing the reliability and availability of systems whose components exhibit strong dependencies. Other system analysis methods (such as those employed for fault tree analysis) often assume component independence which may lead to optimistic predictions for the system availability and reachability parameters.

The major drawback of Markov models is that for large systems the underlying Markov chain is exceedingly large and complicated and difficult to construct. However, Markov chains may be used to analyze smaller systems with strong dependencies requiring accurate evaluation. Other analysis techniques, such as fault tree analysis and simulation, may be used to evaluate large systems using simpler probabilistic calculation techniques. Large systems which exhibit strong component dependencies in isolated and critical parts of the system may be analyzed using a combination of Markov analysis and simpler quantitative models.

When reliability and availability analyses of a system are performed, a Markov Chain is often used [40]-[42]. If the Markov Chain is constructed manually, the analysis is often limited to small systems modeled at the highest level. Without the proper tools, it is difficult to develop a Markov model of a large system detailed enough to include the essential attributes of the modeled system. A higher-level “language” is needed for the description of the system to allow

the automatic generation of the Markov chain. The stochastic Petri net model provides such a language [43]. eCGE provides a graphical means to specify a stochastic Petri net model. A tool such as SPNP can be used to automatically convert a model into a Markov chain and solved numerically for the desired properties.

3.4 Numerical Analysis of a Stochastic Petri Net Model

Methods of obtaining numerical data from a Petri net model are based on well-developed mathematical theory. A Stochastic Petri net is usually converted to the equivalent Markov Chains; the Markov Chain is then used to obtain numeric results [44]-[45]. Though this approach is widely used, its major weakness is the complexity problem: the number of states in the converted CTMC grows exponentially with the number of reachable states of the Petri net model. Thus, the analysis of a Stochastic Petri net with Markov Chains works well only when the system model is relatively small.

An alternative approach is to use numerical, computer-based simulation to imitate the behavior of the system over time. Data is collected from the simulation as if a real system was being observed. However, simulation requires a large number of runs to obtain statistically valid results. This can be very time consuming, especially for certain types of problems (for example, modeling a rare event).

The actual implementation of such a numerical solver is outside the scope of this project. A number of software packages have been developed to provide this type of analysis without the need to translate a Petri net model into an equivalent Markov chain. It is possible to use a model developed in eCGE as input to one such tool, the Stochastic Net Package (SPNP) [2]. It uses analytical solutions whenever possible, and simulations when necessary. In a sense, this project

can be viewed as a wrapper application for SPNP, using a graphical user interface to abstract the details of CSPL. eCGE can check for syntactic errors but it cannot check for runtime errors.

3.5 Survey of Related Work

Despite the advantages of Petri nets in modeling dynamic system behavior, Petri nets are largely confined to academia. A number of concerns about its applicability in a business environment are presented here:

- The development of a Petri net model tends to be quite time consuming and expensive. Skimping on resources for modeling and analysis may result in a model that does not sufficiently represent the actual system.
- Because few software developers have the necessary background to work with Petri nets, extensive training is required. Translating the essential characteristics of an actual system into a Petri net model is an art that is learned over time and through experience.
- It is difficult to use the models as a communication mechanism for technically unsophisticated customers.

These concerns notwithstanding, it is likely that research in user-friendly tools will assist in the increased use of Petri nets for the analysis of certain types of problems. However, while a graphical modeling tool removes the learning curve due to language syntax, it does not remove the need for analyzing the procedural logic of a real world system and the debugging required to create an accurate model.

This section provides a brief survey of systems that have been developed for modeling and analyzing Stochastic Petri nets. Studying these tools, particularly CPN2000, was useful in gaining insight into improving the interface and design of CGE.

3.5.1 *Petri*

Petri is a tool for editing and analyzing Petri nets. It supports three forms of analyzing a Petri net: simulation, reachability analysis, and invariant analysis. In addition, the tool provides an interactive debugger which can be used to study the runtime behavior of a model [46].

3.5.2 *DSPNexpress2000*

DSPNexpress2000 takes Unified Modeling Language (UML) [47] specifications enhanced by timing constraints, and maps them into a Generalized semi-Markov Process (GSMP). This program can evaluate deterministic Markov chains specified as either a Petri net model or as system specifications in UML [48].

3.5.3 *CPN2000*

CPN2000 is a graphical tool for editing and simulating Colored Petri Net. This tool contains a graphical editor, as well as advanced simulation, analysis, and data collecting facilities. One interesting aspect of this project is its use of video in the design process to research how modelers use the program. This allowed the developers to find areas in which to improve the user interface and design of the program [49].

3.5.4 *UltraSAN*

UltraSAN is a software environment for modeling computer systems and networks. Models are specified using Stochastic Activity Network [50], a stochastic variant of Petri nets. It supports a number of analysis and simulation-based solution techniques [51].

3.5.5 *Doodle*

Doodle is a declarative rule-based language for querying database objects visually. One aspect of this language is of interest to this project: its ability to graphically display database objects in a number of layouts. Cruz [52] discuss how to write visual programs for constructing

planar drawings of trees [53], series-parallel diagraphs [54], and acyclic diagraphs [55] using doodle. These ideas may be useful is constructing layout algorithms for a Petri net model.

CHAPTER FOUR

CONTRIBUTION OF THIS WORK

4.1 Starting point for the project

The original CGE developed by Gravelle [4] was a proof of concept demonstration that a graphical interface, rather than a text editor, could be utilized to generate CSPL coding language constructs [2]. An overview of these constructs is presented in Section 6.5. The primary goal for this project was to extend the original CGE in the following areas. The following sections expand on each of these points.

Improvements in reading and saving a file:

- Added the ability to read in a CSPL file and display its contents graphically. The CSPL parser is designed around SPNP version 6 [2] and is backward compatible to version 4 [55].
- Added the ability to display and edit a CSPL file that has been previously read.
- Improved the ability to save a Petri net model in CSPL format. This format is compatible with SPNP version 6.
- Added the ability to specify graphically the parameters associated with the *options* function. These parameters specify how SPNP obtains numerical results from a Petri net model.
- Added the ability to read and save a model in a CGE format (an application-specific format).

Improvements in the design:

- Developed an easy to use interface for developing a graph layout algorithm. This capability provides a means of organizing the elements of a Petri net model in an understandable way.
- Improved the maintainability of the design and implementation of the application.

Improvements in the user interface:

- Added the ability to specify a variable firing rate or probability.
- Improved the interface for defining a guard function.
- Added extensive consistency checking to assist the user in creating a syntactically correct Petri net model. For example, attempts to connect two places with an arc generates an error message.

4.2 Improvements in reading and saving a file

4.2.1 Background

A discussion of the challenges and rewards in developing a standard simulation environment is presented by Wagner [56]. Such an environment would combine various aspects of the simulation process into one complete powerful tool. The environment should provide the necessary state-of-the-art advances and concepts that a modeler may require during the simulation development and execution process. The “environment” requirements must address the end user needs while minimizing the necessity of learning a new language, and additionally specify the useful features, independent of different platforms and implementation languages. Hence the opportunity exists for standardization among existing tools (or environments).

The concept of an ideal Petri net modeling environment combines different aspects of the modeling process and analysis techniques into one powerful tool. This project explores a number of these ideas. The application is designed using standard object-oriented methodology

and is implemented in Java. Perhaps more importantly, the project addresses the issue of compatibility at the level of the modeling language.

4.2.2 File Format Compatibility

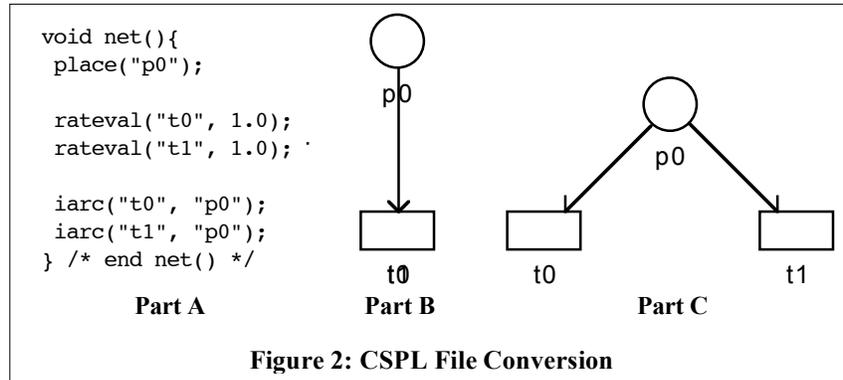
There are a number of tools on a variety of systems for developing a Petri net model. Although these environments provide similar features, they rarely use the same file format to represent a Petri net model. A possible source of frustration for a modeler is to utilize a large model developed by another group using an unfamiliar or different tool. Obstacles such as learning a new paradigm, incompatible models, and incompatible data formats prevent the exploitation of previous work. One example is an analyst who has developed a model for his/her own use. Later, another organization (or group within the organization) discovers the model and wants to use it on another system. Many times, consultants have models, which could be used by clients, if the model could execute on the client's system. In universities, students would like to develop or use a model from a class on their home-based systems [57].

A number of ideas has been presented as to possible avenues that can be taken to ease model exchange, re-use of models between tools, and other amenable connotations of moving towards a more standard modeling environment [56]. The goal of creating a tool-independent modeling language may indirectly assist in combining the best features of the available modeling tools. One challenge in creating such a modeling language is defining a standard set of language constructs.

The eCGE application has taken a step in the direction of file format compatibility with its support of the CSPL (C-based Stochastic Petri-net Language) file format. This format is the specification language for the Stochastic Petri Net Package (SPNP) [2], a tool for modeling and analyzing complex system behaviors (such as system reliability and availability analysis).

Currently, the main drawback of this format is that no graphical information is encoded for the model's elements. The eCGE application defined its own file format, which encodes positional data for each element as well as the attributes contained in CSPL. For example, a place saves the name, number of tokens, and its position on the drawing window.

4.2.3 CSPL Conversion Example



Importing a CSPL file into eCGE is done in two steps. The first step reads the CSPL file to determine the elements and attributes of the model. Since the CSPL format does not include graphical information for its elements, eCGE assigns graphical information to each element to display the model in a window. This step is performed with a graph layout algorithm. The Tree Algorithm is currently used to assign a position to each element.

A simple example of the conversion process is shown in Figure 2. Part A shows the CSPL input file. Part B shows the model after the CSPL file has been read. Part C shows the layout after the Tree Algorithm has assigned each element a position.

4.3 Basis for the enhanced CGE

Besides the original CGE, this thesis draws from two pre-existing components: the graph layout algorithms developed by Wen Wei, and the CSPL parser written by the author. These components served as the foundation for accomplishing the objectives described in section 4.1.

The challenge for this thesis was combining these components into a logical, consistent whole. The framework for the enhanced CGE, along with the framework for each of its underlying components, is shown in Table 1.

Component	Language	GUI Framework	Design Methodology	OS Supported
Original CGE	C++	MS Visual Studios	procedural	MS Windows
CSPL parser	C++	N/A – command line application	object-oriented	any
Spring and Tree Algorithms	C++	N/A – selected by a menu item	procedural	any
Enhanced CGE	Java	AWT / Swing	object-oriented	any supporting Java 1.4

Table 1: Components of the enhanced CGE

The enhanced CGE improves upon the original design through the application of object-oriented analysis and design techniques. The design of the original CGE was procedural-based, in which the program is viewed as a collection of objects that interact. This architecture is relatively “flat,” rather than hierarchical. A significant percentage of the source code is grouped into a few large files, which makes maintenance and enhancement difficult. The approach taken to remedy this situation is described in the next section. Essentially, this process translated the procedural-based design of the original into an object-oriented framework. The resulting design provided a more modular solution than the original. A side effect of the improvements in the design was the increased readability of Wen Wei’s graph layout algorithms. (A more detailed analysis of these algorithms is given in Appendix A.)

Although the majority of the improvements to the application were at the design level, some work was done to improve the user interface. The goal at the interface level was to simplify the process of specifying and modifying a Petri net model. The most significant improvements at

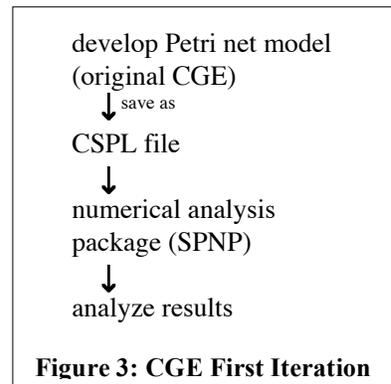
this level are to the design of many dialog boxes. The ability to open and save a model was also added.

Implementing the ability to save and read a file in eCGE format was straightforward, but adding the ability to read in a CSPL file required implementing a parser. This one-pass parser was based on standard compiler design techniques [55]. Accomplishing these goals required a re-design of both the original CGE and the Spring and Tree Algorithms.

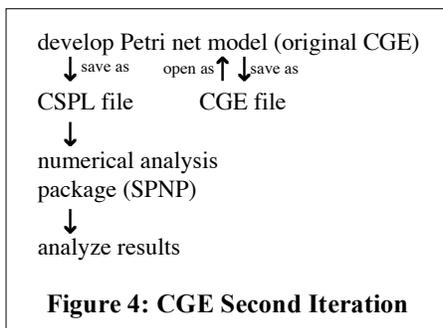
The approach taken to design and implement the enhanced application is one of the more important differences between this project and the original application. Object-oriented methodology was used in the design process. The Java language provided an ideal framework for implementing this project.

4.4 Past and Present Contributions to CGE

- The first iteration of CGE, shown in Figure 3, had the ability to graphically model a Petri net, and to save this



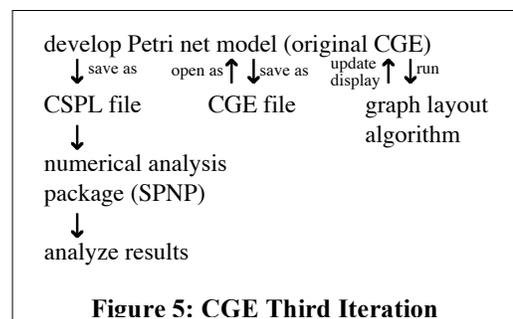
model as a CSPL file.



- The second iteration of CGE, shown in Figure 4, was a class project at WSU. It added, among other things, the ability to save and open a model in CGE format. This allowed the ability to modify a Petri net model without

having to completely re-draw the model.

- The third iteration of CGE, shown in Figure 5, was developed when Wen Wei, in her thesis project, implemented two graph layout



algorithms (the Spring and Tree Algorithms) to improve the aesthetics of a Petri net model.

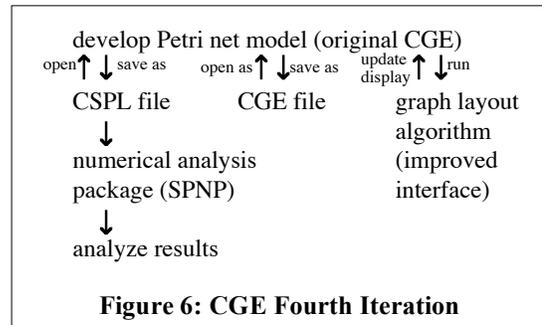
- The fourth iteration of CGE, shown in Figure 6, is the current version. One of the features it adds is the ability to read in a CSPL file. Combining the CSPL parser and the previous version of CGE required a significant amount of work at the design level.

4.5 Design of the enhanced CGE

The result of the translation process is a design that is object-oriented, rather than procedural, and it divides the application into a relatively large number of classes, whereby each class implements a single

part of the overall application. This effort involved decomposing each component of the application into classes, and defining the interaction between these classes. Conceptually, these classes are organized hierarchically into a number of levels. This is shown in section B.2 of Appendix B. The design decomposed the application into a collection of cooperating objects. The remainder of this section describes each level, moving from a high to low level of abstraction, which presents the design as a top-down, modular structure.

One of the basic problems with the original application was that the design was difficult to understand and modify. The design of the original application was procedural-based in which the program is viewed as a collection of functions that interact. The architecture is relatively “flat,” with few levels of abstraction, and as evident in the implementation, a significant percentage of the source code is grouped into a few source files. The current version of the application improves upon the original design. These improvements are described in section 4.5.1.



4.5.1 Design Approach

An original contribution of this work is related to the re-design and integration into a single program with a user-friendly interface. The translation was done using an iterative approach as shown in Figure 7; each step is described in detail below. This approach provided a way to explore the possible alternatives before committing to a solution.

- The translation process started by understanding the procedural based design of the original CGE application. This process was made more manageable by first partitioning the application into its constituent components. The application could then be translated one component at a time.

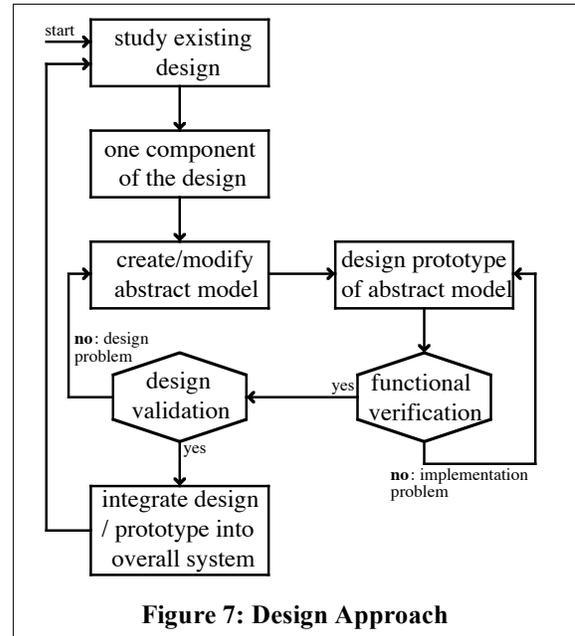


Figure 7: Design Approach

- One of these components in the design was then selected for further study. This step involved determining the function and current implementation of the selected component.
- The functionality of the original component was duplicated in the enhanced CGE. Exploring a possible design often involved creating a prototype to determine and correct potential flaws.
- This prototype was used to help determine whether the design represents a valid solution. At times, the construction of a prototype was the only means through which requirements can be effectively derived.
- The research done in the previous step was incorporated into the application as a whole.

- These steps were repeated until the entire design was translated.

The focus of the iterative process was to translate the application from a procedural framework (written in Visual Studio) into an object-oriented framework (written in Java). The first iteration produced a skeletal application with minimal functionality. Successive iterations developed the application one component at a time. Using an iterative approach allowed insight into each component of the project. The process of studying one aspect of the design often suggested a direction for the subsequent iteration: it identified new issues that required additional research, and provided answers or justifications for particular design decisions.

The use of prototyping throughout the design process proved helpful in managing the tension between qualitative details and design abstractions. To identify, separate, and understand the reasoning behind an aspect of the original design, its functionality was reversed engineered. This process often involved researching the theory behind a component's function. Prototyping was done to explore ideas on how to recreate this functionality in the enhanced application. Going back and forth between a prototype and the underlying theory ensured that the design principles were well grounded and that the design details were organized in a conceptually useful and accessible way.

4.5.2 Levels of Abstraction

The design process involved work within two domains, the technology and use domains: The *technology domain* consists of the factors that influence the architecture of the system, its functionality, and the interaction techniques. The *use domain* consists of design guidelines, overall work styles, and the individual patterns of use. Addressing the tension between these domains has been the cornerstone of Human-Computer Interaction research for the past two decades.

The goal of the design process is to manage the complexity of the application. The design is divided into layers to separate the parts that work in computer-science terms from the part that works in problem-domain terms. The top level of the design describes the problem that is being solved. The actual implementation is handled at a lower level.

Most of the effort for this project went into the first domain, mainly working out the details of the architecture and conceptual model of the updated tool. Developing the architecture involved decomposing each component of the application into objects, and defining the interactions between these objects. The result of the design process divides the application code into a relatively large number of classes, whereby each class implements a single part of the overall application. A set of class charts, shown in Appendix B, provides an overview of how each class relates to the application as a whole.

4.5.2.1 User Interface Level

Work at this level is very abstract; it consists of determining which instruments best mediate the interaction between a user and the objects in the interface. Many of the constructs at this level are borrowed from (and possibly improve upon) the original application, such as:

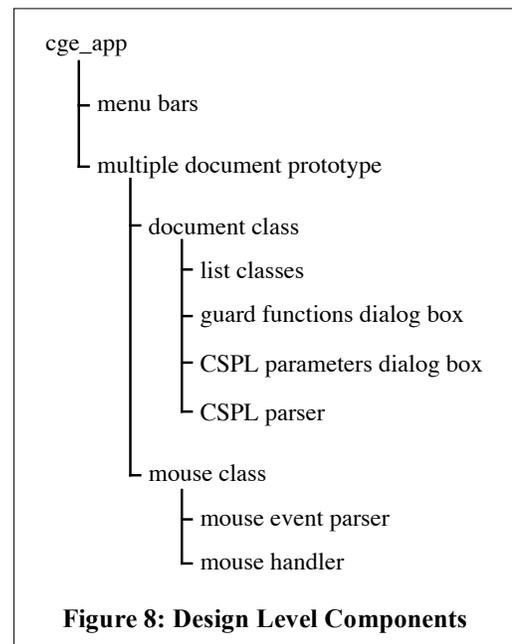
- Interface for the graph layout algorithms
- How each Petri net construct is represented graphically: places are drawn as circles and transitions as boxes.
- User-level layout of each dialog box

4.5.2.2 Design Level Components

Work at the design level is a multi-step process that focuses on a number of the program's attributes: software architecture, interface representations, and data structures. The design level

components into which the program is divided are shown in Figure 8 and the listed below. The class charts in Appendix B show the design of the application in more detail.

- Menu bars contain user-selectable commands.
- The multiple document prototype coordinates the activities of a document at a high level.
- The dialog boxes allow the user to define the attributes of a Petri net element. Each dialog box is implemented by a set of classes.
- The CSPL parser is used to input a CSPL file into eCGE.
- The list classes store the elements of a Petri net model. Three lists are maintained for a model: one list for the set of places, one for transitions, and one for arcs.
- The mouse handler coordinates how each semantic mouse event is processed by the application.



4.5.2.3 Algorithmic Level

Work at this level focuses on translating the functional abstraction specified by the design into a working implementation. At this level, decomposing the design-level abstraction into an implementation is based on data design and procedural design approaches discussed for conventional software engineering. Many of the individual cases involve a simple computational or procedural sequence that is implemented in an individual class or function. Examples of elements at this level include:

- Event handling code (parsing individual mouse events to form semantic events, parsing key events)
- Implementation of the list classes (AVL tree / linked list)
- File format (eCGE / CSPL)

4.5.2.4 Java Virtual Machine

The Java Virtual Machine is at the lowest level of abstraction: it handles the interaction between the environment and the application. The use of Java to implement the application permits the source code to be modified and recompiled on a number of different platforms. The set of libraries included in the language (such as Swing and file I/O) provide an industrial-strength user interface framework for implementing the application in an architecturally neutral manner. The most important requirement to running the eCGE on a target system is the presence of the Java run time system. Research into these areas of the Java Virtual Machine specification was required in the implementation of eCGE:

- Graphical library functions: the Swing and AWT libraries were used extensively in implementing the user interface components.
- Event listeners: the virtual machine monitors the environment for events such as keystrokes or mouse clicks. The virtual machine then reports these events to a higher-level function.
- File Input/Output: the virtual machine handles the low-level details of how to read and write a file on the native file system.

CHAPTER FIVE

eCGE TOOL OVERVIEW AND IMPLEMENTATION DETAILS

5.1 Introduction

The eCGE tool is written in Java. The user interface uses the conventions found in other applications, such as mouse gestures, accelerator keys, placement of menus, and icons and toolbar glyphs. A window can be moved, modified in size or shape, and closed into icon form using the mouse as the primary input device. All drawing, menu selection, and analysis activities are initiated by mouse key clicks. The keyboard is used mainly in situations when arbitrary text input is needed.

5.2 User Interface Level

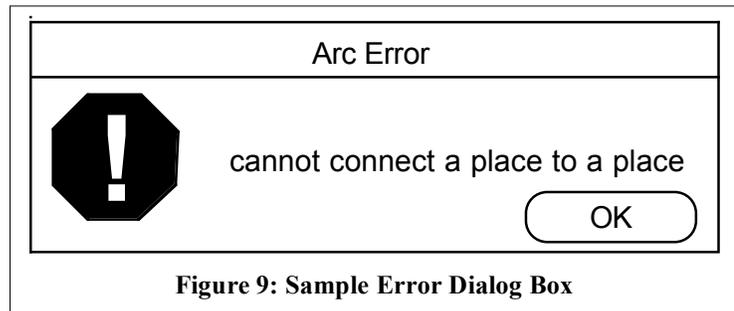
The basic philosophy of the user interface is simple; it should be flexible enough to permit design creativity while removing any tedium from the activity. This is accomplished by adopting the following principles:

- The user interface provides very rapid, easily selectable options.
- The tool does anything that is repetitious or algorithmic.
- Any operation can be undone. This capability is implemented in the dialog boxes, but not for operations done on the drawing window.
- Options should be split into logical categories with different access for each category.
- A graphical representation of options is preferable to a textual representation.

Because a rigid syntax is critical to correct language interpretation, any attempt to compromise it (such as connecting a transition to a transition) results in an error. Since computer programs cannot exactly recognize the command “How many tokens are in place p ?” A formal,

concise, and rigid set of syntactic rules is essential for precise communication. The eCGE is designed to enforce these rules. The user is shown an error message when an illegal action is taken, and the action is not allowed to occur. Figure 9 shows an example of an error dialog box. Enumerated below are some of the rules:

- Transitions can only be connected to places, and vis-versa.
- Deleting a place or transition deletes all of the arcs associated with that node.



- Tokens can only be associated with places.
- Only transitions can be annotated with firing rates [59].

Incorporated in the user interface design is the principle of polymorphism, which allows similar operations to be applied to a variety of objects. For example, double-clicking on an element in the drawing window (such as an arc, transition, or place) opens a dialog box showing the properties of the element. The design of the user interface follows three principles:

- Each Petri net construct and user interface construct is represented with meaningful visual metaphors.
- Physical metaphors (such as inserting an element with the mouse) are used rather than a complex syntax (i.e., CSPL format).
- Rapid incremental, reversible operations are used on the object of interest. The result of the operation should be immediately visible.

5.2.1 Interaction Techniques

User interaction with the eCGE application uses the Windows, Icons, Menus, and Pointing (WIMP) model [49].

- A Petri net model is developed in the main window. A dialog box is opened by double-clicking on an element in the drawing window, or (in some cases) by selecting a menu item. A dialog box, namely places, transitions, arcs, guard functions, and CSPL parameters, can access a number of Petri net constructs.
- The menu bars define the operations for the applications. The eCGE adds the ability to save and open a model.
- The mouse is used frequently for operations such as clicking or dragging elements in the drawing window, selecting or moving a Petri net model window, or selecting an item in the menu bar.
- The keyboard is used only to input and edit text. For convenience, some commands on the menu bars also have a keyboard shortcut.

5.2.2 Drawing Window

The drawing window is used to create and edit a model. Making up each window are three basic components, which are more fully described in the following sub-sections.

5.2.2.1 Toolbar

The toolbar is the following collection of icons representing drawing and editing features. The highlighted item also appears in the drawing window's message box.

- Select the cursor
- Create a place
- Create a transition

- Create an arc

5.2.2.2 Drawing Canvas

A Petri net model is developed on the drawing canvas, which performs the following functions:

- Move an element of the Petri net model
- Edit a place
- Edit a transition (specify firing rate, etc.)
- Edit an arc
- Delete a place
- Delete a transition
- Delete an arc (Deleting a place or transition also removes all the arcs associated with the element.)

5.3 Design Level

Work at this level deals with the design issues of the application. The design of the application is based on the model-view-controller pattern in which the look and feel of a component is implemented with one object and the data associated with the component is stored in another object. The use of Java shaped the design of the application. One example is the mouse handler, which is designed around the event-handling mechanism. Most of the work on this project focused on this level and the implementation.

The goal at this level was specify the architecture of the system, as well as object interaction. The design of the enhanced application is object-oriented, rather than procedural (as in the original). The application of object-oriented analysis and design techniques significantly improved the design of the application while maintaining the good aspects of the original. In a

number of cases, translating the individual pieces of the application (i.e., dialog boxes) was relatively straightforward. In a few cases, it was as simple as figuring out how to translate the C++ structures to Java.

Specifying the architecture involved decomposing each user-level construct (such as a dialog box) into a set of objects, where an object performs a specific task. Each class defines an interface through which it communicates with other classes. A major challenge was to minimize the amount of coupling and maximize object cohesion in the design.

Design documentation was created for each component. The level of detail varies from component to component. The dialog boxes are documented at a high level of abstraction; the mouse handler at the algorithmic level, and AVL tree at the implementation level.

Code reuse typically occurs at the bottom levels of a system design hierarchy while design reuse results in whole “branches” of the tree being reused. The design of the guard function dialog box is an example of design reuse. The *function_dialog* class provides a template for handling guard functions. This template can be extended (namely, by classes *guard_dialog*, *arc_function_dialog*, and *trans_function_dialog*) by inheriting its specifications and incorporating additional features.

5.4 Algorithmic Level

Work at this level deals with the algorithms that implement each design-level construct. The approach taken at this level differs little from the data design and procedural design approaches discussed for conventional software engineering. In many cases, the algorithm is a simple computational or procedural sequence that is implemented in an individual class. The remainder of this section describes the more complex of the algorithms used in the project.

5.4.1 *AVL Tree*

An AVL tree is a type of binary tree that is always ‘partially’ balanced. Each node in the tree has a left and right sub-tree. The criteria that is used to determine balance of a tree is the difference between the heights of the sub-trees of any node in the tree. The running time of the lookup, insertion, and deletion operations is logarithmic for both the average and worst cases. The details of these algorithms are discussed in detail in Appendix B.4. The efficiency of the list classes improves the running time of the CSPL parser and the graph layout algorithms, since these components rely heavily on the lookup operations.

5.4.2 *CSPL Parser*

The mechanism for importing a CSPL file into eCGE is a one-pass parser [58]. This parser is designed to import the following CSPL functions into eCGE: **net** (containing the Petri net elements), **options** (CSPL parameters), **assert**, **ac_init**, **ac_reach**, and **ac_final** [2]. Any other functions in the input file are ignored. One way to improve this parser is to add the ability to import guard functions into eCGE.

Conceptually, this parser is divided into lexical and syntactic analyses. The lexical analyzer filters the CSPL file into tokens, eliminating any unnecessary information, such as white space and comments, in the process. The syntax analyzer uses these tokens to determine the structure of the underlying Petri net model. Error detection is done at the lexical and syntactic levels. Any errors are recorded in the parser’s log files.

5.4.3 *Mouse Handler*

The mouse handler coordinates many of the activities in the eCGE application. Conceptually the design of the mouse handler is divided into two layers to separate the responsibilities of receiving user input and responding to this input. The *mouse event parser*, defined in

mouse_class.java, translates the mouse-related events generated by the Java Virtual Machine into one of three possible cases. The *mouse handler* (mouse_handler.java) coordinates the program's response to each of these cases.

5.4.3.1 Mouse Event Parser

The low-level mouse event functions (namely mousePressed, mouseReleased, and mouseDragged) confer a lot of information about the mouse. The mouse event parser uses this information to form one of the following semantic events:

- A *single click* is used to select an element (mouse_single.java)
- A *double click* is used to edit the properties of an element (mouse_double.java)
- A place or transition may be *moved* (mouse_dragged.java). The position of any arcs adjacent to a moved place or transition is automatically updated.

The mouse event classes as defined by the Java library include functions to detect these three cases. Unfortunately, these methods cannot be used directly for the eCGE mouse handler. The problem is that the built-in functions assume that the actions in each of the three cases are mutually exclusive. This assumption does not hold for the eCGE application. For example, during a double-click the mouse may move slightly between the two successive mouse clicks. If the library functions are used directly, this case is interpreted as two semantic events (double click and mouse movement) rather than one (double click).

5.4.3.2 Mouse Handler

The mouse handler receives semantic events from the mouse event parser and initiates a response. This response can be divided into three cases: opening a dialog box (double-click), selecting an element (single-click), or moving an element. The mouse handler delegates the

actual implementation for each response to other components of the project. For example, moving a place or transition is implemented in the list classes.

5.4.4 Multiple Document Prototype

The multiple document prototype explores the possibility of having more than one Petri net model open at once. This prototype is skeletal at this point. The implementation is based on Java's internal frames.

5.4.5 Arc Line Calculations

The basic approach to modeling an arc is almost identical to that of the original application. An arc is modeled as a single line segment. The calculations for this line are based on elementary trigonometry. At an abstract level, these calculations perform the following functions:

- Calculate the two end points of the arc
- Calculate the position of the arrowhead (for an ordinary arc) or circle (for an inhibitor arc).

5.5 Case Study 1: use of a pre-existing model

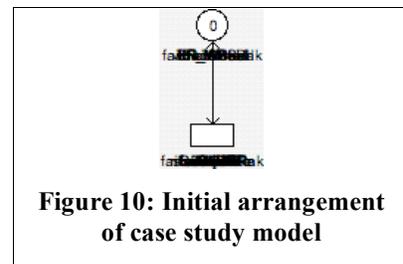
This case study is based on a safety and reliability analysis using stochastic Petri nets [60]. The purpose of this analysis was to determine the operational fitness of an embedded computer system in a vehicle in terms of reliability and availability. Since the actual system is complex, performing the analysis consisted of decomposing the overall system into a set of Petri net models. Each model analyzes a specific aspect of the overall system. Three scenarios were considered in the analysis: skidding, slipping, and steering. An embedded computer system controlled each of these systems.

- The *skidding* model represents the anti-lock braking system
- The *slipping* model considers the factors involved during the vehicle’s acceleration.
- The *steering* model represents the system that detects and counteracts oversteer and understeer.

For simplicity, this case study focuses on one of these Petri net models: the “connected cyclic reliability” (CCR) model for the anti-lock braking system. This model predicts the mean time to failure for the system, given the failure rates for each component. The main components for this model are the speed sensors, braking system components, and the electronic brake control module computer.

5.5.1 Importing the model into eCGE

The CCR model was originally developed using the CSPL language. Figures 10 – 12 show three possible arrangements of this model in eCGE. Figure 10 shows the initial arrangement of



the elements in a drawing window. Figure 11 shows a possible arrangement of the model after applying the Random graph layout algorithm (described in Section A.2.3). The Spring (A.2.1) and Tree (A.2.2) Algorithms were not designed to handle a model this complex: these give results similar to the Random Algorithm in this case. Figure 12 shows the model after it has been layed out manually.

One existing tool for graph layout is *graphviz*. This tool focuses on two types of graph layout algorithms: hierarchical layouts of trees and DAGS (directed acyclic graphs), and virtual physical (“spring model”) layouts of undirected graphs. The theory behind this tool is described in [61].

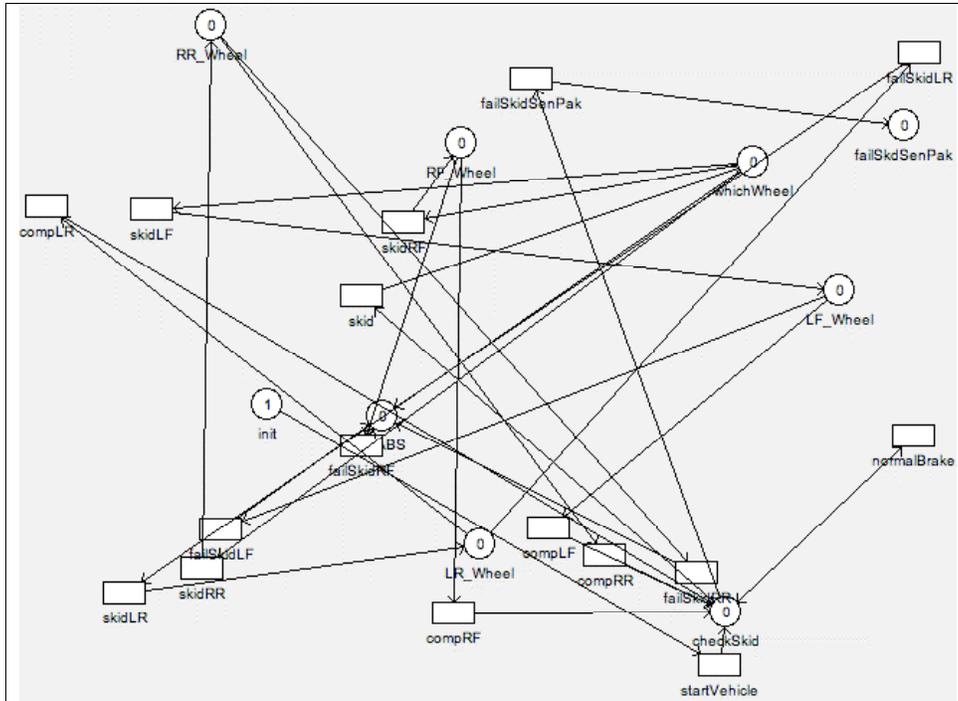


Figure 11: Random arrangement of case study model

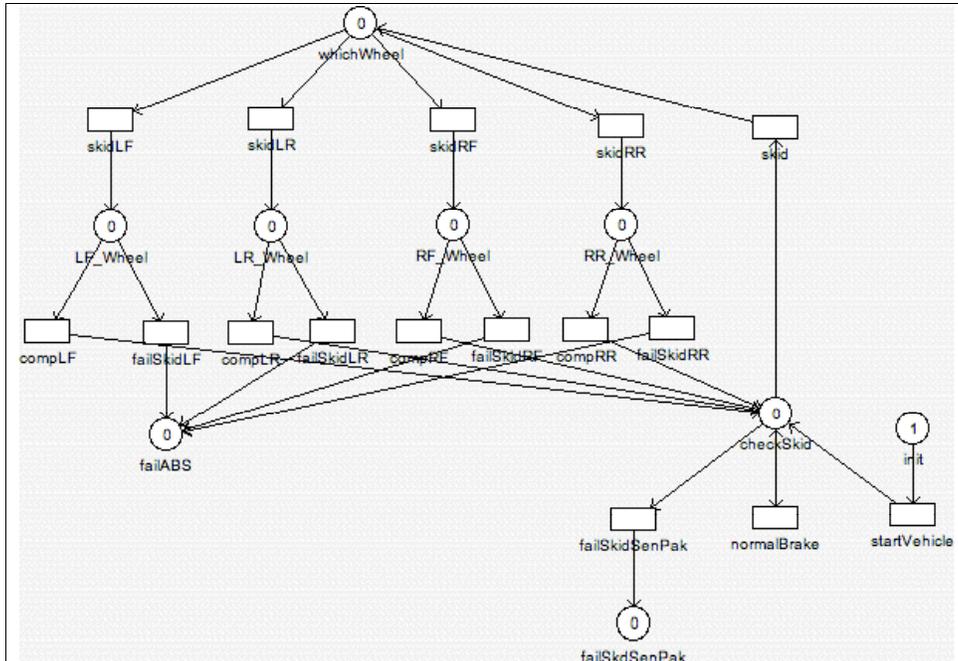


Figure 12: Organized layout of case study model

5.5.2 Description of the Model

The initial marking of the CCR model consists of one token in the *init* place. The *failSkidSenPak*, *failSkidLF*, *failSkidLR*, *failSkidRF*, and *failSkidRR* transitions represent the probability (or rate) of component failure. These probabilities are represented in the CSPL syntax `rateval,(transition, rate)`; The other transitions represent the operational transitions.

The operational transitions fire alternately indicating that a particular sub-component has functioned. This allows for the model to include the different operational dependencies that are present in the actual system.

The firing rates assigned to the various operational transitions and failure transitions represent a competition among the operational components and the possibility of a failure occurring. As long as the token continues to recycle among the operational places, the system will continue to run. If a failure transition fires, it will consume the token and the system will have failed.

Depending on the failure rates assigned to the failure transitions, the token will eventually enter a fail transition, representing system failure. One example is the competition between transitions *compLF* and *failSkidLF* when place *LF_Wheel* is enabled. Under normal operation, *compLF* fires. Component failure is represented by the firing of *failSkidLF*.

5.5.3 Steps used to develop the model

Originally the CCR model, shown in Figure 11, was developed in the CSPL language using a text editor. The eCGE tool improves the situation by replacing the text editor with a graphical editor. The stochastic analysis can be performed with another tool (SPNP) [2]. The input file for SPNP is automatically generated by eCGE when a model is saved.

For backward compatibility, it is possible to import the textual version of the CCR model (shown in Figure 11) into eCGE. The CSPL constructs that are recognized by eCGE are described in section 6.5. For example, the line `rateval("startVehicle", 1.0);` creates transition `startVehicle` with rate 1.0. In eCGE, the transition is shown in the drawing window and the rate (along with `startVehicle`'s other attributes) can be edited in a dialog box. This dialog box is described in Figure 40 in Appendix C.

Figure 10 shows the drawing window after the CSPL file has been imported and the elements have been manually layed out. A future version of eCGE would lay out the elements automatically.

```
#include "user.h"

void options()
{ /* options */
  iopt(IOP_ITERATIONS, 2000);
  fopt(FOP_PRECISION, 1.0E-6);
  fopt(FOP_ABS_RET_M0, 0.0);
  iopt(IOP_CUMULATIVE, VAL_YES);
  iopt(IOP_SENSITIVITY, VAL_YES);
  iopt(IOP_DEBUG, VAL_NO);
  iopt(IOP_OK_ABSMARK, VAL_YES);
  iopt(IOP_OK_VANLOOP, VAL_NO);
  iopt(IOP_OK_VAN_M0, VAL_YES);
  iopt(IOP_OK_TRANS_M0, VAL_YES);
  iopt(IOP_ELIMINATION, VAL_REDONTHEFLY);
  iopt(IOP_MC, VAL_CTMC);
  iopt(IOP_SS_METHOD, VAL_SSSOR);
  iopt(IOP_TS_METHOD, VAL_FOXUNIF);
  iopt(IOP_PR_MARK_ORDER, VAL_CANONIC);
  iopt(IOP_PR_RSET, VAL_YES);
  iopt(IOP_PR_RGRAPH, VAL_YES);
  iopt(IOP_PR_MERG_MARK, VAL_YES);
  iopt(IOP_PR_FULL_MARK, VAL_NO);
  iopt(IOP_USERNAME, VAL_YES);
  iopt(IOP_PR_DERMC, VAL_NO);
  iopt(IOP_PR_MC, VAL_YES);
  iopt(IOP_PR_MC_ORDER, VAL_FROMTO);
  iopt(IOP_PR_PROB, VAL_YES);
  iopt(IOP_PR_PROBDTMC, VAL_NO);
  iopt(IOP_PR_DOT, VAL_NO);
  iopt(IOP_SIMULATION, VAL_NO);
  iopt(IOP_SIM_CUMULATIVE, VAL_YES);
  iopt(IOP_SIM_RUNS, 0);
  fopt(FOP_SIM_LENGTH, 0.0);

  fopt(FOP_SIM_ERROR, 0.0);
  fopt(FOP_SIM_CONFIDENCE, 0.95);
} /* options */

/* Stochastic Petri Net Specification */
void net(){
  /* places */
  place("init");
  init("init", 1);
  place("checkSkid");
  place("whichWheel");
  place("LF_Wheel");
  place("RF_Wheel");
  place("LR_Wheel");
  place("RR_Wheel");
  place("failABS");
  place("failSkdSenPak");

  /* transitions */
  rateval("startVehicle", 1.0);
  rateval("normalBrake", 1.0);
  rateval("skid", 1.0);
  rateval("skidLF", 1.0);
  rateval("skidRF", 1.0);
  rateval("skidLR", 1.0);
  rateval("skidRR", 1.0);
  rateval("compLF", 1.0);
  rateval("compRF", 1.0);
  rateval("compLR", 1.0);
  rateval("compRR", 1.0);
  rateval("failSkidLF", 1.0);
  rateval("failSkidRF", 1.0);
  rateval("failSkidLR", 1.0);
}
```

```

rateval("failSkidRR", 1.0);
rateval("failSkidSenPak", 1.0);

/* arcs */
iarc("startVehicle", "init");
oarc("startVehicle", "checkSkid");
iarc("normalBrake", "checkSkid");
oarc("normalBrake", "checkSkid");
iarc("skid", "checkSkid");
oarc("skid", "whichWheel");
iarc("skidLF", "whichWheel");
oarc("skidLF", "LF_Wheel");
iarc("skidRF", "whichWheel");
oarc("skidRF", "RF_Wheel");
iarc("skidLR", "whichWheel");
oarc("skidLR", "LR_Wheel");
iarc("skidRR", "whichWheel");
oarc("skidRR", "RR_Wheel");
iarc("compLF", "LF_Wheel");
oarc("compLF", "checkSkid");
iarc("compRF", "RF_Wheel");
oarc("compRF", "checkSkid");
iarc("compLR", "LR_Wheel");
oarc("compLR", "checkSkid");
iarc("compRR", "RR_Wheel");
oarc("compRR", "checkSkid");
iarc("failSkidLF", "LF_Wheel");
oarc("failSkidLF", "failABS");
iarc("failSkidRF", "RF_Wheel");
oarc("failSkidRF", "failABS");
iarc("failSkidLR", "LR_Wheel");
oarc("failSkidLR", "failABS");
iarc("failSkidRR", "RR_Wheel");
oarc("failSkidRR", "failABS");
iarc("failSkidSenPak", "checkSkid");
oarc("failSkidSenPak", "failSkdSenPak");
} /* end net() */

void assert(void)
{ /* assert */
return(RES_NOERR);
} /* assert */

void ac_init(void)
{ /* ac_init */
fprintf(stderr, "\nSkidding Sub-Model");
    fprintf(stderr, "    Generating SRN data
... \n\n");
    pr_net_info();
} /* ac_init */

void ac_reach(void)
{ /* ac_reach */
fprintf(stderr, "\nThe reachability graph
is being generated ... \n\n");
    pr_rg_info();
} /* ac_reach */

void ac_final(void)
{ /* ac_final */
int i;
/*
    for ( i = 1; i < 10; i++ )
    {
        printf("\n i = %d\n", i);
        time_value( (double) i );
        printf("\n i = %d\n", i);
        pr_expected("mark(ABSF)", rfunc_p);
    }
*/
} /* ac_final */

```

Figure 13: Case study model in CSPL format

5.6 Case Study 2: layout of a model

The main purpose of the layout of a model is to show the logical structure of a model. It is possible to model logical constructs, such as a loop, if-then-else, or synchronization, with a Petri net [74]. A good layout can also help in detecting potential problems in a model. This case study focuses on detecting two types of problems: conflict and confusion.

5.6.1 Conflict in a Petri net

Figure 14 shows an example of conflict in a Petri net. Transitions t_1 and t_2 are both enabled by the presence of the token in p_1 . If t_1 fires then t_2 is no longer enabled. Conversely, if t_2 fires, t_1 is disabled. Figure 14 shows a number of examples of conflict, namely between the transition representing a functional component and the transition representing component failure.

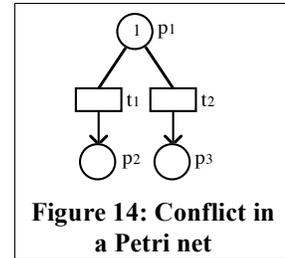


Figure 14: Conflict in a Petri net

5.6.2 Confusion in a Petri net

While conflict is a local phenomenon in the sense that only the firing conditions of the transitions with common input places are involved, confusion involves firing sequences. Confusion occurs when one firing sequence leads to conflict while another one does not. Figure 15 shows an example of confusion [63]. Transitions t_1 and t_3 are enabled concurrently. At this point, there is no conflict. If transition t_3 fires first, then transition t_2 fires and execution terminates.

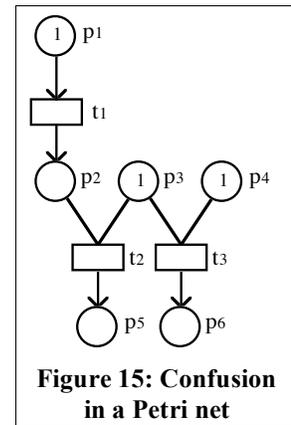


Figure 15: Confusion in a Petri net

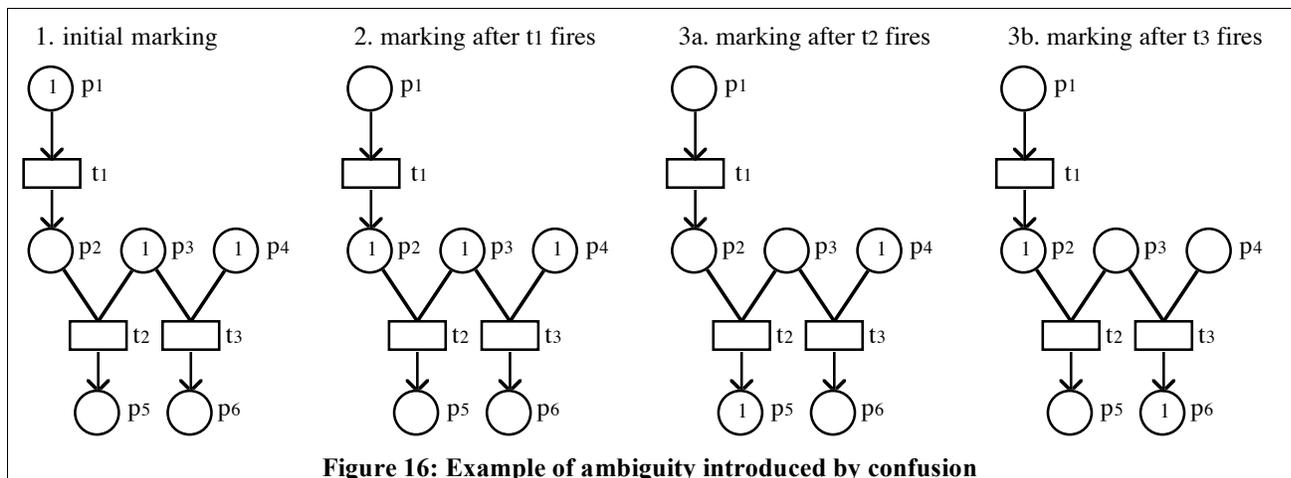


Figure 16: Example of ambiguity introduced by confusion

But if transition t_1 fires first (case 2), there is a conflict between transitions t_2 and t_3 . There is an ambiguity as to which transition will fire. The sequence of markings which can occur in the

situation is shown graphically in Figure 16. If t_2 fires (case 3a), a token is placed in p_5 and a token is removed from p_2 and p_4 . If t_3 fires (case 3b), a token is placed in p_6 and a token is removed from p_3 and p_4 .

It is sometimes possible to resolve confusion. There are three possible mechanisms for resolving the confusion in this example. The idea is to cause t_3 to fire before t_1 .

- Transition t_1 could be assigned a lower priority than t_2 and t_3 .
- A guard function could be defined for t_2 disabling it whenever t_3 is enabled.
- An inhibitor arc could be defined between p_4 and t_1 .

5.6.3 Detecting the structure of the model through the layout

A number of researchers [64]-[67] have studied what is described in the literature as a dual-task situation. As it applies to this case study, the user's attention will be focused on the primary task of analyzing the structural relations of a model, but at times it may be necessary to divert partial attention to a secondary task that involves understanding the syntax or organization of the model. The findings imply that the introduction of a secondary task negatively impacts the performance on a primary task.

Currently, one of the more effective methods of detecting confusion is by means of analyzing structural relations. The graphical editor of eCGE provides a means of organizing a Petri net model. Detecting potential concerns such as conflict and confusion is easier with an organized layout than with a random layout. A graph layout algorithm can assist in organizing a model.

A poor graphical layout is one example of a dual-task situation. Another is the modeling language itself: a graphical representation of a Petri net generally allows for quicker recognition of specific information than the equivalent textual representation [68]. The structure of the

underlying Petri net tends to be hidden by CSPL's textual constructs. For example, the CSPL file in Figure 17 is equivalent to the Petri net shown in Figure 15.

```
place("p1"); init("p1",1);
place("p2");
place("p3"); init("p3",1);
place("p4"); init("p4",1);
place("p5");
place("p6");

rateval("t1",1.0);
rateval("t2",1.0);
rateval("t3",1.0);

iarc("p1","t1");
iarc("p2","t2");
iarc("p3","t2");
iarc("p3","t2");
iarc("p4","t3");

oarc("p2","t2");
oarc("p5","t2");
oarc("p6","t3");
```

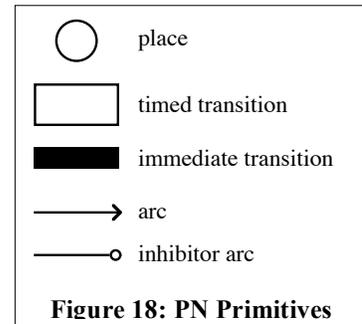
Figure 17: Confusion in a Petri net – CSPL version

CHAPTER SIX

THE PETRI NET LANGUAGE PRIMITIVES

6.1 Elements of a Petri Net

A Petri net is a directed bipartite graph whose nodes are partitioned into two sets, places and transitions. Arcs can only connect a place to a transition (input arcs), or a transition to a place (output arc). Figure 17 shows how each Petri net element is drawn.



6.2 Places

A place represents a constraint on the system; this constraint is represented by the concept of tokens. Tokens are indistinguishable markers that reside in places. Each place contains zero or more tokens. For ordinary Petri nets, the constraint is true if a token is present in the place, and false if the place contains no tokens.

When a transition fires, one token is removed from each of its input places, and one token is added to each of its output places. The movement of tokens in the places defines the dynamic state of a Petri net model. A *marking* defines the configuration of tokens at some point in time. The set of markings is used to obtain numerical results from a model, described in Section 3.5.

6.3 Transitions

The Stochastic Petri net model defines two types of transitions: timed and immediate. The firing time of a timed transition is based on an exponential distribution; the firing time of an immediate transition is zero. Timed transitions are useful in describing the time lapse between

consecutive events. Immediate transitions provide a probabilistic way of describing the selection among different possible events. This distinction provides a useful method of defining complex types of behavior in a single model.

The Petri net formalism provides a mechanism for selectively enabling which transitions may fire in a given marking. A transition is enabled when all of the listed conditions are met. The following sub-section describes each of these conditions in more detail.

- The number of tokens in each input place is at least equal to the multiplicity of the input arc from that place.
- The number of tokens in each input place *with an inhibitor arc* is less than the multiplicity of the input inhibitor arc from that place.
- The enabling function of the transition (if assigned) returns **true**.
- The priority of a transition is a factor in whether it is enabled in a marking.

6.3.1 *Selectively Disabling a Transition*

Additional constructs are used to selectively disable a transition in a marking that would otherwise enable it. Many types of behavior can be specified in a compact way using these constructs.

6.3.1.1 Priority

A priority is associated with each transition. If S is the set of transitions enabled in a marking and if the transition with the highest priority among them is k , then any transition in S with priority lower than transition k will be disabled. To avoid theoretical difficulties, timed and immediate transitions cannot have the same priority [45],[63].

6.3.1.2 Inhibitor Arc

Another way to disable a transition is to define an inhibitor arc from a place to a transition. An inhibitor arc allows a model to test for the absence of tokens in a place. The enabling conditions for an inhibitor arc are the reverse of those of a standard arc: the transition is disabled (rather than enabled) by the presence of tokens in the associated place. Stated more formally, an inhibitor arc from place p to transition t with multiplicity m will disable t in any marking where p contains at least m tokens [2].

6.4 Arcs

A directed arc represents the relationships between constraints (places) and events (transitions). Input arcs and inhibitor arcs connect places to transitions and output arcs connect transitions to places. The firing of a transition is conditioned by the presence of tokens in each of its input places. It is possible to condition this firing by the absence of tokens in an input place: this is represented by an inhibitor arc. An inhibitor arc from a place to a transition has a small circle rather than an arrowhead at the transition.

A multiplicity (positive integer) may be attached to each arc, which is then called a multiple arc. A multiple arc with multiplicity k can be thought of as k arcs having the same source and destination.

The multiplicity of an arc may be defined as a function of the current marking rather than a constant value. This type of function can be used to define behavior that would otherwise require defining a complex set of inhibitor arcs and transition priorities.

6.5 Introducing SPNP's C-based Stochastic Petri net Language

It is possible to import a Petri Net model developed in CSPL. This language provides a textual representation of a Stochastic Petri net model, and allows a large variety of probabilistic

and deterministic system behaviors to be specified. The syntax and the semantics of CSPL are based on the ANSI C language. What distinguishes CSPL from ANSI C is a set of predefined functions available for the definition of Stochastic Petri net elements.

A CSPL file must specify the following functions: **options**, **net**, **assert**, **ac_init**, **ac_reach**, and **ac_final**. Each of these functions is designed to carry out one (or several) specific task(s) by calling some relevant subroutines (functions). The tasks for each of these functions are described in sub-sections 6.5.1 through 6.5.2. The implementation of these functions in eCGE is described in detail in Appendix C.

6.5.1 Specification of “options”

SPNP [2] defines a number of options which allow the user to customize how numerical results are obtained from a Petri net model. The various options are set in the **options** function of a CSPL file. The function for setting an option has two parameters: the *option* and the *value*. The *option* parameter is a constant defined by SPNP. These option parameters are described in detail in Appendix C. The *value* is user-defined. The data type of the *value* is either Boolean, integer, or floating point number. Function **iopt** is used for Boolean and integer data types; **fopt** is used for a floating point parameter.

The eCGE application provides an abstraction by allowing the user to set these options via a dialog box rather than working with these functions directly. The Parameters section of the User’s Manual in Appendix C describes these dialog boxes. When eCGE saves a model in CSPL format, the settings in the dialog box are automatically translated into function calls to **iopt** and **fopt**.

6.5.2 Specification of “net”

The **net** function includes a set of functions to define a Petri net model. The functions described in this section are recognized by the CSPL parser.

6.5.2.1 Place Functions

- void **place**(char *p); defines a place with name p.
- void **init**(char *p, int n); defines the initial number of tokens in place *p* to be *n*. By default, the number of tokens in a place is zero.

6.5.2.2 Transition Functions

- void **imm**(char *t); defines an immediate transition. (A timed transition is defined with **rateval** or **probval**.) The default firing weight for an immediate transition is 1.0.
- void **rateval**(char *t, double val); defines the firing rate of timed transition *t* as a constant value *val*. This function implicitly defines a timed transition.
- void **ratedep**(char *t, double val, char *p); defines the firing rate of transition *t* to be *val* times the number of tokens in place *p*.
- void **probval**(char *t, double val); defines the firing weight (an un-normalized probability) of immediate transition *t* as a constant value *val*.
- void **probdep**(char *t, double val, char *p); defines the firing weight of transition *t* to be *val* times the number of tokens in place *p*.
- void **priority**(char *t, int prio); defines the priority for transition *t* to be *prio*. A timed transition has the lowest priority (0) by default.

Note: For **ratedep** and **probdep**, it is an error for a firing rate or firing weight to be zero in a marking where its associated transition is enabled. Hence these methods cannot be used for an element that is not an input place for a transition and can have 0 tokens.

6.5.2.3 Arc Functions

- void **iarc**(char **t*, char **p*); defines an *input arc* from place *p* to transition *t* with multiplicity one.
- void **oarc**(char **t*, char **p*); defines an *output arc* from place *p* to transition *t* with multiplicity one.
- void **harc**(char **t*, char **p*); defines an *inhibitor arc* from place *p* to transition *t* with multiplicity one.
- void **miarc**(char **t*, char **p*, int *mult*); defines an *input arc* from place *p* to transition *t* with multiplicity *mult*.
- void **moarc**(char **t*, char **p*, int *mult*); defines an *output arc* from place *p* to transition *t* with multiplicity *mult*.
- void **mharc**(char **t*, char **p*, int *mult*); defines an *inhibitor arc* from place *p* to transition *t* with multiplicity *mult*.

6.5.3 Assert Function

This function is called by SPNP during the reachability graph construction to check the validity of each newly found marking. It returns RES_ERROR if the marking is illegal or RES_NOERR if the marking is legal. If RES_ERROR is returned, then SPNP halts.

The *assert* function allows the evaluation of a logical condition on a Petri net model. This function can especially be useful with debugging a large and complex model, since it is possible to discover simple errors, such as a missing arc or an incorrect cardinality specification. The purpose of this function is made partially obsolete with a graphical editor such as eCGE, since these tools can be designed to check for certain types of errors as a model is developed. For example, the place dialog box displays an error message if the user attempts to assign a negative

number of tokens to a place.

6.5.4 Other Functions: `ac_init`, `ac_reach`, and `ac_final`

These functions are useful for gathering information about a Petri net model during the numerical analysis of the model. The first step in performing the numerical analysis is to generate the set of all reachable markings, called the reachability graph. The functions **`ac_init`** and **`ac_reach`** are called before and after the reachability graph has been generated. The **`ac_final`** function is called after the solution has been calculated. These functions can be used to output information about the markings generated from a model.

CHAPTER SEVEN

CONCLUSIONS

7.1 Summary

This thesis introduced the eCGE tool, the enhanced version of a software package for developing a Stochastic Petri net model and has presented the design approach and functionality of this tool. This thesis combines the work of a number of contributors, and provides a framework for future development.

7.2 Future Plans

Although eCGE provides significant enhancements to the original application, there are a number of areas where the tool could be further developed. The following list describes a number of examples:

- Add other graph layout algorithms. One example would be to have an algorithm which lines up each place or transition in a model with an invisible grid so that instead of being close to lining up it becomes exactly lined up. A case study found in [69] provides a comparison between a number of algorithms.
- The CSPL format does not contain graphical information. When a CSPL file is opened, the parser assigns a default value to each Petri net element. One idea to extend this version of eCGE would be to develop a graph layout algorithm to lay out properly the elements of the opened model.
- Have more than a simple line segment for each arc. One example would be to use splines. One possibility would be to implement an algorithm to minimize arc crossings.

- Add support for multiple open documents. A partially working prototype has been implemented. However, this prototype should be taken as a general framework rather than a functional solution - there are still a number of problems to be worked out.
- Make the file format compatible with other Petri Net modeling tools. This may assist in the re-use of previous work developed using other tools that support this format. (<http://www.oasis-open.org/cover/pnml.html> describes a proposal for a Net Markup Language based on XML)
- Analyze similar applications to utilize the best qualities of each in a newly enhanced application.
- Allow a transition to represent a separate Petri net model, rather than a single element (that is, add support for hierarchical modeling of systems).
- Add scroll bars to a document window to increase the potential “work space” for developing a model.
- Document the design of the application using some formal method. Some design documentation has been developed, but it is incomplete. Examples include high level abstractions (i.e., class diagrams or dialog boxes) or the algorithms of individual pieces (i.e., the AVL tree).
- Add the ability to select and move a number of elements at once. (In the current version, only one element at a time can be selected or moved.) This ability can be useful in editing a model.
- Empirical evaluation <new bullet point>

BIBLIOGRAPHY

1. Wei, W., *Adaptation, Implementation and Integration of Graph Layout Algorithms for a Petri Net Graphical Editor*, MS Thesis in Computer Science, School of EECS, Washington State University May 2001, 80 p.
2. Travedi, K., *Stochastic Petri Net Package (SPNP) User's Manual Version 6.0*, Duke University, 1999, 184 p.
3. Hahn, J. and Kim, J., "Why Are Some Representations (*Sometimes*) More Effective?" *ACM Transactions on Computer-Human Interaction*, Volume 6, Number 3, 1999, p. 181-213.
4. Gravelle, N., *The C-Based Stochastic Petri Net Language (CSPL) Graphical Editor*, A project submitted to the Graduate Faculty of the University of Colorado at Colorado Springs in partial fulfillment of the requirements for the degree of Master of Engineering in Software Systems Engineering, Master of Engineering Program Office, 1999.
5. Douglas, J. and Kemmerer, R., *Aslantest: A Symbolic Execution Tool for Testing Aslan Formal Specifications*, International Symposium on Software Testing and Analysis, Seattle Washington, August 1994, p. 15-27.
6. Bockman, G.V., Vaucher, J., "Adding Performance Aspects to Specification Languages," *Protocol Specification, Testing, and Verification VIII*, New York, Elsevier Science, 1988.
7. Quemada, J., and Fernandez, A., "Introduction of quantitative relative time into LOTOS," *Protocol Specification, Testing, and Verification VII*, New York, Elsevier Science, 1987.

8. Reed, G. and Roscoe A., "A timed model for communicating sequential processes," *Proceedings of the 13th ICALP*, LNCS 226, Springer-Verlag, 1986, p. 314-321
9. Gerth, R. and Boucher, A., "A timed failures model for extended communicating processes," *Proceedings of the 13th ICALP*, LNCS 267, Springer-Verlag, 1987, p. 95-114.
10. Moller, F. and Tofts, C., "A temporal calculus of communicating systems," *Lecture Notes in Computer Science 458*, Springer-Verlag, 1990, p. 401-415.
11. Yi, W., "Real-time behavior of asynchronous agents," *Lecture Notes in Computer Science 458*, Springer-Verlag, 1990, p. 502-520.
12. Maxemchuk, N., and Sabnani, K., "Probabilistic verification of communication protocols," *Protocol Specification, Testing, and Verification VII*, New York, Elsevier Science, 1987.
13. Dimitrijevic, D. and Chen, M., "An integrated algorithm for probabilistic protocol verification and evaluation," *Proceedings of IEEE INFOCOM '89*, Ottawa, Ont. Canada, 1989.
14. Giacalone, A., Jou, C., and Smolka, S., "Algebraic reasoning for probabilistic concurrent systems," *Proceedings of IFIP TC2 Working Conference Programming Concepts and Methods*, 1989.
15. Larsen, K. and Skou, A., "Bisimulation through probabilistic testing," *Proceedings of the 16th ACM Symposium Principles Programming Languages*, 1989.
16. Van Glabbeek, R., Smolka, S., Steffen, B., and Tofts, C., "Reactive, generative, and stratified models of probabilistic processes," *Proceedings of the 5th IEEE International Symposium on Logic Computer Science*, 1990.

17. Marsan, M.A., Bianco, A., Ciminiera, L., Sisto, R., Valenzano, A., "A LOTOS Extension for the Performance Analysis of Distributed Systems," *IEEE/ACM Transactions on Networking*, Vol. 2 No. 2, 1994, p. 151-165.
18. Symons, F.J.W., "Introduction to numerical Petri nets, a general graphical model of concurrent processing systems," *Australian Telecommun. Res.*, vol. 14, no. 1, 1980, p. 28-33.
19. Florin, G. and Natkin, S., "Les Reseaux de Petri Stochastiques," *Technique et Science Infomaniques*, vol.. 4, no. 1, 1985.
20. Molloy, M., "Performance analysis using stochastic Petri Nets," *IEEE Transactions on Computers*, vol. 31, 1982, p. 913-917.
21. Marsan, M, Balbo, G., and Conte, G., "A class of generalized stochastic Petri nets for the performance anaysis of multiprocessor systems," *ACM Transactions in Computing Systems*, vol. 2, 1984.
22. Marsan, M., Balbo, G., Chiola, G., and Conte, G., "Generalized stochastic Petri nets revisited: Random switches and priorities," *Proc. International Workshop Petri nets Performance Models*, IEEE-CS Press, Madison, WI, 1987, p. 44-53.
23. Marsan, M., Balbo, G., Chiola, G., Conte, G., and Cumani, A., "The effect of execution policies on the semantics and analysis of stochastic Petri nets," *IEEE Transactions in Software Engineering*, vol. 15, 1989, p. 832-846.
24. Dugan, J., Trivedi, K., Geist, R., Nicola, V., "Extended Stochastic Petri nets: Applications and analysis," *Proceedings Performance 1984*, Paris, 1984.

25. Marsan, M. and Chiola, G., "On Petri nets with deterministic and exponentially distributed firing times," *Advances in Petri Nets '87*, New York: Springer-Verlag, LNCS, vol. 266, 1987, p. 132-145.
26. Meyer, J., Movaghar, A., and Sanders, W., "Stochastic activity networks: Structure, behavior, and application," *Proceedings of the International Workshop on Timed Petri Nets*, IEEE-CS Press, Torino, Italy, 1985.
27. Razouk R. and Phelps, C., "Performance analysis using timed Petri Nets," *Proceedings of the International Conference on Parallel Processing*, 1984, p. 126-129.
28. Holliday, M., and Vernon, M., "A generalized timed Petri net model for performance analysis," *Proceedings of the International Workshop on Timed Petri Nets*, IEEE-CS Press, Torino, Italy, 1985.
29. Zuberek, W., "M-timed Petri Nets, priorities, preemptions, and performance evaluation of systems," *Lecture Notes in Computer Science: Advances in Petri Nets 1985*, vol. 222, 1986, p. 478-498.
30. McConnell, S., *Code Complete*, Microsoft Press, 1993, 857 p.
31. Harel, D., Rumpe., B., "Meaningful Modeling: What's the Semantics of 'Semantics'?" *IEEE Computer*, 2004, p. 64-72.
32. Harel, D., "On Visual Formalisms," *Communications of the ACM*, Volume 31, Number 5, 1988, p. 514-530.
33. *Kommunikation mit Automaten*. Petri, C.A., Bonn: Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962, Second Edition:, New York: Griffiss Air Force Base, Technical Report RADC-TR-65--377, Vol.1, 1966.

34. Ramchandani, C., *Analysis of Asynchronous Concurrent Systems by Petri Nets*, Technical Report, MIT, Laboratory of Computer Science, Cambridge, Massachusetts, 1974.
35. Sifakis, J., "Structural Properties of Petri Nets," *Mathematical Foundations of Computer Science*, New York, 1978, p. 474-483.
36. Holliday, M.A. and Vernon, M.K., *A Generalized Timed Petri Net Model for Performance Analysis*, Proceedings of the International Workshop on Timed Petri Nets, 1995, p. 181-190.
37. Wilhoft, G., *Petri net Evaluation using APL2*, Proceedings of the International Conference on APL, ACM Press, 1992, p. 286-300.
38. Marsan, M., Balbo, G., and Conte, G., "A Class of Generalized Stochastic Petri Nets," *ACM Transactions on Computer Systems*, Vol. 2, May 1984, p. 93-122.
39. Haas, P., Shcdlcr. G., *Regenerative Simulation of Stochastic Petri Nets*, Proceedings of the Workshop on Timed Petri Nets, Torino, Italy, IEEE Computer Society Press, July 1985.
40. Jajodia, S. and Mutchler, D., "Dynamic Voting," *ACM SIGMOD*, 1987, p. 227-238
41. Pâris, JF, "Voting with a Variable Number of Copies," *Proceedings of the Sixteenth International Symposium on Fault-Tolerant Computing*, 1986, p. 50-55.
42. Pâris, JF, "Voting with Witnesses: A Consistency Scheme for Replicated Files," *Proceedings of the Sixth International Conference on Distributed Computing Systems*, 1986, p. 606-612.
43. Dugan, J. and Ciardo, G., "Stochastic Petri net Analysis of a Replicated File System," *IEEE Transactions on Software Engineering*, Vol. 15, No. 4, 1989, p. 394-401.

44. Mureta, T., "Petri Nets: Analysis, and Applications," (in Japanese) Kindai-Kagakusha, Tokyo, Japan, 1992.
45. Ciardo, G. and Trivedi, K., "A Decomposition Approach for Stochastic Reward Net Models," *Performance Evaluation*, Vol. 18, No. 1, 1993, p. 37-59.
46. Forman, I., "Petri - A UNIX Tool for the Analysis of Petri Nets," *Proceedings of 1986 ACM Fall joint computer conference*, Dallas, 1999, p. 1092 – 1098.
47. Fowler, M., *UML Distilled: Applying the Standard Object Modeling Language*, Addison-Wesley, 1997.
48. Lindemann, C., Thummler, A., Klemm, A., Lohmann, M., Waldhort, O., "Quantitative System Evaluation with DSPNexpress 2000," *Proceedings of the Second International Workshop on Software and Performance*, 2000, p. 17-19.
49. Mackay, W., Ratzer, A., Janecek, P., "Video Artifacts for Design: Bridging the Gap Between Abstraction and Detail," *Symposium on Designing Interactive Systems*, 2000, p. 72-82.
50. Movaghar, A., Meyer, J.F., Performability Modeling with Stochastic Activity networks, Proc. of the 1984 Real-Time Systems Symp., Austin, TX, 1984, p. 215-224.
51. Sanders, W., Obal, W., Qureshi, A., and Widjanarko, K., "The UltraSAN modeling environment," *Performance Evaluation*, vol. 24, no. 1-2, 1995, p. 89-115.
52. Cruz, I. And Garg, A., "Drawing Graphs by Example Efficiently: Trees and Planar Acyclic Digraphs," *Graph Drawing '94*, Princeton, New Jersey, October 1994, p. 404-415.
53. Cruz, I., "Doodle: A Visual Language for Object-Oriented Databases, ACM-SIGMOD International Conference on Management of Data, 1992, p. 71-80.

54. Cruz, I., User-defined Visual Query Languages, IEE Symposium on Visual Languages, 1994.
54. Cruz, I., Expressing Constraints for Data Display Specifications: A Visual Approach, Principles and Practice of Constraint Programming, The MIT Press, 1995, p. 443-468.
55. Travedi, K., *Stochastic Petri Net Package (SPNP) User's Manual Version 4.0*, Duke University, 1994, 57 p.
56. Wagner, M., Tew, J.D., Manivannan, S., Sadowski, D.A., and Seila, A.F., *A Standard Simulation Environment: A Review of Preliminary Requirements*, Proceedings of the 1994 Winter Simulation Conference, Lake Buena Vista, Florida USA , 1994, p. 664-672.
57. Schwetman, H., *Portable Simulation Models*, Proceedings of the 1994 Winter Simulation Conference, Lake Buena Vista, Florida USA, 1994, p. 671-672.
58. Aho, A., Sethi, R., Ullman, J., *Principles of Compiler Design*, Addison-Wesley, 1986.
59. Molloy, M., *A CAD Tool for Stochastic Petri Nets*, Proceedings of 1986 ACM Fall joint computer conference, 1986, p. 1082-1091.
60. Sheldon, F., Brake-Safe Analysis Final Report: Safety and Reliability Analysis Using Stochastic Petri Nets.
61. Gansner, E., Koutsofios, E., North, S., and Vo, K., "A Technique for Drawing Directed Graphs," *IEEE Transactions in Software Engineering*, vol. 19, no 3, 1993, p. 214-230.
62. Heiner, M., "Petri Net Based Software Validation: Prospects and Limitations," no. TR-92-022, Berkeley, CA, 1992, 69 p.

63. Dutheillet, C. and Haddad, S., "Conflict Sets in Colored Petri Nets," Proceedings of the 5th International Workshop on Petri Nets and Performance Models, Toulouse, France, 1993, p. 76-85.
64. Czerwinski, M., Cutrell, E., and Horvitz, E., "Instant Messanging: Effects of Relevance and Time," People and Computers XIV: Proceedings of HCI 2000, Vol. 2, British Computer Society, 2000, p. 71-76.
65. Maglio, P. and Campbell, C.S., "Tradeoffs in Displaying Peripheral Information," *Proceedings of the ACM Conference on Human Factors in Computing Systems*, 2000.
66. Somervill, J., Srinivassan, R., Woods, K., and Vasniak, O., "Measuring Distraction and Awareness Caused by Graphical and Textual Displays in the Periphery," *Proceedings of the 39th Annual ACM Southeast Conference*, Athens, GA, 2001.
67. Tessendorf, D., Chewar, C.M., Ndiwalana, A., Pryor, J., McCrickard, D.S., and North, C., "An ordering of secondary task display attribuites," submitted to *Conference Companion of the ACM Conference on Human Factors in Computing Systems*, 2002.
68. Somervell, C.M., Chewar, D., McCrickard, D., "Evaluating Graphical vs. Textual Secondary Displays for Information Notification," *40th Annual ACM Southeast Conference*, Raleigh, NC — April 26-27, 2002, pp. 153-160
69. Purchase, H.C., Carrington, D. and Allder, J., Empirical Evaluation of aesthetics-based graph layout, *Empirical Software Engineering*, **7**(3), pp 233-255, Kluwer Academic Publishers, 2002.
70. Cruz, I., and Tamassia, R., "Graph Drawing Tutorial," *IEEE 10th International Symposium on Visual Languages*, VL '94, St. Louis, October 1994.
71. Eades, P., *A heuristic for graph drawing*, Congressus Numerantium, 1984, p. 146-160

72. Reingold, E. and Tifford, J., Tidier drawing of trees, *IEEE Transactions on Software Engineering*, 1981, p. 223 – 228.
73. Tamassia, R., “On embedding a graph in the grid with the minimum number of bends,” *SIAM Journal of Computing*, 1987, p. 421 – 444.
74. Coleman, M.K. and Stott-Parker, D., “Aesthetics-based graph layout for human consumption,” *Software – Practice and Experience*, 1996, p. 1415 – 1438.

APPENDIX A

LAYOUT ALGORITHMS

A.1 Introduction

A Petri net model, whether it is developed on paper or using a computer-based tool, shows a syntactic representation of a system. It is theoretically impossible to exactly show the underlying meaning of the model. However, a good layout can make it easier to convey the meaning. One of the goals in the design of the application has been to create an easy-to-use architecture for developing a layout algorithm. This section provides an overview of two such algorithms: the Spring and Tree Algorithms developed by Wen Wei [1].

A.2 General Principles of a Graph Layout Algorithm

There are several aesthetics for obtaining an attractive layout of a graph. The project currently supports three such algorithms: the Spring Algorithm [1], Tree Algorithms [1], and Random Algorithm. A graph layout algorithm can provide an approximate solution for competing aesthetics. However, some readability aspects require knowledge about the semantics (as discussed in Section 2.5) of the specific graph. The main aesthetic goals are as follows. Note that some of these goals are in conflict (such as the second and third ones) [70].

- Displaying symmetry – where possible, a symmetric view of the graph should be displayed [71].
- Minimizing the number of edge crossings in a drawing [72].
- Minimizing bends – the total number of bends in polyline edges should be minimized [73].
- Distributing nodes uniformly [74].

A.2.1 Spring Algorithm

The Spring Algorithm is based on the concept of the “spring embedder” algorithm [60], which is a heuristic approach to graph drawing based on a physical system. This algorithm simulates a mechanical system consisting of springs (arcs) and nodes (places and transitions). From the initial configuration or ring positions, the system oscillates until it stabilizes at a minimum-energy configuration. It has been noted that in such a configuration, all the edges typically have relatively uniform length and nodes not connected tend to be far apart.

A.2.2 Tree Algorithm

A.2.3 Random Algorithm

The Random Algorithm is much more simple than the other two algorithms. This algorithm simply assigns each place and transition a random position on the drawing window. Its main purpose is to provide a starting point for organizing a model imported from CSPL format.

A.3 Interface for Designing a Graph Layout Algorithm

eCGE provides an environment for the development of graph layout algorithms. This environment is based on the use of lists. This section provides an overview of this interface; a more detailed description is found in section B.4.4. One of the purposes of including the Spring and Tree Algorithms was to give examples of how to use this interface.

The elements of a Petri net model are stored in three lists: PlaceList, TransList, and ArcList. Each element of a Petri net model is assigned a unique integer index. Each list defines a number of functions for working with the elements of a Petri net model. Section B.4.4 describes each of these functions in more detail. Conceptually, these functions can be divided into two categories:

querying functions and *list traversal* functions. Querying functions are used to retrieve / modify the attributes of an element.

List traversal functions are for traversing the list in order of the index. Each of the lists maintains a variable (initially `null`) to the currently selected element. This variable is updated each time one of the following traversal functions is called:

- `first()` initializes the traversal variable to start at the element with the smallest index.
- `last()` initializes the traversal variable to start from the element with the largest index.
- `prev()` backs up to the previous element in the list.
- `next()` advances to the next element in the list.

Example: Suppose the following elements are in `ArcList`: 1, 2, 5, 6, and 8. Calling `ArcList.first()` sets the pointer variable to 1. Repeated calls to `ArcList.next()` returns 2, then 5, then 6, then 8. If `next()` is called at this point, a null value (defined by `constants.invalid`) is returned.

APPENDIX B

DESIGN DOCUMENTATION

B.1 Introduction

Appendix B shows the design documentation for this project. The first section shows the architectural model of the design. The second section shows the timing diagrams for the mouse handler.

B.2 Architectural Model of the Project

The three charts in this section show the architecture of the implementation. These diagrams show the hierarchical structure of the classes. The boxes represent major system elements. Figure 19 shows a top-level chart. Figure 20 shows the classes in the mouse handler. Figure 21 shows the structure of a document.

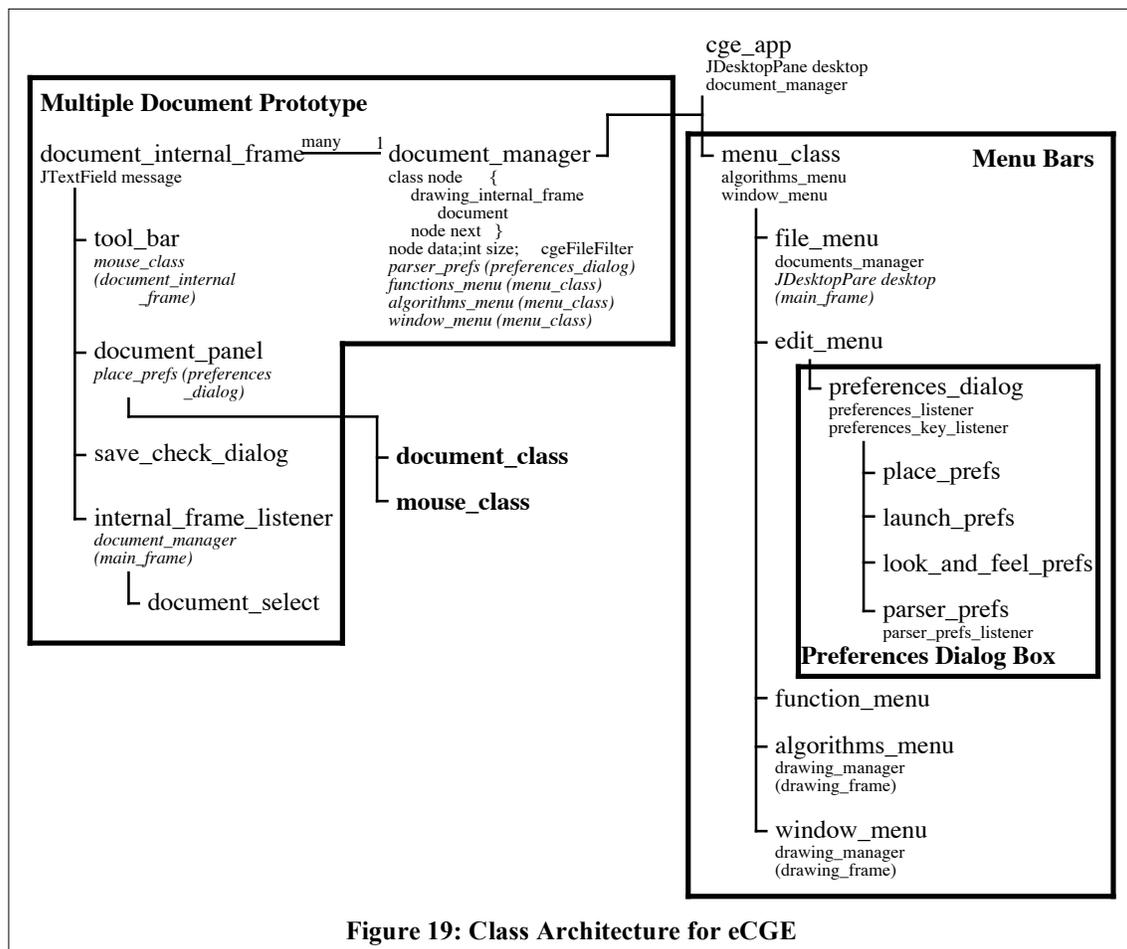
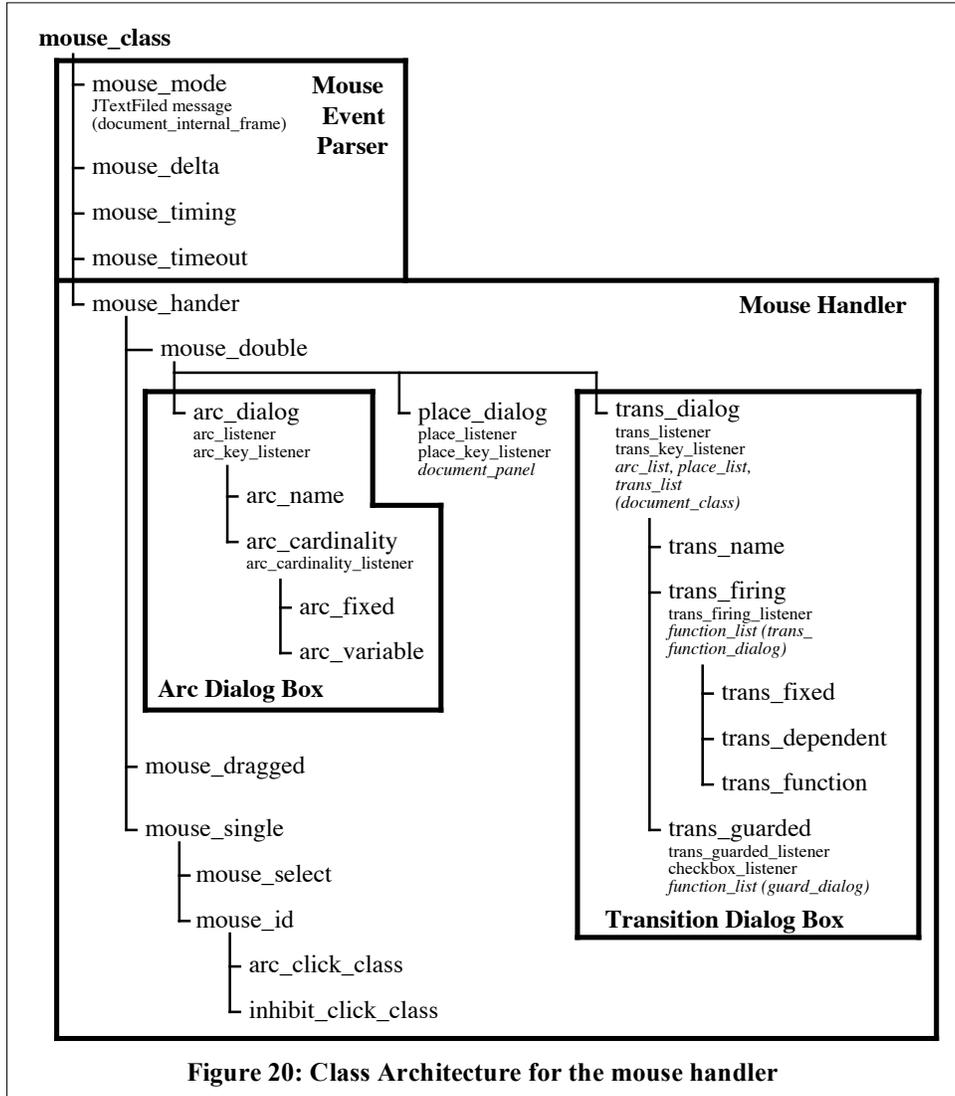


Figure 19: Class Architecture for eCGE



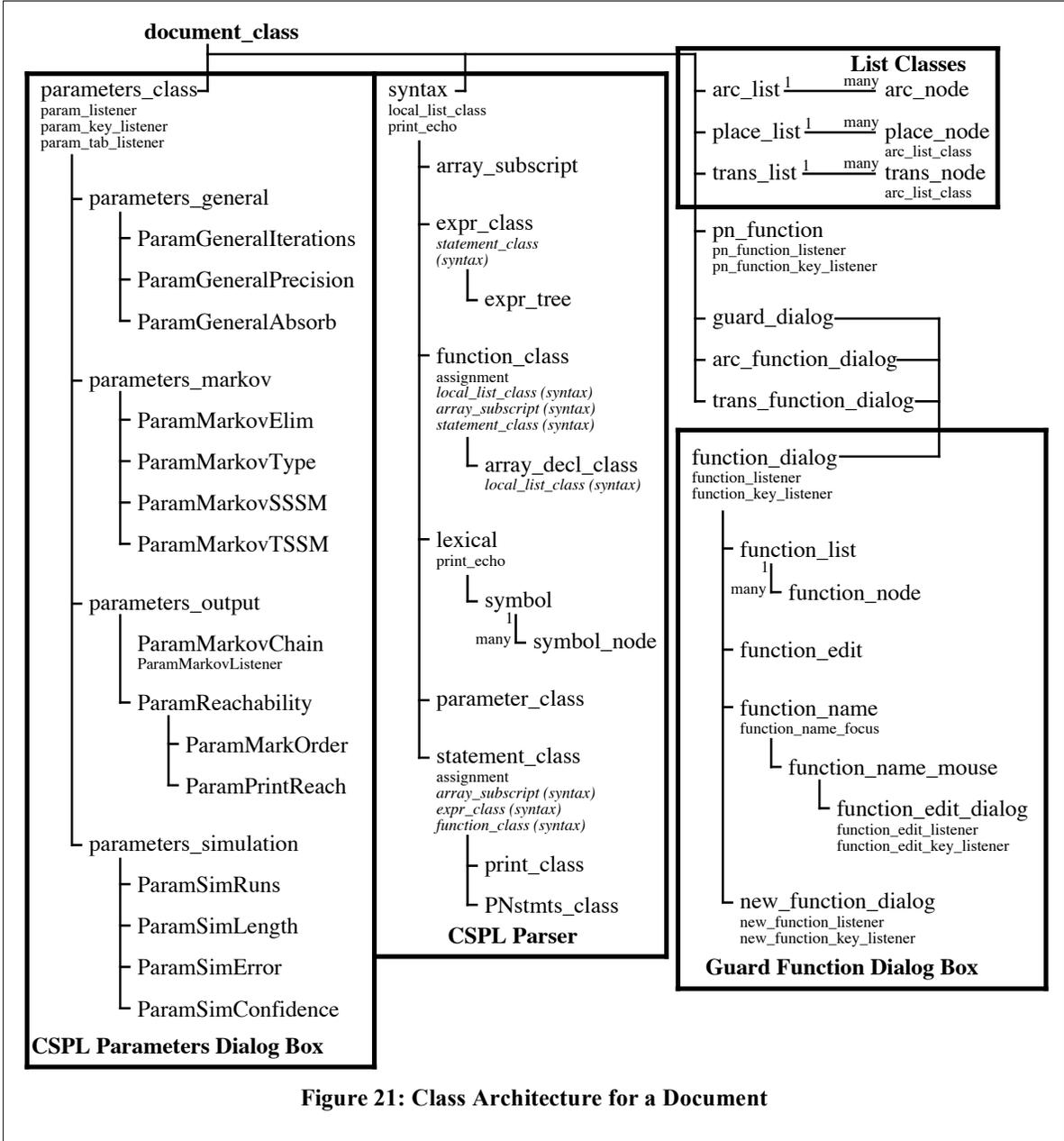


Figure 21: Class Architecture for a Document

B.3 Mouse Handler Timing Diagrams

The Java event handler mechanism provides a number of low-level mouse event functions (namely `mousePressed`, `mouseReleased`, and `mouseDragged`). These functions confer a lot of information about a mouse at a given time. Figure 22 shows how the mouse handler parses this information. This section describes how the mouse handler parses this information into three semantic events: single click, double click, and mouse drag. The mouse event parser uses a number of constants, which currently are hard-coded into the program.

- `mouse.click_time` – a bound on the time between pressing and releasing the mouse button (for distinguishing between a simple mouse click and moving an element)
- `mouse.double_click_time` – the maximum time allowed between two successive clicks for a double-click
- `mouse.max_timeout` – maximum time allowed for a single click (for distinguishing between a single click and a double click)

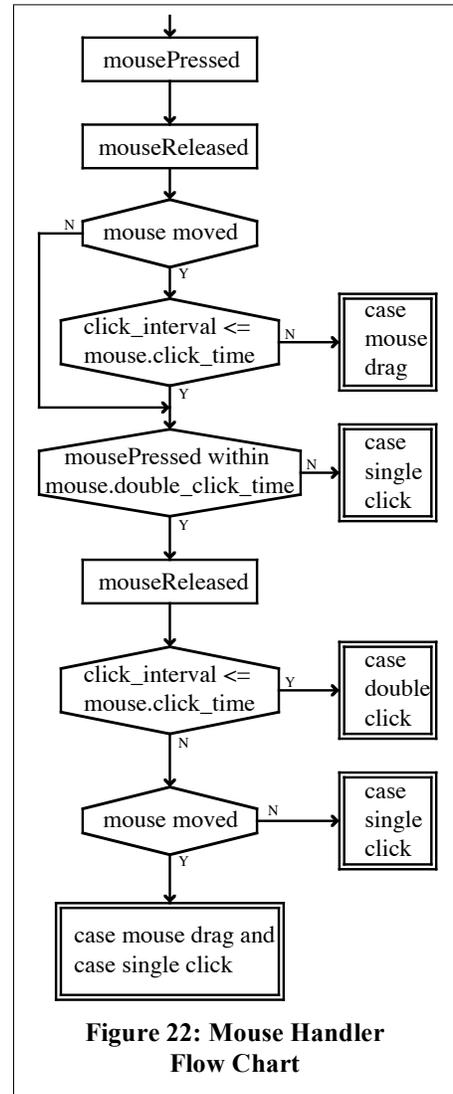


Figure 22: Mouse Handler Flow Chart

B.3.1 Single Click

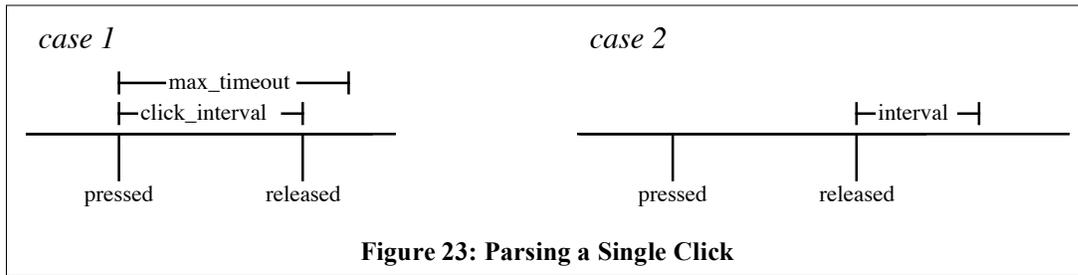


Figure 23: Parsing a Single Click

Parsing a single click has two cases, as shown by Figure 23. Case 1 represents a simple mouse click: $\text{click_interval} \leq \text{mouse.max_timeout}$. Mouse movement between pressed and released is ignored.

In case 2, there is an arbitrary amount of time between when the mouse button is pressed and when it is released. This case has two assumptions. One assumption is that no mouse movement between these two events (otherwise the sequence of events is interpreted as mouse movement). The second assumption is that a mouse pressed event does not occur within time interval.

B.3.2 Double Click

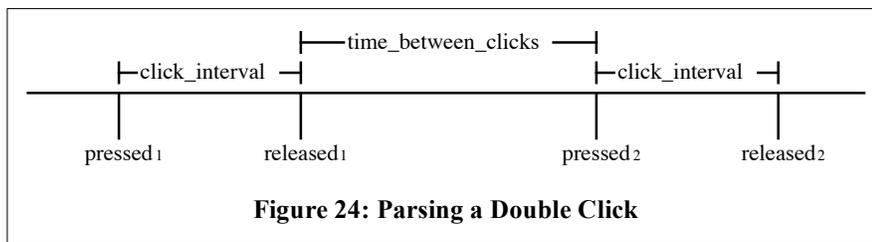


Figure 24: Parsing a Double Click

A double click is basically two single clicks occurring within a certain interval. Figure 24 shows the sequence of events required for a double click. These events are bounded by the following temporal conditions:

- $\text{released}_1 - \text{pressed}_1 \leq \text{click_interval}$
- $\text{released}_2 - \text{pressed}_2 \leq \text{click_interval}$

- $\text{pressed}_2 - \text{released}_1 \leq \text{time_between clicks}$ (mouse movement during this time frame is ignored)

B.3.3 Mouse Drag

The basic assumption for this case is mouse movement between the time the mouse button is pressed and the time the button is released. What distinguishes this case from a

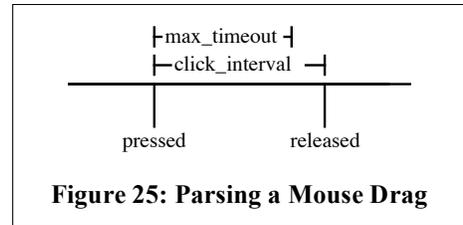


Figure 25: Parsing a Mouse Drag

single click is the duration between the mouse pressed event and the mouse released event ($\text{click_interval} > \text{mouse.max_timeout}$). This is shown graphically in Figure 25.

B.4 AVL Tree Documentation

An AVL tree is a data structure that is very well suited to storing data that supports ordering, such as names, or other miscellaneous things. Due to the fact that it is a balanced tree, very fast access to the data is supported, including inserting, deleting, finding, and querying operations. The following discussion is based on a freely available public domain AVL tree library originally written in C++. The algorithms and implementation are presented here under the terms of the GNU General Public License as published by the Free Software Foundation. The remainder of this section describes the most complex of these operations in detail: *insertion*, *deletion*, and *tree traversal*.

B.4.1 AVL Tree Data Structures

The implementation of an AVL tree uses two basic data structures: the node class and the list class. The node class stores the data and link information for a single node. The list class contains an ordered set of nodes.

The AVL node class has a balance field in addition to the usual members needed for any binary search tree. This balance factor is the maximum height of its right sub-tree minus the

maximum height of its left sub-tree. A node with balance factor -1, 0, or 1 is considered balanced. A node with balance factor -2 or 2 is considered unbalanced and requires rebalancing the tree.

```
public class avl_node{
    public avl_node left,right;    /* pointer to the node's sub-trees */
    public avl_node up;           /* pointer to the node's parent */
    public byte balance;          /* balance factor */

    public int m_id;              /* key field */
    /* ----- */                /* other fields */
} /* class avl_node */

public class avl_list{
    private avl_node data = null; /* pointer to the tree's root node */
    private avl_node curr = null; /* traversal variable: points to the current node */
    private int size = 0;        /* number of nodes in the tree */
} /* class avl_list */
```

B.4.2 Insertion

This implementation divides the algorithm for insertion into three main steps:

- **Search** for the location to insert the new item.
- **Insert** the item as a new leaf.
- **Update** balance factors in the tree that were changed by the insertion. Then rebalance the tree, if necessary.

Steps 1 and 2 are the same as for insertion into an ordinary binary tree. These two steps may lead to a violation of the tree's balancing rule. If this is the case, the third step rearranges nodes and modifies their attributes to restore the AVL balancing rule. An insertion requires at most one single rotation or double rotation.

```
void add(node new_node)
{ /* add */
    boolean found;
    arc_node curr,up;
    boolean height;
    arc_node p1;
    int dir;
```

```

up    = null;
curr  = data;
found = false;
dir   = 0;

```

```

while((curr != null) && (found==false)){
    if(new_node.m_id < curr.m_id){
        dir = 0;
        up  = curr;
        curr = curr.left;
    } /* if */
    else if(new_node.m_id > curr.m_id){
        dir = 1;
        up  = curr;
        curr = curr.right;
    } /* else if */
    else{
        found = true;
    } /* else */
} /* while */

```

Step 1: Search the tree for the insertion point of the new node. The new node will initially be a child of *curr*.

```

if(found == false){
    new_node.up = up;
    size++;

```

Step 2: Insert the node into the tree. There is a special case for a node inserted into an empty tree.

```

if(up != null){
    switch(dir){
        case 0:
            up.left = new_node;
            break;
        case 1:
            up.right = new_node;
            break;
    } /* switch */
    height = false;
} /* if(up != null) */
else{
    data = new_node;
    height = true;
} /* else */

```

```

curr = new_node;
while((curr != data)&&(height==false)){
    up = curr.up;

    /* case LEFT */
    if(up.left == curr){
        switch(up.balance){
            case 1:
                up.balance = 0;
                height = true;
                curr = up;
                break;
            case 0:
                up.balance = -1;
                curr = up;
                break;

```

Steps 3 and 4: Update balance factors and rebalance the tree. Update balance factors and re-balance the tree, starting from the inserted node and moving upward to the root. At each level, update the balance as necessary.

Case 1: Re-balance tree after insertion in *left* sub-tree. The cases for rebalancing are distinguished based on the balance factor of the child of the unbalanced node on its taller side.

```

case -1:
  p1 = up.left;
  if(p1.balance == -1){
    curr      = p1;
    up.left   = p1.right;
    p1.right  = up;
    p1.balance = 0;
    up.balance = 0;
    p1.up     = up.up;
    up.up     = p1;
    if(up.left != null)
      up.left.up = up;
  } /* if(p1.balance == -1) */

```

When *p1* has a negative balance factor, a single right rotation at *up* is required. This is shown in Figure 26.

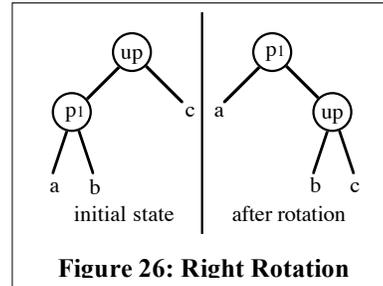


Figure 26: Right Rotation

```

/* else: p1.balance != -1 */
else{
  curr      = p1.right;
  p1.right  = curr.left;
  curr.left = p1;
  up.left   = curr.right;
  curr.right = up;

  switch(curr.balance){
    case -1:
      p1.balance = 0;
      up.balance = 1;
      break;
    case 0:
      p1.balance = 0;
      up.balance = 0;
      break;
    case 1:
      p1.balance = -1;
      up.balance = 0;
      break;
  } /* switch(curr.balance) */

  curr.balance = 0;
  curr.up      = up.up;
  p1.up        = up.up = curr;

  if(p1.right != null)
    p1.right.up = p1;
  if(up.left != null)
    up.left.up = up;
} /* else: p1.balance != -1 */

height = true;
break;
} /* switch(up.balance) */
} /* case LEFT */

```

When *p1* has a positive balance factor, a double rotation is required, composed of a left rotation at *p1* followed by a right rotation at *up*. This is shown in Figure 27. Along with this double rotation comes a small

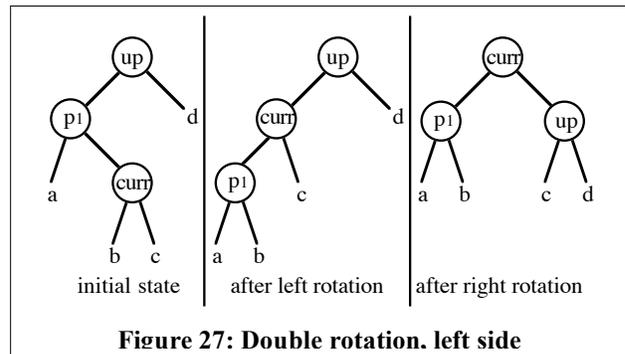


Figure 27: Double rotation. left side

bulk discount in parent pointer assignments. The parent of *curr* changes in both rotations, but the intermediate value can be ignored.

```

/* case RIGHT */
else{ /* up.right == curr */
  switch(up.balance){
    case -1:
      up.balance = 0;
      height = true;
      curr = up;
      break;
    case 0:
      up.balance = 1;
      curr = up;
      break;

```

```

case 1:
  p1 = up.right;
  if(p1.balance == 1){
    curr = p1;
    up.right = p1.left;
    p1.left = up;
    p1.balance = 0;
    up.balance = 0;
    p1.up = up.up;
    up.up = p1;
    if(up.right != null)
      up.right.up = up;
  } /* if */

```

```

/* else: p1.balance != 1 */
else{
  curr = p1.left;
  p1.left = curr.right;
  curr.right = p1;
  up.right = curr.left;
  curr.left = up;

  switch(curr.balance){
    case -1:
      p1.balance = 1;
      up.balance = 0;
      break;
    case 0:
      p1.balance = 0;
      up.balance = 0;
      break;
    case 1:
      p1.balance = 0;
      up.balance = -1;
      break;
  } /* switch */
  curr.balance = 0;
  curr.up = up.up;
  p1.up = up.up = curr;

  if(p1.left != null)
    p1.left.up = p1;
  if(up.right != null)
    up.right.up = up;

```

Case 2: Re-balance tree after insertion in *right* sub-tree. This case is symmetric to re-balancing a left sub-tree, but is included for completeness.

A positive balance factor at *p1* requires a single left rotation at *up*, as shown by Figure 28.

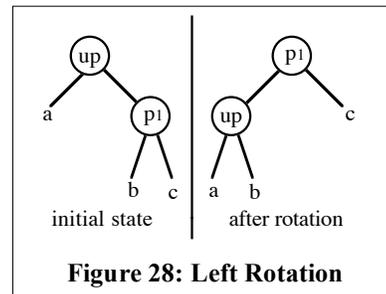


Figure 28: Left Rotation

When *p1* has a negative balance, a double rotation is required. This is composed of a right rotation at *p1* followed by a left rotation at *up*. Figure 29 shows this graphically.

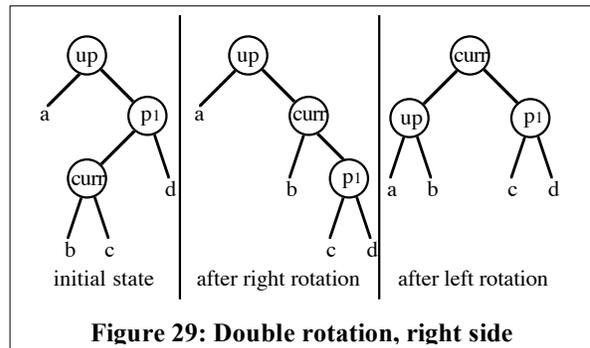


Figure 29: Double rotation, right side

```

        } /* else: pl.balance != 1 */

        height = true;
        break;
    } /* switch(up.balance) */
} /* case RIGHT */
} /* while */

if(curr.up != null){
    if(curr.up.left == up)
        curr.up.left = curr;
    else
        curr.up.right = curr;
} /* if */

else{
    data = curr;
} /* else */
} /* if(found == false) */
} /* add */

```

B.4.3 Deletion

Deletion in an AVL tree is very similar to insertion. The steps that are analogous:

- **Search** for the item to delete.
- **Delete** the item.
- **Update** balance factors and **rebalance** the tree, if necessary.
- **Finish up** and return.

When rebalancing does become necessary after a deletion, its effects are limited to the nodes along or near the direct path from the inserted or deleted node up to the root of the tree. Usually, only one or two of these nodes are affected, but, at most, a single rotation is performed at each of the nodes along this path. The actual updating of balance factors and rebalancing steps are similar to those used for insertion.

```

public boolean remove(final int m_id)
{ /* remove */
    boolean found;
    boolean root;
    arc_node curr, up;
    arc_node r, s;
    arc_node y;
    arc_node w, x;
    int dir;

```

```

boolean height;
dir = 0;
curr = data;
found = false;
while((curr!=null) && (found==false)){
    if(m_id < curr.m_id){
        curr = curr.left;
        dir = 0;
    } /* if */
    else if(m_id > curr.m_id){
        curr = curr.right;
        dir = 1;
    } /* else if */
    else{
        found = true;
    } /* else */
} /* while */

```

Step 1: Find the node to delete. This node is pointed to by *curr*.

```

if(found == true){
    up = curr.up;
    if(up == null){
        up = data;
        dir = 0;
        root = true;
    } /* if */

```

Step 2: Delete the node. At this point, we've identified *curr* as the node to delete. It is more difficult to remove some nodes from a tree than to remove other nodes. This is the actual deletion step. There are three distinct cases, described in detail below.

```

else{
    root = false;
} /* else */

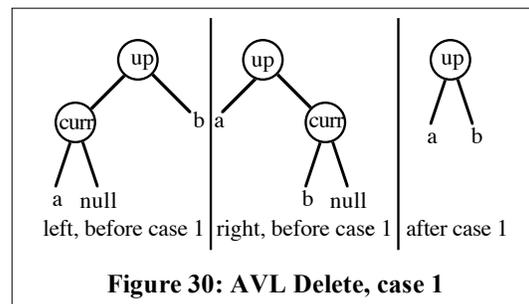
if(curr.right == null){
    if(dir == 0){ /* left side */
        up.left = curr.left;
        if(up.left != null)
            up.left.up = curr.up;
    } /* if */

    else{ /* right side */
        up.right = curr.left;
        if(up.right != null)
            up.right.up = curr.up;
    } /* else */
} /* if(curr.right == null) */

/* else: curr.right != null */

```

Case 1: the node has no right child. It is trivial to delete a node with no right child. The pointer leading to *curr* is replaced with *curr*'s left child, if it has one, or by a null pointer, if not. In other words, the deleted node is replaced by its left child. Figure 30 shows this case graphically.



```

else{
  r = curr.right;

  if(r.left == null){
    r.left = curr.left;

    if(dir == 0)
      up.left = r;
    else
      up.right = r;

    r.up = curr.up;
    if(r.left != null)
      r.left.up = r;

    r.balance = curr.balance;
    up = r;
    dir = 1;
  } /* if(r.left == null) */

```

Case 2: the node's right child has no left child. This case deletes any node *curr* with a right child *r* that itself has no left child. In this case, *r* is moved into *curr*'s place, attaching *curr*'s former left sub-tree, if any, as the new left sub-tree of *r*. The process is shown in Figure 31.

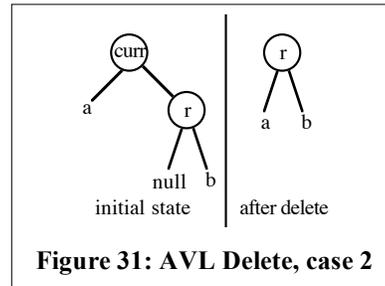


Figure 31: AVL Delete, case 2

```

/* else: r.left != null */
else{
  /* find curr's in-order successor */
  s = r.left;

  while(s.left != null){
    s = s.left;
  } /* while */

  /* update pointers for r and s */
  r = s.up;
  r.left = s.right;
  s.left = curr.left;
  s.right = curr.right;

  if(dir == 0)
    up.left = s;
  else
    up.right = s;

  if(s.left != null)
    s.left.up = s;

  s.right.up = s;
  s.up = curr.up;

  if(r.left != null)
    r.left.up = r;

  s.balance = curr.balance;
  up = r;
  dir = 0;
} /* else: r.left != null */
} /* else: curr.right != null */

```

Case 3: the node's right child has a left child. This is the "hard" case, and is shown in Figure 32. The algorithm can be divided into the following steps to make it easier to understand:

1. Let *curr*'s in-order successor, that is, the node with the smallest key value greater than *curr*, be *s*.
2. Detach *s* from its current position in the tree and put it into the spot formerly occupied by *curr*, which disappears from the tree.

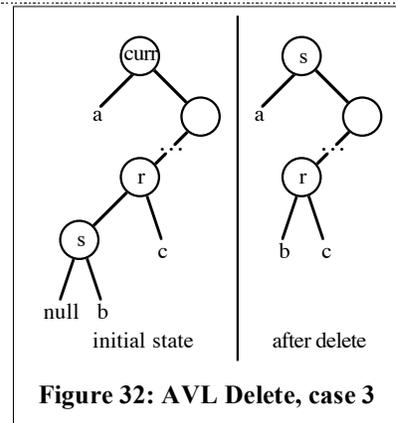


Figure 32: AVL Delete, case 3

Node *s* exists because otherwise this would be case 1 or case 2 (the reason is related to how the current node's successor is found). It is easy to detach *s* from its position for a more subtle reason: *s* is the in-order successor of *curr* and is therefore has the smallest key value in *curr*'s right sub-tree, so *s* cannot have a left child. (If it did, then this left child would have a smaller value than *s*, so it, rather than *s*, would be *curr*'s in-order successor.) Since *s* doesn't have a left child, *s* can simply replace it by its left child, if any. This is the mirror image of case 1.

```

size--;
if(root == true){
    if(size > 0){
        data = data.left;
        height = true;
    } /* if */
    else{
        data = null;
        height = false;
    } /* else */
    curr = null;
} /* if */

else{
    curr = null;
    height = true;
} /* else */

```

Special Case: *deleting the root node.* Removing the root node is a special case. This section updates *data* (the root pointer for the tree) after such a deletion.

Step 3: *Update balance factors and re-balance the tree, if necessary.* Rebalancing begins at node *up*, from whose side *dir* node *curr* was deleted.

```

while((up!=data) && (height == true)){
    y = up;

    if(y.up != null)
        up = y.up;
    else
        up = data;

    /* re-balance the on left side */
    if(dir == 0){
        if(up.left == y)
            dir = 0;
        else
            dir = 1;

        switch(y.balance){
            case -1:
                y.balance = 0;
                break;
            case 0:
                y.balance = 1;
                height = false;
                break;

```

Node *up* at the beginning of the iteration becomes node *y*, the root of the balance factor update and rebalancing. Variable *dir*, initialized at the beginning of each iteration, is used to separate the left-side and right-side deletion cases.

The loop also updates the values of *up* and *dir* for rebalancing and for use in the next iteration of the loop, if any. These new values can only be assigned after the old ones are no longer needed, but must be assigned before any rebalancing so that the parent link to *y* can be changed. For *up* this is after *up* receives *up*'s old value and before rebalancing. For *dir*, it is after the branch point that separates the left-side and right-side deletion cases, so the *dir* assignment is duplicated in each branch. The code used to update *up* is discussed later.

Re-balance tree after deletion in left sub-tree

```

case 1:
  x = y.right;

  if(x.balance == -1){
    w = x.left;

    x.left = w.right;
    w.right = x;
    y.right = w.left;
    w.left = y;

    switch(w.balance){
      case -1:
        x.balance = 1;
        y.balance = 0;
        break;
      case 0:
        x.balance = 0;
        y.balance = 0;
        break;
      case 1:
        x.balance = 0;
        y.balance = -1;
        break;
    } /* switch(w.balance) */

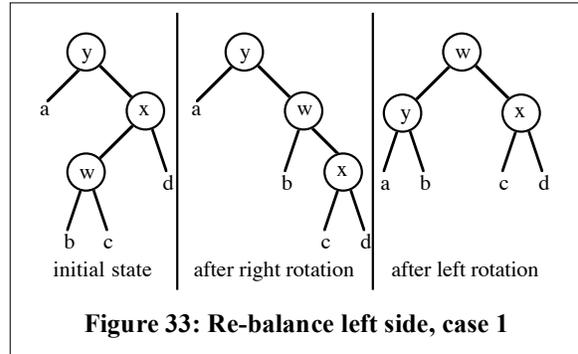
    w.balance = 0;
    w.up = y.up;
    x.up = y.up = w;

    if(x.left != null)
      x.left.up = x;
    if(y.right != null)
      y.right.up = y;

    if(dir == 0)
      up.left = w;
    else
      up.right = w;
  } /* if(x.balance == -1) */

```

Case 1: *x* has a $-$ balance. When *x* has a negative balance, a double rotation is required. This is composed of a right rotation at *x* followed by a left rotation at *y*, as shown in Figure 33.



Case 2: *x* has a $+$ or 0 balance factor.

```

/* else: x.balance != -1 */
else{
    y.right = x.left;
    x.left = y;
    x.up = y.up;
    y.up = x;

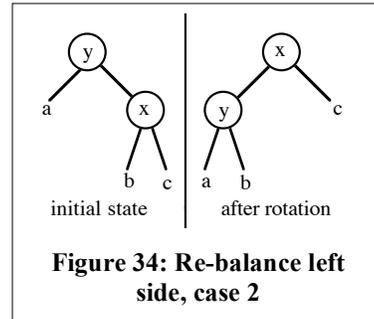
    if(y.right != null)
        y.right.up = y;

    if(dir == 0)
        up.left = x;
    else
        up.right = x;

    if(x.balance == 0){
        x.balance = -1;
        y.balance = 1;
        height = false;
    } /* if */
    else{
        x.balance = 0;
        y.balance = 0;
        y = x;
    } /* else */
} /* else: x.balance != -1 */
break;
} /* switch(y.balance) */
} /* if(dir == 0) */

```

If x has a positive or 0 balance factor, a left rotation at y is required. This case is shown in Figure 34. If x started with a balance factor of 0, then the re-balancing step is done. Otherwise, x becomes the new y for the next loop iteration, and re-balancing continues.



Re-balance tree after deletion in right sub-tree

```

/* else: dir == 1 */
else{
    if(up.left == y)
        dir = 0;
    else
        dir = 1;

    switch(y.balance){
        case 1:
            y.balance = 0;
            break;
        case 0:
            y.balance = -1;
            height = false;
            break;
    }
}

```

```

case -1:
  x = y.left;

  if(x.balance == 1){
    w = x.right;

    x.right = w.left;
    w.left = x;
    y.left = w.right;
    w.right = y;

    switch(w.balance){
      case -1:
        x.balance = 0;
        y.balance = 1;
        break;
      case 0:
        x.balance = 0;
        y.balance = 0;
        break;
      case 1:
        x.balance = -1;
        y.balance = 0;
        break;
    } /* switch(w.balance) */

    w.balance = 0;
    w.up = y.up;
    x.up = y.up = w;

    if(x.right != null)
      x.right.up = x;
    if(y.left != null)
      y.left.up = y;
    if(dir == 0)
      up.left = w;
    else
      up.right = w;
  } /* if(x.balance == 1) */

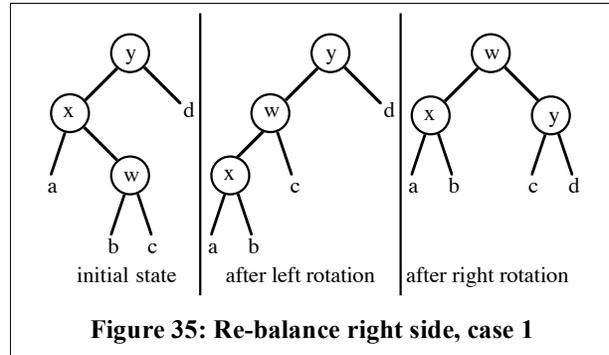
/* Case 2: x.balance != 1 */
else{
  y.left = x.right;
  x.right = y;
  x.up = y.up;
  y.up = x;

  if(y.left != null)
    y.left.up = y;

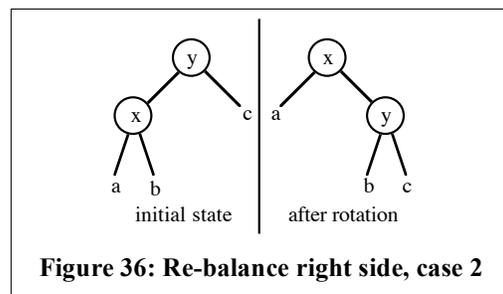
  if(dir == 0)
    up.left = x;
  else
    up.right = x;
  if(x.balance == 0){
    x.balance = 1;
    y.balance = -1;
    height = false;
  }
}

```

Case 1: *x has a positive balance factor.* When *x* has a positive balance factor, a double rotation is required. This is composed of a left rotation at *x* followed by a right rotation at *y*, as shown in Figure 35.



Case 2: *x has a - or 0 balance.* If *x* has a negative or 0 balance factor, a right rotation at *y* is required. Figure 36 shows this operation. If *x* started with a balance factor of 0, then the re-balancing step is done. Otherwise, *x* becomes the new *y* for the next loop iteration, and re-balancing continues.



```

        } /* if */
        else{
            x.balance = 0;
            y.balance = 0;
            y = x;
        } /* else */
    } /* else: x.balance != 1 */
    break;
} /* switch(y.balance) */
} /* else: dir == 1 */
} /* while(up != data) */
} /* if(found == true) */
return found;
} /* remove */

```

B.4.4 Detailed description of the list traversal functions

This section provides a detailed description of the list traversal functions presented in section A.3 of Appendix A. The functions *first*, *last*, *next*, and *prev* can be used to perform a walk of the elements in a tree. An enumeration can be in ascending or descending order, starting from the smallest item, the largest item, or somewhere in the middle. The current item is either an item in the tree or the “null item.” From the viewpoint of the tree’s user, the “null item” is represented by `constants.invalid`.

These traversal functions continue to work when the tree is modified. Any number of insertions and deletions may occur in the tree without affecting the currently selected item, with this exception: deleting the current item invalidates the traversal variable (even if the item is later re-inserted).

B.4.4.1 Starting at the first node

Sets the current pointer to the node with the smallest key field. Finding the smallest node in the tree is simply a matter of starting from the root and descending as far to the left as possible.

```

int first()
{ /* first */
    int m_id;

    if(data != null){
        current = data;
        while(current.left != null){

```

```

        current = current.left;
    } /* while */
    m_id = current.m_id;
} /* if */
else{
    m_id = constants.invalid;
} /* else */
return m_id;
} /* first */

```

B.4.4.2 Starting at the last node

Sets the current pointer to the node with the largest key field. The code for this case is the mirror image of starting from the least item.

```

int last()
{ /* last */
    int m_id;

    if(data != null){
        current = data;
        while(current.left != null){
            current = current.right;
        } /* while */
        m_id = current.m_id;
    } /* if */
    else{
        m_id = constants.invalid;
    } /* else */
    return m_id;
} /* last */

```

B.4.4.3 Advancing to the next node

Returns the next item in the tree. "Next" is defined as the item *after* the current item in alphabetical order. By convention, if there's no current item, the first item in the tree is returned. The algorithm of next(), the function for finding a successor, divides neatly into three cases.

```

int next()
{ /* next */
    int m_id;
    arc_node p,q;
    boolean found;

```

```

/* case 1 */
if(current == null){
    m_id = first();
} /* if(current == null) */

```

The current node is null. In this case the smallest node in the tree is returned.

```

/* case 2 */
else if(current.right == null){
    p = current;
    q = p.up;
    found = false;
    m_id = constants.invalid;

    while(found == false){
        if((q == null) || (p == q.left)){
            current = q;
            found = true;

            if(current != null){
                m_id = current.m_id;
            } /* if */
        } /* if */

        else{
            p = q;
            q = q.up;
        } /* else */
    } /* while(found == false) */
} /* else if(current.right == null) */

```

In this case the current node has no right child. What happens here is that the current pointer moves up the tree, one node at a time, until it turns out that the pointer moved up to the right (as opposed to up the left)

The code uses *q* to move up the tree and *p* as *q*'s child, so the termination condition is when *p* is *q*'s left child or *q* becomes a null pointer. There is a non-null successor in the former case; Figure 37 shows the situation in this case.

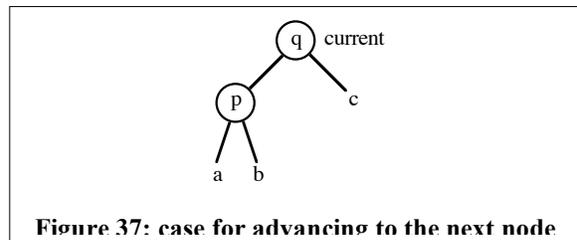


Figure 37: case for advancing to the next node

```

/* case 3 */
else{
    current = current.right;

    while(current.left != null){
        current = current.left;
    } /* while */

    m_id = current.m_id;
} /* else */
return m_id;
} /* next */

```

The current node has a right child. The successor is found by stepping to the right, then to the left until it isn't possible to go any farther (the successor is the smallest item in the node's right sub-tree).

B.4.4.4 Backing up to the next node

Returns the previous item in the tree. "Previous" is defined as the item *before* the current item in alphabetical order. By convention, if there's no current item, the last item in the tree is returned. This is the same as advancing to the next node, except that the direction is reversed (reverse *left* for *right*).

APPENDIX C

USER'S MANUAL

C.1 Introduction

A Petri net model has a main window displaying the elements of the model. The attributes of a particular element are available for editing by a pop-up dialog box. A dialog box appears when an element is selected. A text field shows any errors in the input.

C.2 Toolbar

The tool bar shown in Figure 38 is used extensively in developing and editing a Petri net model. The following list describes the use of each of the icons on the tool bar.

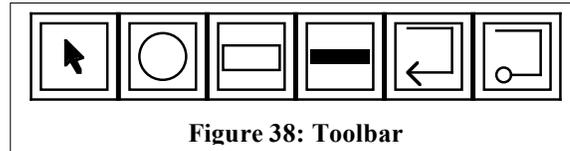


Figure 38: Toolbar

- The Cursor is used to edit a Petri net model
- Add a place to a Petri net model.
- Add a timed transition to a model.
- Add an immediate transition to a model.
- Add an arc to a model. An arc can be between a place and a transition (input arc), or from a transition to a place (output arc).
- Add an inhibitor arc to a model. An inhibitor arc can only be between a place and a transition.

C.3 Menu Bars

Many of the operations for the eCGE application can be accessed via the menu bars. The menu bar contains the File, Edit, Function, and Algorithm menus.

C.3.1 File Menu

The **File** menu focuses on commands to work with files. These commands are as follows:

- New - Creates a new (empty) document.
- Open - This command opens an existing Petri net model in an edit window. The model can be in eCGE or CSPL format.
- Close – This command closes the active window. Any changes to the model can be saved.
- Save - The **save** command saves the file in the active edit window to disk.
- Quit - This command quits eCGE. Any open files will be saved before the application exits.

C.3.2 Edit Menu

The **Edit** menu currently contains one command. In the future, this menu will be extended to include the standard editing commands (Cut, Copy, and Paste). These commands will be implemented after eCGE has been upgraded to support multiple documents.

C.3.3 Functions Menu

C.3.3.1 Parameters

The Parameters dialog boxes provides a way to set the CSPL parameters for a Petri net model. Each of these parameters is described in section C.7.

C.3.3.2 Guard Functions

At times, inhibitor arcs or transition priorities can specify a given behavior only through awkward subnets that only obfuscate the logic of the model. An alternative in these cases is the definition of a marking-dependent enabling function (or guard). This dialog box provides a way

to define and edit guard functions. Once a guard function is defined, it can be associated with a transition (see the section on Transitions).

C.3.3.3 Transition Functions

This dialog box is used to specify guard functions. A guard function is used to selectively disable a transition based on marking-dependant quantities (for example, the number of tokens in a place).

C.3.4 Algorithms Menu

The eCGE application currently supports three graph layout algorithms: the Spring Algorithm, the Tree Algorithm, and the Random Algorithm. These algorithms are described in Appendix A.

C.4 Places

Figure 39 shows the dialog box for defining the attributes of a place. Two fields are defined: the *name* and the *number of tokens*. Each place may contain any non-negative number of tokens. The default is zero. In this

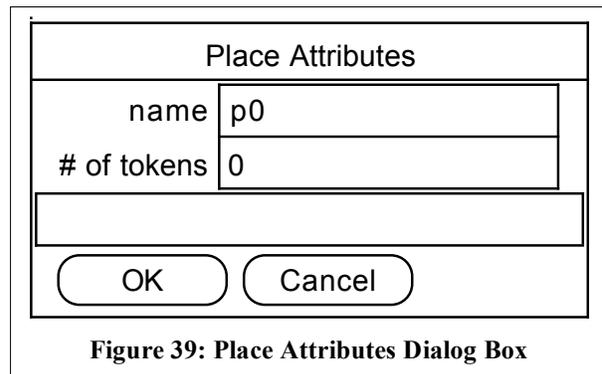


Figure 39: Place Attributes Dialog Box

example, the CSPL functions generated would be: `place("p0"); init(("p0",0);`

C.5 Transitions

In a Stochastic Petri net model, only two distribution types are allowed: exponential and deterministic with value 0. Transitions with an associated exponential distribution are said to be timed (drawn as a box); transitions with zero time distribution are said to be immediate (drawn as a solid bar).

Timed transitions are useful in describing the time lapse between consecutive events. Immediate transitions provide a probabilistic way of describing the selection among different possible events. A transition is enabled if all of the following conditions (enabling rules) are met:

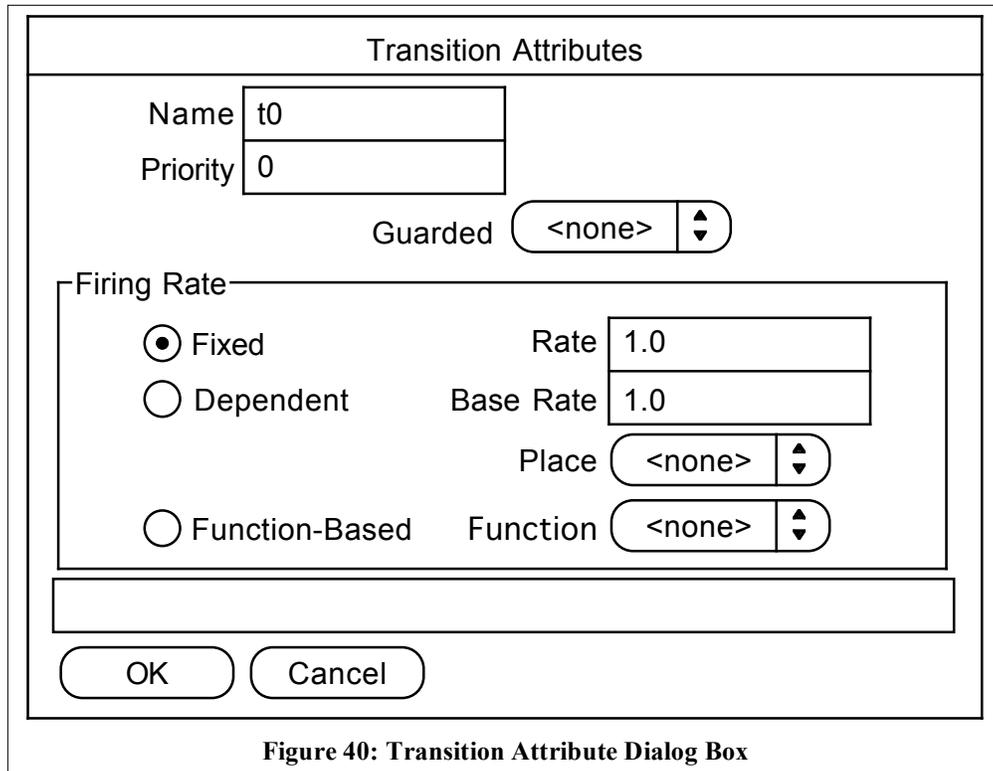
- The number of tokens in each input place is at least equal to the multiplicity of the input arc from that place.
- The number of tokens in each input place *with an inhibitor arc* is less than the multiplicity of the input inhibitor arc from that place.
- The enabling function of the transition (if any is assigned) returns **true** – which is the default if a function is not assigned to the transition.

C.5.1 Transition Properties

Figure 40 shows the dialog box for defining the attributes of a transition. The first field defines the *name* of the transition. The second field sets the *priority*: this is one method of selectively disabling a transition in a marking that would otherwise enable it. If S is the set of transitions enabled in a marking and if the transition with the highest priority among them is k , then any transition in S with priority lower than that of transition k will be disabled. The priority can be used to define explicit precedence relationships within a model. The third field associates a transition with a previously defined guard function.

C.5.2 Firing Rate

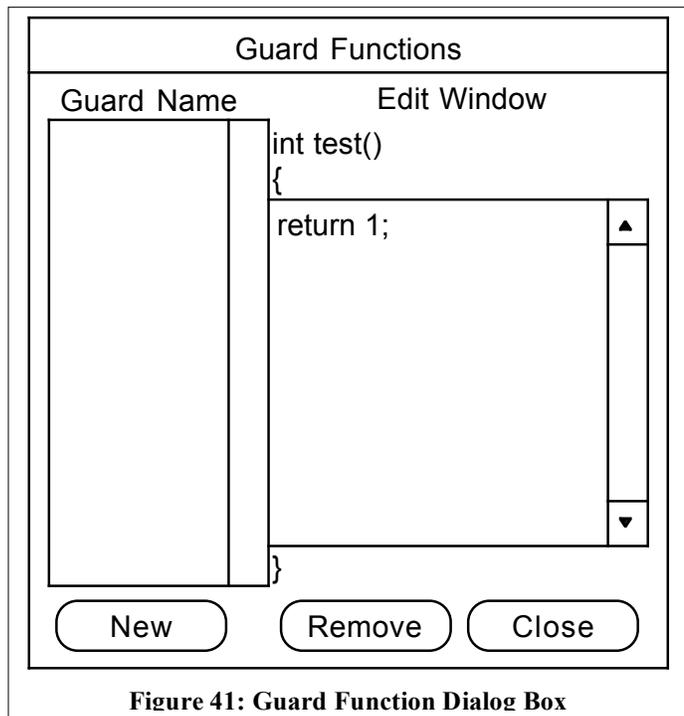
The Firing Rate panel allow the user to specify the firing rate for a transition. This value can be fixed (constant), dependent on a marking-dependent quantity, or defined by an arbitrary function. An example of a marking-dependent quantity is to define firing rate to be proportional to the number of tokens in a place.



In many other cases, though, a more general type of marking dependency is required. This is achieved by defining a marking-dependent function of type **double**. This function is evaluated whenever an associated transition may be enabled. The value this function returns specifies the firing rate or firing probability for the transition in the current marking.

C.5.3 Guard Functions

At times, inhibitor arcs or transition priorities can specify a given behavior only through awkward sub-nets that only obfuscate the actual logic of a model. In



these cases, the definition of a marking-dependent enabling function (or *guard*) may be

preferable. If the selected function evaluates to 0 in a marking, then the transition is disabled in the current marking. The dialog box for specifying a guard function is shown in Figure 41.

C.6 Arcs

Places and transitions are connected by directed arcs. Input arcs and inhibitor arcs connect places to transitions and output arcs connect transitions to places. The firing of a transition is conditioned by the presence of tokens in each of its input places. It is possible to condition this firing by the absence of tokens in an input place; this is represented by an inhibitor arc. An inhibitor arc from a place to a transition has a small circle rather than an arrowhead at the transition. Arcs can only connect a place to a transition (input arcs), or a transition to a place (output arcs). Figure 42 shows the dialog box for defining the properties of an arc.

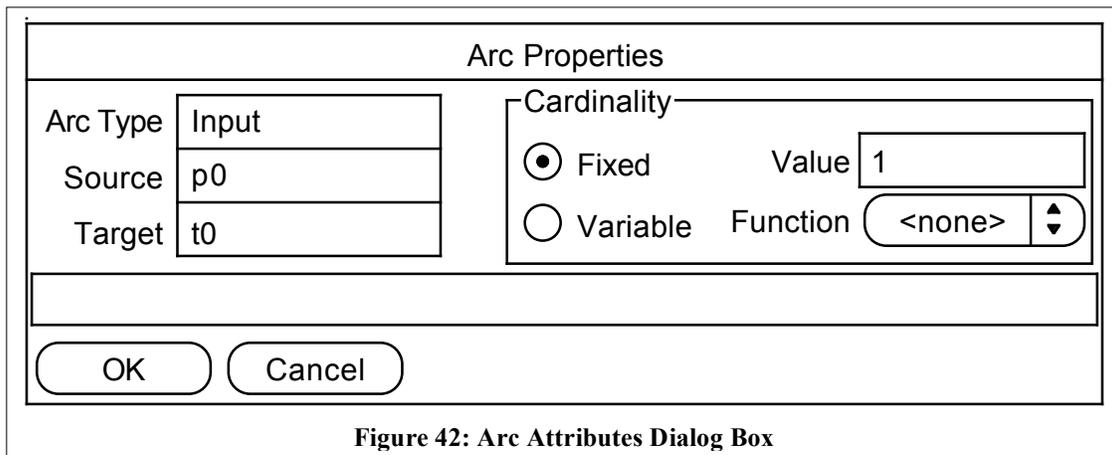


Figure 42: Arc Attributes Dialog Box

C.6.1 Cardinality

This panel defines the cardinality of an arc. The cardinality can be a fixed value or based on a marking-dependent function.

C.6.1.1 Fixed

A multiplicity (positive integer) may be attached to each arc. A multiple arc is an arc where the multiplicity is greater than one. Intuitively, a multiple arc with multiplicity k can be thought of as k arcs having the same source and destination.

C.6.1.2 Variable (or Marking Dependent) Cardinality

At times, inhibitor arcs or transition priorities can specify a given behavior only through awkward subnets that only obfuscate the actual logic of a model. In these cases, the definition of a marking-dependent enabling function is preferable. The dialog box for defining a marking-dependent enabling function for an arc is similar to defining a guard function for a transition (described in section C.5.3).

C.7 Parameters

The CSPL parameters dialog box contains settings to direct how the how SPNP translates the specified Petri net model into the underlying Markov chain for analysis, and to direct how the analysis is run. The various CSPL parameters are divided between four panels: General (Figure 43), Output (Figures 44-45), Markov (Figure 46), and Simulation (Figure 47). The following sections describe each of these panels in more detail.

C.7.1 General Parameters

Iterations specifies the maximum number of iterations allowed for the numerical solution. Any nonnegative integer can be specified. `iopt(IOP_ITERATIONS, 2000);`

The **Precision** specifies the minimum precision required from the numerical solution. The numerical solution will stop either if the precision is reached, or if the maximum number of iterations is reached. `fopt(FOP_PRECISION, 1.0e-6);`

The **Absorbing Marking Value** specifies the value of the rate from each absorbing marking back to the initial marking. If this rate is positive, these markings will not correspond to absorbing states in the continuous time Markov Chain. This is useful to model a situation that would otherwise require a large number of transitions to model this “restart.” The numerical results will depend on the value specified for this option. `fopt(FOP_ABS_RET_M0, 0.0);`

2000	Iterations
1.0E-6	Precision
0.0	Absorbing Marking Value

- Calculate Cumulative Probs
- Perform Sensitivity Analysis
- Debug Mode On
- Absorbing Markings OK
- Vanishing Loops OK
- Vanishing Initial Marking OK
- Transient Initial Marking OK

Figure 43: General Parameters

The first check box specifies whether the **cumulative probabilities** should be computed: `iopt(IOP_CUMULATIVE, VAL_YES);`. The next check box specifies whether **sensitivity analysis** should be performed. Note: if this option is selected, the solution methods type (Markov panel) must have the *Continuous Time* value, and vanishing markings must be eliminated during the reachability graph construction (Markov panel). At present, the application does not explicitly check these constraints.

If **Debug Mode On** is selected, SPNP outputs (on the “stderr” stream) the markings as they are generated. This feature is extremely useful when debugging a Petri net model. The next options specify respectively whether **absorbing markings**, **transient vanishing loops**, a **vanishing initial marking**, and a **transient initial marking** are acceptable or not.

- If an option is selected, the program will signal such occurrences, but it will continue the execution if it is possible.
- If an option is not selected, the program will stop if the condition is encountered.

C.7.2 Output File Parameters

Reachability Graph Options

Marking Print Order

Canonic

Lexical

Matrix

Eliminate Vanishing Markings

Yes

No

Only Tangible Markings

Print Reachability Graph

Merge Tangible and Vanishing Markings

Print Full Markings

Print Place and Transition Names

Print CTMC Derivative

Figure 44: Reachability Graph Options

Figure 44 shows the output options for the reachability graph. The **Marking Print Order** specifies the order in which the markings are printed. Three orderings are defined: canonic, lexical, and matrix.

- With **Canonic** order, markings are printed in the order they are found, in a breadth-first search starting from the initial marking and in increasing order of enabled transition indices. It is the most natural order and it is particularly helpful when debugging a model.
- With **Lexical** order, markings are printed in increasing order, where markings are compared as words in a dictionary. This order may be useful when searching for a particular marking in a model with a large state space.

- With **Matrix** order, markings are printed in the same order as the states of the two internal Markov chains: the discrete time Markov chain (DTMC) corresponds to the vanishing markings, and the continuous time Markov chain (CTMC) corresponds to the tangible markings. This corresponds to the following ordering: vanishing, tangible non-absorbing, and tangible absorbing, each of these groups ordered in canonical order.

Eliminate Vanishing Markings specifies whether the set of vanishing markings should be printed. The third selection indicates that only the tangible markings should be printed. The **Print Reachability Graph** check box specifies whether the reachability graph should be printed. The **Merge Tangible and Vanishing Markings** option specifies whether these two classes of markings are printed as one list or in two separate lists. **Print Full Markings** specifies whether the markings are printed in long format, where some of the markings have zero number of tokens in all the places; or short format, where for each printed out marking, there is at least one place which has non-zero tokens.

Print Place and Transition Names specifies whether the names should be used to indicate the places and transitions involved when printing the reachability set and graph, instead of the index. Using names increases the size of the output file, but it is useful when debugging a Petri net model. **Print CTMC Derivative** specifies whether the derivative with respect to the model parameters should be printed in the “.mc” output file.

The **Markov Chain File Options**, shown in Figure 45, specifies whether the Markov chain (“.mc”) file should be generated. This file describes the Markov chain (continuous time or deterministic

Markov Chain File Options

Generate Markov Chain File

Use From-To Format

Use To-From Format

Generate Probability File

Generate DTMC Probability File

Generate Dot Graph File

Figure 45: Other Output Options

time, depending on the settings) derived from a Petri net model; the vanishing markings are absent and only numeric rates appear. There are two formats for the data in this file:

- **Use From-To Format** prints the transition matrix.
- **Use To-From Format** prints the transpose of the transition matrix.

Generate Probability File specifies whether or not the “.prb” file is generated. This output file describes the transient and steady-state probability for each tangible marking; it corresponds to the result of the CTMC solution (even when the actual solution used is a DTMC).

Generate DTMC Probability File specifies whether or not the “.prbdtmc” file is generated. This output file contains the numeric results of the embedded Discrete Time Markov Chain (DTMC). **Generate Dot Graph File** specifies whether or not the “.dot” file is generated or not. This output file contains a description of the Petri net in the *dot* graph language.

C.7.3 Markov Chain Options

The Markov Chain Options are shown in Figure 45. **Eliminating Vanishing Markings** specifies the method by how vanishing markings are managed and eventually eliminated. The three options are described in the list below. Users are encouraged to use the first option, as this usually results in the fastest solution and the lowest memory requirements. However, these are rare pathological cases where this option will actually result in larger memory requirements than for the “Never” option.

- Specifying **During Reachability Graph Construction** means that vanishing markings are eliminated during the reachability graph construction. With this option, any type of solution is possible, but vanishing (non-absorbing) loops are considered an error and measures related to intermediate transitions are not computed.

Eliminate Vanishing Markings <input checked="" type="radio"/> During Reachability Graph Construction <input type="radio"/> After Reachability Graph Construction <input type="radio"/> Never
Type <input checked="" type="radio"/> Continuous Time <input type="radio"/> Discrete Time
Steady State Solution Method <input checked="" type="radio"/> SOR <input type="radio"/> Gauss-Seidel <input type="radio"/> Power
Transient State Solution Method <input type="radio"/> Transient/Uniformization <input checked="" type="radio"/> Poisson

Figure 46: Markov Parameters

- Specifying **After Reachability Graph Construction** means that the reachability graph constructed includes explicitly the vanishing markings, but these are then eliminated numerically before generating the underlying CTMC. With this option, any type of solution is possible, and vanishing (non-absorbing) loops present no problem, but measures related to immediate transitions are not computed.

- Specifying **Never** means that the stochastic process being considered

explicitly regards the vanishing markings as ordinary states. With this option, only a steady-state solution is possible. Using an embedded DTMC, measures related to immediate transitions are computed, and vanishing (non-absorbing) loops present no problem.

The solution **Type** specifies the solution method for solving a Petri net model. There are two solution types, *continuous time* and *discrete time*, which are described in the following list.

- Using **Continuous Time** will transform the Stochastic Reward Net into a CTMC. SPNP can perform transient and sensitivity analysis only by reducing the underlying Stochastic Reward Net for a Petri net model to a CTMC. Hence this setting should be used when these type of analysis is needed.

- Using **Discrete Time** will use an alternative solution approach, where the vanishing markings are not eliminated and an embedded DTMC is solved instead.

SPNP provides three methods for solving the **steady-state solution** of a Markov chain.

These algorithms are described in the following list.

- **SOR** for Steady-State SOR (Successive Overrelaxation). This is generally the fastest algorithm.
- **Gauss-Seidel** for Steady-State Gauss-Seidel. This algorithm is useful in those cases where SOR does not converge, and vice-versa.
- **Power** for Steady-State Power-Series Algorithm. This algorithm has a better convergence performance than the other two, but is much slower.

There are two options for solving the **transient-state solution method** for a continuous time Markov chain. **Transient/Uniformization** is used for Transient Solution using Standard Uniformization. The second specifies a **Poisson** distribution, calculated using the Fox and Glynn method. (A Poisson distribution models the number of independent events that occur in a fixed amount of time or space: for example, the number of customers that arrive to a store during one hour, or the number of defects found in 30 square meters of sheet metal.)

C.7.4 Simulation Options

The SPNP package allows the use of discrete-event simulation to study the behavior of a system at or up to a point in time. At present, the eCGE application only supports the specification of the following CSPL options. These are shown in Figure 47. The **Simulate System** check box option toggles whether the solution to a Petri net model is calculated numerically or with discrete-event simulation.

The **Cumulative Simulation Time** option toggles between two modes of data collection. If the field is set, the data is to be collected cumulatively (from zero to Maximum Iteration Run Time). Otherwise the data is collected at a point of time (Maximum Iteration Run Time). This field is set by default (VAL_YES).

Simulate System
 Cumulative Simulation Time
 Number of Runs
 Maximum Iteration Run Time
 Maximum Error
 Confidence Interval
 90%
 95%
 99%

Figure 47: Simulation Parameters

Number of Runs specifies the maximum number of simulation runs to be performed to obtain meaningful statistics. **Maximum Error** specifies the target width of the confidence interval relative to the point estimate. The **Confidence Interval** specifies the confidence to be used when computing the confidence interval. The default value is 95%.

If the *maximum error* is set to zero, then SPNP will perform as many runs as needed to achieve the specified relative error. If the *number of runs* is defined, then SPNP runs the simulation for this number of iterations. If neither the *maximum error* or the *number of runs* is defined, the simulation is run until a five percent relative error is reached.