CNJ: A Visual Programming Environment For Constraint Nets

by

Fengguang Song

B.Eng., Zhengzhou University, 1996

M.Eng., Nanjing University of Aeronautics and Astronautics, 1999

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

we accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

October 2002

ABSTRACT

The *Constraint Nets* model (CN) proves to be useful for a wide variety of purposes, ranging from intelligent agent systems, and real-time embedded systems, to integrated hybrid systems with various time structures: discrete, continuous, and time-based. This thesis describes a new visual programming environment called CNJ (Constraint Nets in Java) which utilizes component-based technology.

CNJ uses JavaBeans, Bean Introspection, drag-and-drop, and Java Swing MDI (Multiple Document Interface) technologies, as well as XML-based CNML as its standard interchange format. The environment supports constraint net modeling, simulation, and animation for hybrid systems. Furthermore, it provides support for a top-down design, middle-out design, and bottom-up design where the *module bean* can be reused anywhere in any other CN model, saving designers time and effort. As an experiment, a hybrid dynamic elevator system is developed successfully, and test results confirm the effectiveness of the tool for hybrid system modeling and real-time simulation.

CONTENTS

ABSTRACT ii

CONTENTS iii

LIST OF TABLES vi

LIST OF FIGURES vii

ACKNOWLEDGEMENTS ix

DEDICATION x

CHAPTER 1 INTRODUCTION 1

1.1 Thesis Statement 1

1.2 Motivation 2

1.3 Related Work 3

1.4 CNJ Overview 5

1.5 Thesis Outline 7

1.6 Contributions 7

CHAPTER 2 BACKGROUND 9

2.1 Component-Based Software Engineering 9

2.2 JavaBeans Component Architecture 10

2.3 Graphical User Interface Design 12

2.3.1 UI Design Guidelines 12

2.3.2 Java Look and Feel Style 13

2.4 Visual Programming Languages 14

2.4.1 Classification and Concepts 14

2.4.2 Automated Control Engine 16

2.5 Summary 16

CHAPTER 3 CONSTRAINT NETS 18

3.1 Constraint Net Model 18

3.1.1 Constraint-Based Intelligent System 18

3.1.2 Constraint Net Syntax and Semantics 20
3.2 Requirement Specification with Timed ∀-Automata 21
3.3 Behavior Verification 23
3.4 Summary 23
CHAPTER 4 CONCEPTS AND ARCHITECTURE OF CNJ 25
4.1 Main Idea 25
4.2 Requirement Analysis 27
4.2.1 System Requirements 27
4.2.2 User Groups 29
4.3 System Architecture 30
4.4 Summary 34
CHAPTER 5 GRAPHICAL USER INTERFACE DESIGN 35
5.1 User Interface Architecture 35
5.1.1 GUI Introduction 35
5.1.2 GUI Composition 37
5.2 Multi-threaded Panels 38
5.3 Important UI Classes 39
5.3.1 DrawPane Class 39
5.3.2 PropertySheetPanel Class 41
5.4 Constraint Net Nodes 44
5.4.1 Transduction Bean 44
5.4.2 Location Bean 46
5.4.3 Clock Bean 46
5.4.4 Module Bean 46
5.4.5 Connection 47
5.5 Summary 47
CHAPTER 6 CNJ SIMULATION 49
6.1 Java Event Mechanism 49
6.2 The Compilation Step of CNJ 53
6.3 Dataflow Issue 54
6.3.1 Data-driven and Demand-driven Models 55

6.3.2 Dataflow in CNJ 55

6.3.3 Transduction Scheduling Algorithm 56

6.4 Discussion of "Real-time" 60

6.5 Summary 62

CHAPTER 7 CONSTRAINT NET MARKUP LANGUAGE **63**

7.1 XML 63

7.2 CNML 65

7.3 Summary 67

CHAPTER 8 A REALISTIC CN MODEL **68**

8.1 The Elevator System 68

8.1.1 Top-level Hybrid Model 69

8.1.2 The Continuous ELEVATOR Module 71

8.1.3 The Discrete CONTROL1 Module 73

8.2 Elevator System Modeled in CNJ 73

8.3 Elevator System Simulation and Animation in CNJ 74

8.4 Summary 76

CHAPTER 9 EXPERIMENTAL RESULTS **78**

9.1 Methodology 78

9.2 Results 79

9.3 Summary 82

CHAPTER 10 CONCLUSIONS AND FUTURE WORK **83**

10.1 Conclusions 83

10.2 Future Work 84

BIBLIOGRAPHY **86**

APPENDIX A CNML FILE FOR THE CAR DYNAMICS MODEL **89**

APPENDIX B THE ELEVATOR SYSTEM MODELED IN CNJ **100**

LIST OF TABLES

Table 5.1 DrawPane class 41

Table 6.1 Simulation output of a simple constraint net 53

Table 9.1 Simulation results for Example 1, 2, 3 80

Table 9.2 Simulation results for the Elevator system (Example 4) 81

List of Figures

Figure 1.1 The Graphical User Interface of CNJ 5
Figure 3.1 The structure of a constraint-based agent 19
Figure 3.2 An integration system in constraint net 20
Figure 3.3 ∀-automata 22
Figure 3.4 Timed ∀-automata 22
Figure 4.1 Putting together a system from bean objects 27
Figure 4.2 Architecture of CNJ 30
Figure 4.3 Class architecture graph 32
Figure 5.1 CNFrame composition 36
Figure 5.2 Containment hierarchy of CNJ GUI 38
Figure 5.3 Method of setTarget( ) in BeanPropertySheet 43
Figure 5.4 CN bean classes 44
Figure 5.5 PropertyEdit of Transduction 45
Figure 5.6 Transduction definition for "minus" 45
Figure 6.1 Three types of PropertyChangeEvents in CNJ Simulation 51
Figure 6.2 A constraint net to represent $x(t) = x(t-1) + step$ 52
Figure 6.3 A transduction scheduling example 58
Figure 6.4 Transduction scheduling algorithm 60
Figure 7.1 XML document life cycle 64
Figure 7.2 CNML Elements 67
Figure 8.1 3-floor elevator system 69
Figure 8.2 A hybrid model of the Elevator system 70
Figure 8.3 The continuous ELEVATOR module 71
Figure 8.4 The discrete CONTROL1 module 73
Figure 8.5 The Elevator system modeled in CNJ 75
Figure 8.6 The Elevator animation window 77
Figure 9.1 Example 1 (a simple adder) 79

Figure 9.2 Example 2 (a two-level adder) 79
Figure 9.3 Example 3 (the Car Dynamics) 80
Figure B-1 BUTTON Module 100
Figure B-2 FlipFlop Module 100
Figure B-3 ResetSignal Module 101
Figure B-4 EVENT Module 101
Figure B-5 CONTROL1 Module 102
Figure B-6 UpRequest Module 102
Figure B-7 DownRequest Module 103
Figure B-8 StopRequest Module 103
Figure B-9 ServingState Module 104
Figure B-10 Command Module 104
Figure B-11 ELEVATOR Module 105
Figure B-12 CONTROL0 Module 105
Figure B-13 BODY Module 106
Figure B-14 HOME Module 106
Figure B-15 FLOOR Module 106

ACKNOWLEDGEMENTS

I would like to gratefully acknowledge my supervisor, Dr. Alan Mackworth. Numerous motivational and instructional discussions with him ensured the success of this project. His wealth of knowledge in computational intelligence and constraint-based systems assisted me in identifying the critical and interesting issues of this research. His attitude towards research and professionalism also inspired me to continuously challenge myself to reach new levels. This work would not have been possible without his enthusiastic supervision, invaluable guidance, and generous financial support. I also wish to thank Dr. Jim Little for taking the time to be my second reader, and for providing me with many insightful comments.

I would like to thank Yu Zhang (IBM) and Ying Zhang (Xerox Parc). They gave me lots of valuable suggestions and long hours of discussion over various issues related to this project. My experience here in the LCI lab is full of opportunities to learn, not only from faculty and students who are experts in the field, but also from industrial fellows and sponsoring companies. I owe my gratitude to those people who are particularly instrumental in contributing to my experience in the LCI lab at UBC. They are Dr. Jim Little, Dr. Anne Condon, Valerie McRae, Robert St-Aubin, and Xiaoming Zhou. I really appreciate Lisa Beckett's great help with my technical writing.

I am deeply grateful to my parents Linbin Song, and Suzhen Ji, for their never ceasing love and support for my whole life. Their confidence in me and encouragement of me are the impetus of this hard work. I am also grateful to my wife Lan Lin who can always lift up my spirits even in the worst of times. I also thank my brother Xuguang Song and my sister Min Song for their constant inspiration and comfort.

This work was supported by IRIS and NSERC.

To my parents and family,

for their support, love, and enthusiasm.

CHAPTER 1 INTRODUCTION

The complexity of software development is rapidly increasing during this decade and it often exceeds human intellectual capability. Component-based technology, an exciting new research area, promises to deliver scalable and reusable software because of its decomposition, abstraction, and hierarchy properties. The Constraint Nets model (CN) was developed by Zhang and Mackworth to represent hybrid dynamic systems. It is a family of visual programming languages (VPLs) with inherent graphical icons and hierarchies.

In this thesis, a visual programming environment called CNJ (Constraint Nets in Java), is established to support constraint net modeling and simulation. CNJ uses component-based technology (JavaBeans), Bean Introspection and Java Swing, as well as XML as the standard interchange format.

1.1 Thesis Statement

CN is a unified and integrated approach for modeling, specifying and verifying discrete, continuous, and event-based hybrid dynamic systems. CNJ aims to provide a practical, feasible environment for constraint net modeling and simulation. Although one of its purposes is to provide such a realistic environment, other more important research objectives include the following:

- Investigation of component-based technology in hybrid system modeling and simulation.
- Proof that visual programming environments and visual programming languages (VPL) are helpful for users.
- Demonstration that Java is useful for virtual real-time programming.

- Feasibility of the efficient and useful implementation of the concept of Constraint Nets.
- Demonstration of an open, portable, executable representation of hybrid dynamic systems.

1.2 Motivation

In the past decade, hybrid systems have become a focus of interest for a wide community for two reasons. One is that analog computation is gaining more attention because of recent technological advances, such as faster computers, and cheaper, more reliable sensors. The other is that the integration of computers that control continuous dynamic systems shows increasing importance [7]. Constraint Nets are developed as a unified formal foundation for hybrid dynamic systems, consisting of modeling, and the specification and verification of languages.

CN is a generalization of dataflow with multiple data and events, in which a dynamic control system can be described as several block diagrams. A constraint net represents a set of equations, with locations as variables, and transductions as functions. It is able to support various time structures: discrete, continuous and event-based. Since it has a graphical representation, and is composed of a set of basic graphical elements, CN is an ideal model for visual programming.

Java is an object-oriented programming language where a Java class is reusable. The JavaBeans technology has the important properties of reusability, bean introspection, and platform-independence. Using the technology of JavaBeans, Java classes are able to be reused, and can run anywhere. It facilitates greater reusage and allows for rapid development through assembling software components rather than writing codes.

The goal of this thesis is to design and implement a visual programming environment for hybrid system modeling and simulation in the constraint net language. The environment uses JavaBeans as atomic blocks to build hierarchical, modular and large-scale system models [35]. In CNJ, several kinds of Java bean classes are implemented and provided as elementary blocks. The tool uses Java's event handling mechanism (registration of event listeners) to connect Java beans, and make them communicate with each other. The Java event mechanism successfully handles the support for various time structures. This JavaBeans-based visual programming environment has the advantages of the reusability of modules, reduction of coding time, and simplification of model development. Since CNJ is a visual programming tool, users only need to drag and drop the provided Java beans, and assemble them together. To store constraint net models, we also define XML-based CNML (Constraint Net Markup Language) as the standard interchange file format. CNML conforms to the specification of XML (eXtensible Markup Language) 1.0.

1.3 Related Work

It has been eight years since CN was first introduced and developed. It works as a formal analysis and modeling framework for the development of hybrid systems. CN proves to be useful for a variety of purposes, ranging from intelligent systems, and real-time embedded systems, to integrated hybrid systems [7]. However, there is no appropriate programming environment available for it. CNJ is the first customized visual programming environment for CN, based on component-based technology.

Nevertheless, research in hybrid system modeling and simulation is not completely new. Many papers are published in this area and several systems are developed in industry. However, most of them focus on some specific time structures, instead of a variety of them, such as discrete, continuous, and event-based.

Matlab/Simulink [3] is a visual programming and simulation environment for continuous and discrete control systems. It enables users to build graphical block diagrams, simulate dynamic systems, evaluate system performance, and refine their designs. It is currently the most popular tool for control system modeling and simulation. However, it is not suited for hybrid system modeling in constraint nets. There are three reasons for this:

- First, Simulink is unable to support an event-based time structure, which is an important characteristic of hybrid systems.

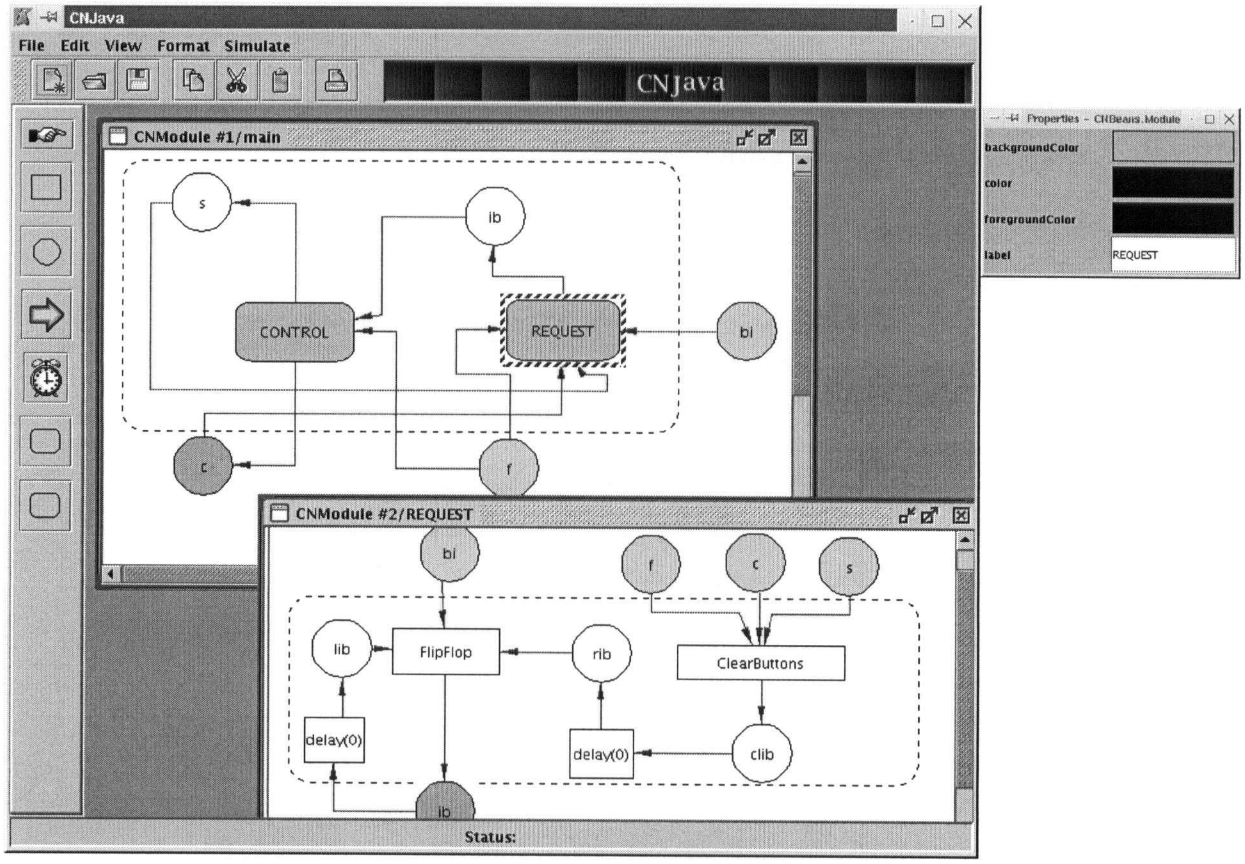
- Second, although it supports bottom-up modeling well (by grouping), it does not support top-down and middle-out modeling methods, which are helpful for some users [3].
- Third, in Simulink, all the system models are stored in MDL format (Model Description Language). In addition, since CN has a different graphical representation from Simulink's models, the MDL file format is not able to store constraint net models.

There is also some component-based simulation work done. [33] [28] and [17] use Sun's BDK (Bean Development Kit) tool directly to model and simulate control systems. BDK is a simple tool for testing Java beans, and for visually manipulating their properties and events. It is convenient to use the BDK to assemble some beans to build a system model or module, through drag-and-drop. However, since BDK is not aimed at system modeling, it actually imposes some limits on the use of those systems. The graphical user interface of BDK is weak because it is implemented in Java AWT (Abstract Window Toolkit), not Java Swing (which is much more powerful). For instance, in BDK, there are only a small number of visual components which display bean properties because of its limited internal support. The wires that connect between beans are invisible. Also, there are only Java-level events in the modeled systems, with no concept of a continuous time structure, which is, however, essential for complex hybrid systems. Furthermore, it is challenging work to build a complex model based on customized JavaBeans. Users might be required to implement a very complicated Java bean as an atomic block. All the limits above make those BDK-based modeling environments difficult to use practically. In addition, it is even harder to apply it to constraint net modeling.

Visual programming languages (VPL) have been researched for dozens of years, and numerous visual programming systems have been developed to address specific application areas, physical simulation, and more general programming tasks. VPLs offer many advantages over traditional textual programming languages, such as fewer programming concepts, concreteness, explicit relation depiction, and visual feedback [8]. Some popular visual programming languages include Alternate Reality Kit (ARK), Visual Imperative PRogramming (VIPR), Cube, and so forth [13]. With the increase in computer speed and graphical display capabilities, it is highly probable that visual programming languages will be widely used.

1.4 CNJ Overview

Our environment is called CNJ (Constraint Nets in Java). It is a specific environment for hybrid system modeling and simulation with constraint nets. CN is a family of visual



backgroundColor	
color	
foregroundColor	
label	REQUEST

Figure 1.1. The Graphical User Interface of CNJ

programming languages, which consist of timed ∀-automata as a requirement specification language, a dynamical system modeling language, and a verification method. CNJ's system design includes support for the requirement specification language, the dynamical modeling language and the verification method. The present version provides visual programming in the dynamical modeling language, as well as real-time simulation.

A constraint net is composed of Locations, Transductions, and Connections. CNJ provides several basic Java beans, accordingly. When designing a constraint net model, users need to draw and define transductions, locations, modules, and clocks (a kind of transduction), and then wire these by connections. Because it is a visual programming environment, CNJ is easy to use and learn.

CNJ is implemented in pure Java, and can run on different platforms. Its graphical user interface consists of two main windows, shown in Figure 1.1. The big window is the CNFrame, used for designing or drawing a constraint net model. The small one is the PropertyEdit window, which is able to edit a selected object's properties in the left window. When users select a different object, the PropertyEdit window changes accordingly. The panel at the center of the CNFrame adopts the style of an MDI (Multiple Document Interface) with child windows. Each child window has a DrawPane for displaying a constraint net module.

CNJ supports various modeling and development methods: top-down, bottom-up, and middle-out. With these modeling methods, user can group system models into hierarchies to create a simplified view of subsystems or modules. To design a constraint net model, the simple drag-and-drop method is used. In constraint nets, the function of a transduction might be arbitrary. CNJ provides predefined basic operators for building complicated functions. If an unsupported function is needed, CNJ developers can easily add it. This tool also provides a compilation step to detect some syntax errors in user designed constraint nets. After passing the compilation step, users might run a simulation to see the output data. Users can also run an animation window to see a more straightforward running result.

A constraint net model, or module, is stored in a Constraint Net Markup Language (CNML). When users need a module for further reuse, they can save it in a separate file. The CNML is based on the specification of XML 1.0, and brings XML's advantages: being content-oriented, giving easy access and reusage, providing validation support, and so forth. Presently CNJ can export a constraint net model to a postscript file.

1.5 Thesis Outline

This chapter provides an introduction to the thesis, including the thesis statement, motivation, contributions and overview of CNJ. Chapter 2 introduces some background knowledge related to our research. In it, Component-Based Software Engineering (JavaBeans), HCI, and VPL are briefly mentioned. Chapter 3 introduces the principles of constraint nets. In Chapter 4, we present our main idea for designing CNJ and its system architecture. Chapter 5 describes the implementation of CNJ's graphical user interface and the CN nodes in Java bean. Chapter 6 illustrates several mechanisms for supporting real-time simulation in CNJ. In Chapter 7, Constraint Net Markup Language is introduced. In Chapter 8, a hybrid Elevator control system is modeled in constraint nets, and developed in CNJ. Chapter 9 presents some experimental results of the environment. Finally, conclusions and future work are described in Chapter 10. Appendix A gives a complete CNML file for the Car Dynamics CN model. Appendix B contains the set of graphical CN modules for the 3-floor Elevator system designed in CNJ.

1.6 Contributions

A hybrid system in general, is a continuous, discrete, and event-based dynamic system. In the past decades, models for hybrid systems developed and matured. Some design environments for these were also created [5]. Matlab/Simulink was developed for continuous and discrete models [3]. DevsJava was developed for Discrete Event System Specification (DEVS) [16]. PNK was developed for Petri Nets [34]. The Automated Control Engine (ACE) was developed for event-based control systems [9]. CN is an ideal model for hybrid systems, and it is unitary, modular, and powerful. It is successfully applied to robotic system modeling. A robotic system is a typical hybrid system. Its controller is modeled in discrete dynamics, its body and environment are modeled in continuous dynamics. With the support of various time structures, CNJ is able to model such hybrid systems in constraint nets. Component-based technology is used to realize the visual programming environment. The simulation part is almost in real-time, even though it is written in Java.

This thesis presents our work of CNJ for modeling and simulating real-time hybrid systems with constraint nets. The primary contributions of our work are as follows:

- It proves that component-based technology works well for control system modeling and simulation.
- It shows visual programming is helpful for users.
- It demonstrates that Java is useful for real-time programming.
- It proves that Constraint Nets are a practical, feasible language for modeling hybrid systems.
- It is helpful for different user groups to use CNJ to study Constraint Nets.
- CNML is defined as the first standard XML-based interchange file format for constraint nets.

CHAPTER 2 BACKGROUND

CNJ is designed and implemented using component-based technology. It works as a visual programming environment. This chapter illustrates the concepts of component-based technology, JavaBeans, user interface design, and visual programming languages (VPL).

2.1 Component-Based Software Engineering

The idea of software components has been developed in software engineering for some time. A software component is defined as "a unit of composition with contractually specified interfaces and explicit context dependencies only" [20]. In this definition, components are independent of each other, and communicate with each other through predefined interfaces. A software component is an independent object that has its own functionality. Through interfaces, a large-scale and complicated system can be built on the available components.

Component-Based Software Engineering (CBSE) is based on software components. It is a process of building systems by combining and assembling predefined objects or components. CBSE is able to guarantee the development of software or systems in less time, with higher quality and reliability. The characteristics of component-based development are as follows [15]:

- Black-box reusage
- Reactive-control and the component's granularity
- Using RAD (rapid application development) tools
- Contractually specified interfaces
- Introspection mechanism provided by the builder systems
- Software component market

In addition, component-based development is closely related to object-oriented programming. It is natural and straightforward to implement software components in an object-oriented language. The traditional techniques in object-oriented software engineering are important for component-based development, such as design pattern, architecture pattern, and meta-pattern. However, component-based development is different from object-oriented development. A software component is not a part of the application or program, but is an autonomous entity. Programmers cannot see its code, except for the interface specification [18]. Typically, it is in the form of an object instance instead of source code. Furthermore, component-based development does not need to use the class hierarchy that is very important in object-oriented programming.

Currently, software components can be developed with ActiveX, CORBA, JavaBeans and EJB (Enterprise JavaBeans). We chose JavaBeans to develop CNJ because it has the advantages of platform-independence, object-orientation, simple strategy for multi-threading, automatic garbage collection, safe memory usage, and easy programming. Although a Java program's speed is not as fast as that of a C++ program, after the utilization of Just-In-Time (JIT) and native code compilers, this problem of speed decreases.

2.2 JavaBeans Component Architecture

Java bean is defined as "a reusable software component that can be manipulated visually in a builder tool" [14]. The JavaBeans component architecture is a platform-neutral architecture for the Java application environment. It is used to develop fully portable network-aware applications within Intranets, or across the Internet. Actually, when using Java, JavaBeans component architecture is the only choice to consider in component-based development.

A Java bean class is able to run on various operating systems, and also within many application environments. As [37] stated, "A JavaBeans developer secures a future in the emerging network software market without losing customers that use proprietary platforms, because JavaBeans components interoperate with ActiveX". Through

JavaBeans Architecture Bridges, JavaBeans can be connected together with other component models, such as ActiveX. Therefore, software components that use JavaBeans APIs are portable to containers such as Internet Explorer, Visual Basic, Microsoft Word, and Lotus Notes.

A Java bean is a Java class. When obeying a few non-critical naming conventions, or using a BeanInfo class, a general Java class becomes a Java bean. A bean's interface includes its *properties*, *methods*, and *event*s. A Java bean has the following typical features [14]:

- Support for "properties", both for customization and for programming use. Properties are the named attributes of a bean. They can be accessed (read and write) by some appropriate methods which obey some rules. The method has to be named as get<Property> or set<Property>. There are four kinds of bean property: *Simple Property*, *Bound Property*, *Constrained Property*, and *Indexed Property* [30]. The Bound Property has the capability that, when the value of the property changes, it notifies the other objects that added a PropertyChangeListener for that property. A Constrained Property handles how a proposed property change can be permitted or vetoed by other objects. An Indexed Property describes a multiple-value property.
- Support for "events" as a simple communication method that can be used to connect up beans. With an event, a bean can notify other beans that something interesting has happened. Also, other beans can do something to respond to the events. For that purpose, event listeners are registered to event sources. The event source provides two methods to support event listener registration: add<Event>Listener and remove<Event>Listener, where <Event> is the name of the event.
- Support for "Introspection" so that a builder tool can analyze how a bean works. The builder tool can detect information of a bean, such as properties, methods, and events if the bean class conforms to naming conventions of JavaBeans.
- Support for "Persistence". A bean can be customized in an application builder, and then have its customized state saved away and reloaded later.
- Support for "Customization". When using an application builder, a user can customize the appearance and behavior of a bean. Sometimes the modification of

simple properties is not sufficient for complex components. Additional editors or customizers may be provided for the ease of customization.

2.3 Graphical User Interface Design

User interface is a bridge between users and computers by allowing for information exchange between them. There are a vast variety of ways to bridge humans and computers, for instance, display areas, digital cameras, full-duplex audio, actuators and sensors, and more recently, even scent through aroma generating machines and artificial noses. With increased user interface sophistication, this, in turn, makes the programming task more difficult. Generally, user interface design and programming occupies a large proportion of software development. From the survey [2], the average time spent on the user interface portion is 45% during the design phase, 50% during the implementation phase, and 37% during the maintenance phase. The most common difficulties that UI programmers confront include getting the user's requirements, writing help text, achieving consistency, getting acceptable performance, and communicating among various parts of the program. In order to decrease the UI programming task, some interface builders and UIMSs (User Interface Management System) are developed.

2.3.1 UI Design Guidelines

A good user interface should be easy to use, and suitable for a specific operating system. However, this is not easy for software developers to realize. In addition, good UI designs do not happen naturally. These require software developers to learn and apply a few basic guidelines. The following items are provided as basic guidelines for designing a user interface [12]:

1. **Maintain Consistency in Look and Feel.** It requires consistent visual appearance and consistent response to user input throughout the user interface design.
2. **Provide Shortcuts and Flexibility.** Some experienced users need the means of going directly to specific locations in UI.

3. **Present Informative Feedback.** Feedback gives users confidence. It tells the user what the program is doing.
4. **Design for Recovery from Error.** If possible, the undo function and meaningful error messages are provided to users.
5. **Reduce Memory Demands.** Too many facts and decisions might overload a person's short-term memory.
6. **Design for Task Relevance.** UI should present information that pertains to the user's task. Any arrangement of items on the screen is task-related.
7. **Aid Orientation and Navigation.**
8. **Maintain a User-Centered Perspective.** Every element of the design should be traceable to the user requirements.

The guidelines above are at a relatively high level. However, a project does not need to include all of them. It can just select the guidelines that are meaningful in the context of its own user interface design.

2.3.2 Java Look and Feel Style

As the Java platform matured, developers recognized the need for consistent, compatible, and easy-to-use Java applications. The Java look and feel provides a platform-independent appearance and standard behavior to meet this need. It can reduce design and development time, and lower training and documentation costs.

The Java look and feel is the default interface for applications built with the Java Foundation Classes (JFC). It has the following characteristics [29]:

- Consistency in the appearance and behavior of common design elements.
- Compatibility with industry-standard components and interaction styles.
- Aesthetic appeal that does not distract from application content

Developers have the choice to choose a look and feel style. They can determine a platform-dependent look and feel, or a cross-platform look and feel. If they specify a cross-platform look and feel, applications appear and perform the same everywhere, simplifying the application's development and documentation. If choosing a particular look and feel, developers can specify an OS-dependent look and feel style. The look and feel styles available in Java 2 SDK as follows:

- **Java look and feel.** (Called "Metal" in the code) It is designed for use on any platform that supports the JFC.
- **Microsoft Windows look and feel.** (Called "Windows" in the code) It can be used only on Microsoft Windows platforms. It follows the behavior of the components in applications that ship with Windows NT 4.0.
- **CDE look and feel.** (Called "CDE/Motif" in the code) It is designed for use on UNIX platforms. It emulates OSF/Motif 1.2.5.
- **Macintosh look and feel.** Developers can download the Macintosh style (called "Mac OS" in the code) separately.

2.4 Visual Programming Languages

Visual Programming Languages (VPLs) are a combination of computer graphics, programming languages, and human computer interaction. With the increase in computer speed and graphical display capability, a great deal of research and experiments are now possible in the field of visual programming languages. A variety of different methodologies originate from this research field, and numerous software systems are developed for both specific application tasks and more general tasks.

2.4.1 Classification and Concepts

In [13], visual programming languages are classified as:

- Purely visual language
- Hybrid text and visual system
- Programming-by-example system

- Constraint-oriented system
- Form-based system

These categories are not exclusive and a programming language might belong to more than one category.

The category of purely visual language is characterized by its reliance on visual techniques throughout the programming process. The programmer manipulates icons or other graphical representations to create a program, and then debugs and executes it in the same visual environment. The program is compiled directly from its visual representation, and is never translated into an interim text-based language [13]. This category can be further subdivided into sections like iconic and non-iconic languages, object-oriented, functional, and imperative languages.

Hybrid text and visual language involves both visual and textual elements. The hybrid systems include programs that are created visually and then translated into a textual language, as well as those utilizing graphical elements in a textual language. Programming-by-example systems provide users with a way to create and design graphical objects to "teach" the system how to perform a particular task. A popular approach for simulation design is the constraint-oriented system. Physical objects can be modeled in the visual environment, subject to a set of constraints, to simulate natural behaviors. These systems have also found applications in the development of graphical user interfaces. The form-based VPLs refer to those borrowing their visualization and programming metaphors from spreadsheets [13].

Some important basic concepts in VPLs are defined by [1], as follows:

Icon: An object with the dual representation of a logical part (meaning) and a physical part (image).

Icon sentence: A spatial arrangement of icons from an iconic system.

Visual language: A set of iconic sentences constructed with given syntax and semantics.

Syntactic analysis: An analysis of an iconic sentence to determine the underlying structure.

Semantic analysis: An analysis of an iconic sentence to determine the underlying meaning.

2.4.2 Automated Control Engine

The Automated Control Engine (ACE) is a software package from International Submarine Engineering Ltd., B.C., Canada. It is designed to help design event-driven, and dataflow, control systems. The ACE is also an implementation of an object-oriented *Events and Actions* programming paradigm using the C++ language [9].

Events and Actions is an object-oriented event-driven programming paradigm. It is based upon two object types: event and action. Event represents some significant occurrence or occurrence with some related data, such as timer interruption, or message from other components. Action is an appropriate response to a particular event. Typically, an action consists of a procedure for execution, a priority level for the procedure, and a reference to any data belonging to the event. The action may signal events after the completion of its task. Component is another key element in ACE. It is a set of connections to input events and output events with a well-defined transformation between them.

ACE programming is therefore, a collection of events and actions, and the connections between them. This type of programming provides a high level of design, which does not require any C++ knowledge. It is a natural approach to real-time or highly interactive systems [9].

2.5 Summary

To develop a visual programming environment for constraint nets, many issues in this chapter must be considered. It is helpful to provide this research's context. For instance, what are visual programming languages, how to design a good user interface for such an environment, which software engineering method is most suited for this application, how to use JavaBeans to develop a component-based software, what is JavaBeans, as well as event-driven programming, and so forth.

CHAPTER 3 CONSTRAINT NETS

Over the last twenty-five years, the Constraint Satisfaction Problem (CSP) model developed and matured. Constraint Programming (CP) also evolved several powerful frameworks. The algorithms developed for the Constraint Satisfaction Problem are made more useful and available when they are incorporated into the Constraint Programming (CP) paradigm [23]. This is successful, and as a result, some constraint-satisfying devices are developed. Generally, these devices are offline. A challenge for CP researchers is to develop a theoretical and practical tool for the constraint-based embedded intelligent system. The Constraint Nets model (CN) is developed by Ying Zhang and Alan Mackworth as an abstraction and a unitary framework for developing a hybrid dynamic system, analyzing its behavior, and understanding its underlying physics.

This chapter first describes the definition of constraint nets. Then the requirement specification language of timed ∀-automata, and the behavior verification method are described.

3.1 Constraint Net Model

Intelligent systems embedded as controllers in real systems, or virtual systems, are designed in an online model based on various time structures: discrete, continuous, and event-based. It is a typical hybrid dynamic system. However, the CSP paradigm and CP paradigm are inadequate for this kind of task since they are primarily off-line. Generally, CN is a semantic model for those hybrid dynamic systems. A constraint net model is built as an online dataflow-like distributed programming language with formal algebraic denotational semantics, a requirement specification language, and real-time temporal logic [25].

3.1.1 Constraint-Based Intelligent System

An intelligent agent system is normally a hybrid system that runs on the domains of discrete, continuous and event-driven structures. It is modeled as the symmetrical coupling of an agent with its environment. An agent situated in an environment has three machines: the agent controller, the agent body, and the environment, as shown in Figure 3.1 [23].

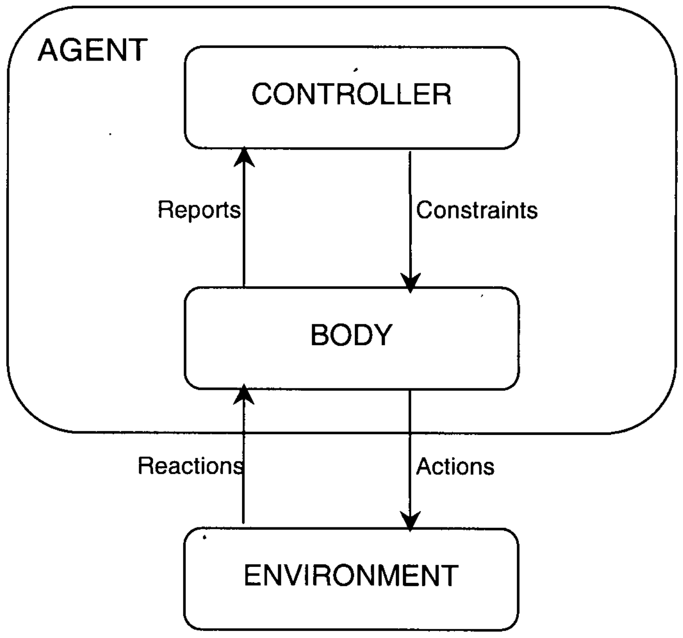


Figure 3.1. The structure of a constraint-based agent

The controller, body and environment are modeled as separate dynamic systems. Both the controller and the body consist of discrete, continuous or event-driven components operating over a discrete or continuous domain. The controller has the perceptual subsystem that can (partially) observe the state of the body, and through the body, partially observe the state of environment. The agent's body is the interface to the environment. The body executes actions in the environment, senses the change of the environment, and reports the environment's state to the controller. Whenever the controller gets the report about the environment's state from the body, it calculates the constraints and sends them to the body. Then, the body performs the actions obeying the constraints.

3.1.2 Constraint Net Syntax and Semantics

A constraint net model is a dataflow and equation based model with formal syntax and semantics. It consists of a finite set of locations, a finite set of transductions, and a finite set of connections. Locations can be regarded as communication channels, memory cells, variables, or wires. Transductions are causal mappings from input traces to output traces, operating on a global reference time, or activated by external events [7]. Connections represent interaction structures of the modeled system by relating locations with ports of transduction. Formally, a constraint net is a triple CN = <Lc, Td, Cn>, where Lc is a finite set of locations, Td is a set of transductions, each with an output port and a set of input ports, and Cn is a set of connections between locations and ports [6, 10, 25]. A Connection has to follow some restrictions: one, there is at most one output port connected to each location; two, each port of a transduction connects to a unique location; and three, no location is isolated.

A location is an *output* of the constraint net if it is connected to the output of some transductions; otherwise, it is an *input.* A constraint net is open if there is an input location; otherwise, it is *closed.*

A constraint net represents a set of equations, with locations as variables and transductions as functions. The semantics of the constraint net, with each location denoting a trace, is the least solution of the set of equations [4]. The semantics is defined on abstract data types and abstract reference time, which can be discrete or continuous. Graphically, a constraint net is depicted by a bipartite graph where locations are depicted by circles, transductions by rectangle blocks, and connections by arcs.

For instance, integration is the basic transduction on a continuous time structure. In Figure 3.2, a differential equation $ds / dt = f(s) = s'$ is represented by a constraint net, where s_0 is an initial state.

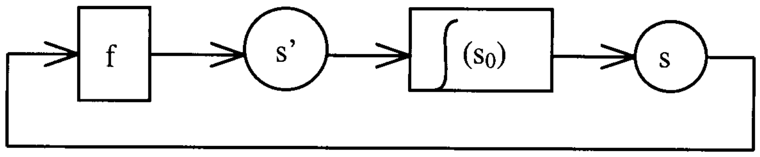


Figure 3.2. An integration system in constraint net

3.2 Requirement Specification with Timed ∀-Automata

CN graphical modeling language focuses on the underlying structure of a system, such as the organization and coordination of components or subsystems. However, the overall behavior of the modeled system is not explicitly expressed. For many cases, it is important to specify some global properties, and guarantee that these properties hold in the proposed design.

Zhang and Mackworth advocate timed ∀-automata for specifying requirement properties. A discrete ∀-automaton is a non-deterministic finite state automaton over infinite sequences [6]. It was originally proposed as a formalism for the specification and verification of temporal properties of concurrent programs. Then, discrete ∀-automata were augmented to timed ∀-automata by generalizing time from discrete to continuous, and by specifying time constraints on automaton-states. Therefore, timed ∀-automata can be used to specify languages composed of traces on continuous, as well as discrete, time.

A ∀-automaton *A* is a quintuple {Q, R, S, e, c}, where Q is a finite set of automaton-states, $R \subseteq S$ is a set of recurrent states, and $S \subseteq Q$ is a set of stable states. Let *r* be a run of *A* over a trace *v*. *r* is accepting if every state that appears infinitely many times is a *stable state*, or if some state that appears infinitely many times is a *recurrent state* [6]. With each $q \in Q$, an assertion e(q) is associated, which characterizes the entry condition under which the automaton may start its activity in q. With each pair $q, q' \in Q$, an assertion c(q, q') is associated, which characterizes the transition condition under which the automaton may move from q to q'. R and S are the generalization of accepting states to the case of infinite inputs. $B = Q - (R \cup S)$ is the set of non-accepting (bad) states.

A ∀-automaton can be depicted by a labeled directed graph where automaton-states are depicted by nodes, and transition relations by arcs. Furthermore, some automaton-states are marked by a small arrow, an entry arc, pointing to it. Each recurrent state is depicted by a diamond inscribed within a circle. Each stable state is depicted by a square inscribed within a circle. Nodes and arcs are labeled by assertions. A node or an arc that is left unlabeled is considered to be labeled with true. The labels define the entry conditions and the transition conditions of the associated automaton.

Some examples of ∀-automaton are shown in Figure 3.3 [6]: (a) states that the system should finally satisfy G; (b) states that the system should never satisfy B and (c) states that whenever the system satisfies R, it satisfies S in some bounded time.

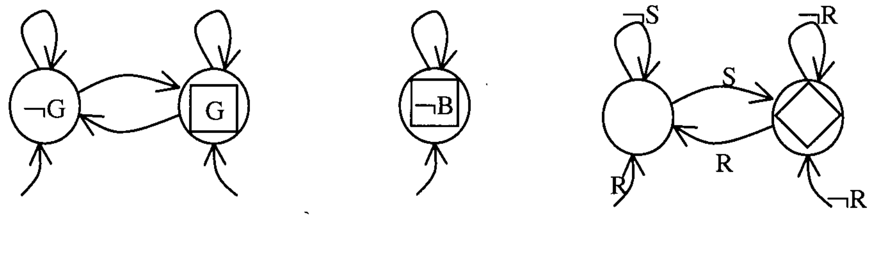


Figure 3.3. ∀-automata: (a) reachability (b) safety (c) bounded response

Timed ∀-automata are ∀-automata augmented with timed automaton-states and time bounds. Let R^+ be the set of non-negative real numbers. A timed ∀-automaton *TA* is a triple {A, T, τ }, where *A* = {Q, R, S, e, c} is a ∀-automaton, T ⊆ Q is a set of timed automaton-states, and $\tau : T \cup \{bad\} \rightarrow R^+ \cup \{\infty\}$ is a time function. A ∀-automaton is a special timed ∀-automaton with T = ∅ and τ (bad) = ∞. Graphically, a T-state is denoted by a nonnegative real number indicating its time bound. Figure 3.4 shows an example of timed ∀-automata. This system should satisfy S within 3 time units, whenever it satisfies R.

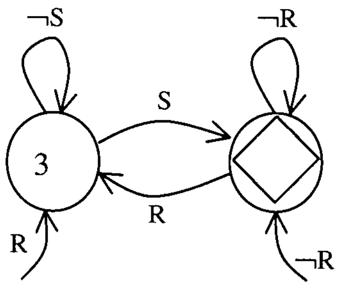


Figure 3.4. Timed ∀-automata: real-time response

3.3 Behavior Verification

In constraint nets, timed ∀-automata in Section 3.2 work as a requirement specification language. It can specify the required properties for the designed models. To decide whether the model is well-designed or ill-designed, a model checking method is developed to verify the relationship between the behavior of a dynamic system and its requirement specification. Given a constraint net model of a system and a timed ∀-automaton specification of a behavior, the behavior of the system satisfies the requirement specification if and only if the (behavior) traces of the system are accepting for the timed ∀-automaton.

The model checking method developed by Zhang and Mackworth uses the induction principle, and generalizes both Liapunov stability analysis for dynamic systems and monotonicity of well-foundedness in discrete-event systems [21]. A representation between constraint nets and timed ∀-automata is a state-based transition system, such as Kripke structure. The verification rules are applied to the Kripke structure.

A useful and important type of behavior is state-based and time-invariant. A state-based and time-invariant behavior is a behavior whose traces, after any time, are totally dependent on the current snapshot. State-based and time-invariant behaviors can be defined using the generalized Kripke structure.

The formal method for behavior verification consists of a set of model-checking rules, which is a generalization of the model-checking rules developed for concurrent programs. There are three types of rules: invariant rules (I), stability or eventuality rules (L) and timelines rules (T) [21].

3.4 Summary

In Constraint Nets, timed ∀-automata are simple to use for requirement specifications. However, they are not powerful enough to represent all possible behaviors. The verification method provides a set of formal model checking rules, which can be used to guide a formal proof procedure. However, the invariants, Liapunov, and timing functions are not automatically created, and the verification of the rules is not automatic, in general. Nevertheless, the CN approach can be used to solve many problems in hybrid system modeling, designing, specification, and verification.

This chapter does not attempt a comprehensive illustration of constraint nets. It endeavors to provide basic CN principles for CNJ, and make the thesis self-contained. While the system design of CNJ includes the modeling language, the requirement specification language as well as the verification method, until now, CNJ has only provided the support for the modeling language. We propose in future work, adding the specification and verification languages to the CNJ tool based on the existing CNJ's architecture since our design allows for it.

CHAPTER 4 CONCEPTS AND ARCHITECTURE OF CNJ

In CNJ, there are predefined icons provided for users to choose, drag and drop, and connect together. CNJ supports modularity and hierarchy in CN programming. After completing the model design, users can compile it, simulate it, and see the animation effect. This chapter describes the design idea, requirement analysis and implementation mechanism of CNJ, as well as the class architecture implemented in CNJ.

4.1 Main Idea

CN is a family of visual programming languages. A constraint net consists of locations, transductions, and connections, where locations are depicted by circles, transductions are depicted by rectangles, and connections are depicted by arcs. When users write programs in constraint net language, intuitively, the simplest method is to allow them to drag and drop the predefined nodes, then customize their functions, size, and appearance. Finally, the designed or drawn program is able to run.

In CNJ's framework, the realization of such a visual programming environment is based on a few ideas. There are three specific types of nodes for composing a constraint net graph. From the viewpoint of component-based software engineering, each type of node may be implemented as a software component, which is easier for users and program developers. In Java, JavaBeans is the only candidate to support software components. We plan to provide a set of elementary, yet powerful building blocks for constraint net modeling and simulation. The building blocks are implemented with JavaBeans. Each constraint net node is a Java bean, and is added to the CNJ environment, then customized.

To allow for adding a Java bean, the CNJ environment is required to support the Bean Introspection mechanism. There should be a library of utilities and support objects provided to identify the properties, events, and methods that a Java bean supports. With it, the environment can customize the constraint net beans in their functionality in order to meet programming requirements.

The communication between Java beans uses Java's event mechanism. Locations and ports of transductions are wired by connections. The Java event mechanism allows Java beans to act as event sources for event notifications that can be caught and processed by some other listeners. The designed constraint net is like a dataflow graph where data are relayed and processed from input locations to output locations. Constraint net modules are coupled by *interface locations*. If a module's *interface location* has the same name as one of this module's outside connected locations, the two locations are actually connected together transparently, and communicate with each other (see Figure 4.1).

There are several different methods for modeling a hybrid dynamic system, such as bottom-up, top-down, and middle-out. In CN, these modeling methods are generally realized by hierarchical composition and the coupling of modules through input and output *interface locations*.

In fact, a transduction can probably be specified as an arbitrary function. It is certainly impossible to provide all of the arbitrary functions. A few basic functions are provided in CNJ, such as +, -, *, /, *or*, <, >, *sine*, *if...then...else...*, *delay*, ... Based on these, complicated functions for particular complex systems are able to be compositionally represented.

After users complete designing a constraint net, the constraint net is stored away. Unlike JavaBean's *serialization,* where a bean is converted into data stream and written into storage, an XML-based file format is used to save a constraint net. We defined *Constraint Net Markup Language* (CNML) as CNJ's standard interchange file format. CNML provides CNJ with a lot of advantages that XML inherently owns.

Figure 4.1 illustrates how to put together a system model by bean objects. Using this method, a system model can be built with much less effort by simply picking, zooming, customizing, and coupling components from libraries. The locations labeled with "loc" are modules' interface locations, with which the two modules are connected with each other.

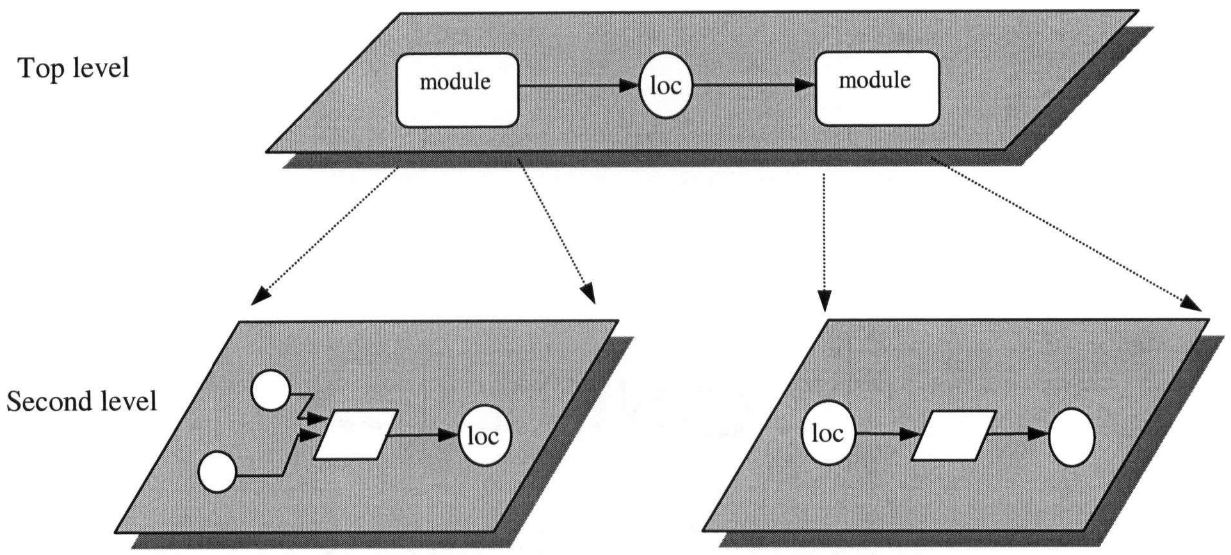


Figure 4.1. Putting together a system from bean objects

4.2 Requirement Analysis

To work as a visual programming environment, CNJ should be able to support users to 'draw' a program in constraint nets. When programming visually, users choose *locations*, *transductions*, *connections*, and *modules* to put together a constraint net. To mimic the realistic manual CN design, a component-based modeling and programming framework is the most convenient and beneficial method for building CNJ.

4.2.1 System Requirements

There are several requirements for this kind of modeling and simulation environment:

- First, it should enable developers to interactively pick components, and place them onto a work area. These components are CN's atomic nodes: *locations*, *transductions*, *connections*, and *modules*.

- Second, connecting has to be accomplished in a way where events and data can be exchanged correctly among the components.
- Third, the convenient interactive customization of bean properties should be supported by the environment.
- Fourth, there has to be a method to check each CN node's dynamic values. For instance, users might wish to see a *location*'s changing values while a simulation is running.
- Finally, such a modeling and simulation environment has to be simple to use and execute. Also the designed model should be reusable as a new component in any other hybrid system. In this case, a very complicated system can be built by assembling some less complicated components.

During our system design period, we hoped to make CNJ run on the Internet as a Java applet, for it is very helpful and important for people to learn and study constraint nets. However, Java applets have a few limits to running on the Internet for several reasons. For instance, an applet runs very slowly under severe network restrictions, it cannot read or write to the local disk, and it has less rich GUI components than a Java application. Therefore, we chose to implement CNJ as a Java application first. Moreover, the implementation of CNJ in Java brings some advantages, such as platform portability, simple implementation, robustness, reliability, and so forth.

In such a visual programming environment, users 'draw' constraint net programs, instead of writing code for them. The look and feel is intended to resemble the style of some popular drawing tools such as Adobe Illustrator, MS Painter, and Unix xfig to support constraint net designing. In addition, to make the GUI respond as quickly as possible, we adopt multi-threaded programming to minimize the response time to users' action. In Constraint Nets, a model possibly consists of dozens of modules that are hierarchically located in different levels. To support as many modules as possible, CNJ uses Multiple Document Interface (MDI) to display each module in a child window. Thus, every module component corresponds to a child window in the MDI desktop.

4.2.2 User Groups

CNJ is supposed to be used by three different types of users in three different ways. They are the *CNJ Users*, *CNJ Experts* and *CNJ Programmers*. A *CNJ user* uses a set of predefined, ready-to-use Java beans (*locations*, *transductions*, *modules*, and *clocks*) to model a hybrid dynamic system. After picking the beans, he or she can customize them by setting their properties in a dynamic interactive window (PropertyEdit Window), then connect them together, and run simulations with them. He or she needs to know nothing about Java language except for constraint nets. There should be a simple installation and easy-to-use configuration for a CNJ user in different platforms. In most cases, this type of user only needs to know how to design a constraint net. We consider *CNJ users* as the main client. Also, they profit most from the component-based modeling and simulation environment. It should be less effort for them to model an application-specific control system.

Although CNJ provides all the necessary building blocks for constraint nets, they are still not sufficient to build an arbitrary hybrid system because a transduction bean is not powerful enough to support an arbitrarily complex function. There is a set of basic function options for transduction beans. In most cases, a complex function can be built by those basic transductions. However, occasionally, when building a complex function, a user might find that there is an indispensable basic function not yet provided by CNJ. In this situation, another type of user, *CNJ experts*, is needed. Besides the capabilities a *CNJ user* has, a *CNJ expert* also knows how to write programs in Java in order to implement new functions without which a necessary function cannot be built. A CNJ expert heavily relies on Java development tools to implement new functions. However, writing Java code to add new functions is not too troublesome. Based on explicit guidelines, he or she can realize this by modifying a few codes in several specific places.

Although the task of a *CNJ expert* is challenging, it is still not the same as that of a *CNJ programmer*. A *CNJ programmer*'s task is the most demanding. He or she needs a good understanding of CNJ's source code, detailed structure, modeling and simulation concepts, Java Beans, CNML, as well as the maximization of components' reusability.

4.3 System Architecture

CNJ is a hybrid system modeling and simulation environment. With this tool, users can model or render a control system visually in constraint nets (CNRender). Also CNJ is able to run or execute the designed models that are driven by Clocks (CNRun). When there is an animation package connected to that control system, the system's running result can be shown explicitly, and also, users can set or change *input locations*' values interactively through the animation window. The architecture of CNJ is shown in Figure 4.2.

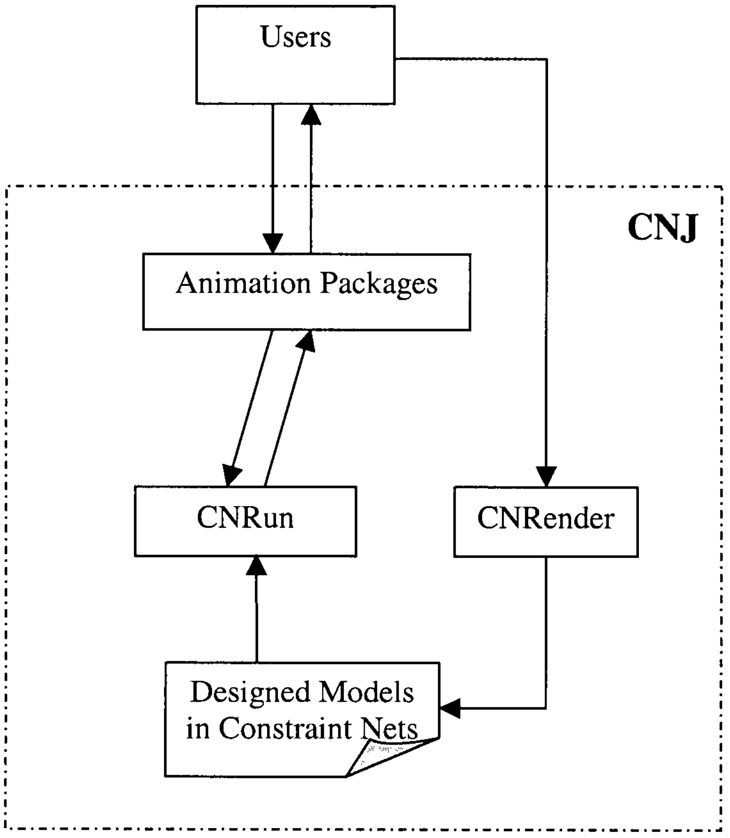


Figure 4.2. Architecture of CNJ

The graphical user interface is an important part, and has the same style that an interactive drawing tool has. It has two main windows: CNFrame and PropertyEdit, as shown in Figure 1.1 in Chapter 1. The CNFrame window is the bigger one. It consists of MenuBar, ToolBar, ToolPane, and DrawPane. All the necessary CN nodes are contained in the ToolPane panel (the left column) where there are the *Connection* bean, *Location* bean, *Transduction* bean, *Clock* bean, and *Module* bean. DrawPane is the center area for drawing constraint nets and has a MDI (Multiple Document Interface) for designing different hierarchies of modules. PropertyEdit window is the smaller one lying to the right, close to the CNFrame, and displays corresponding properties for the focused CN node in the left DrawPane area.

To design a constraint net model, users need to choose a graphical CN node from the ToolPane, then drag-and-drop it to the DrawPane. After that, they customize the node's properties in the PropertyEdit window. They can also use *connections* to wire two CN nodes from the source's output port to the destination's input port. In order to support bottom-up modeling, we implement the "grouping" function to allow designers to generate a higher level of new module component which accommodates those grouped nodes. To support top-down modeling and middle-out modeling, we provide child windows to display modules' content. By double-clicking a module component, its child window pops up accordingly. To make the user interface respond as quickly as possible, CNJ uses multiple-threaded programming with an appropriate synchronization mechanism.

In CNJ, there are seven packages. For different purposes, Java classes are placed under the packages separately (See Figure 4.3).

- **GUI package**

There is plenty of development work related to user interface implementation. There are several windows and a few panels for designing constraint nets interactively. CNFrame.java corresponds to the left main window. PropertySheet.java and PropertySheetPanel.java correspond to the right main window. Connection.java implements the *connection* class in CN. ToolPane.java is for the left column panel in the CNFrame window, in which constraint nets' graphical elementary nodes are located. BeanWrapper.java seems like a "wrapper" for a Java bean. Every instantiated bean is actually contained and packed in a BeanWrapper object when it is shown in CNJ. EditedAdapter.java works as an adapter which can listen to the PropertyChangeEvent, and transfer the event to a proper object to handle it. In the center of the CNFrame window, it is a Multi-Document desktop with a set of child windows within it. The child window is realized in DrawFrame.java. PrintUtilities.java enables the printing of a constraint net, and the export of a constraint net to a postscript file. DrawPane.java is a big file which has approximately 2,000 lines of code. It works as a drawing area embedded in the DrawFrame child window. Lots of constraint net drawing and designing mechanisms are implemented in the class of DrawPane.

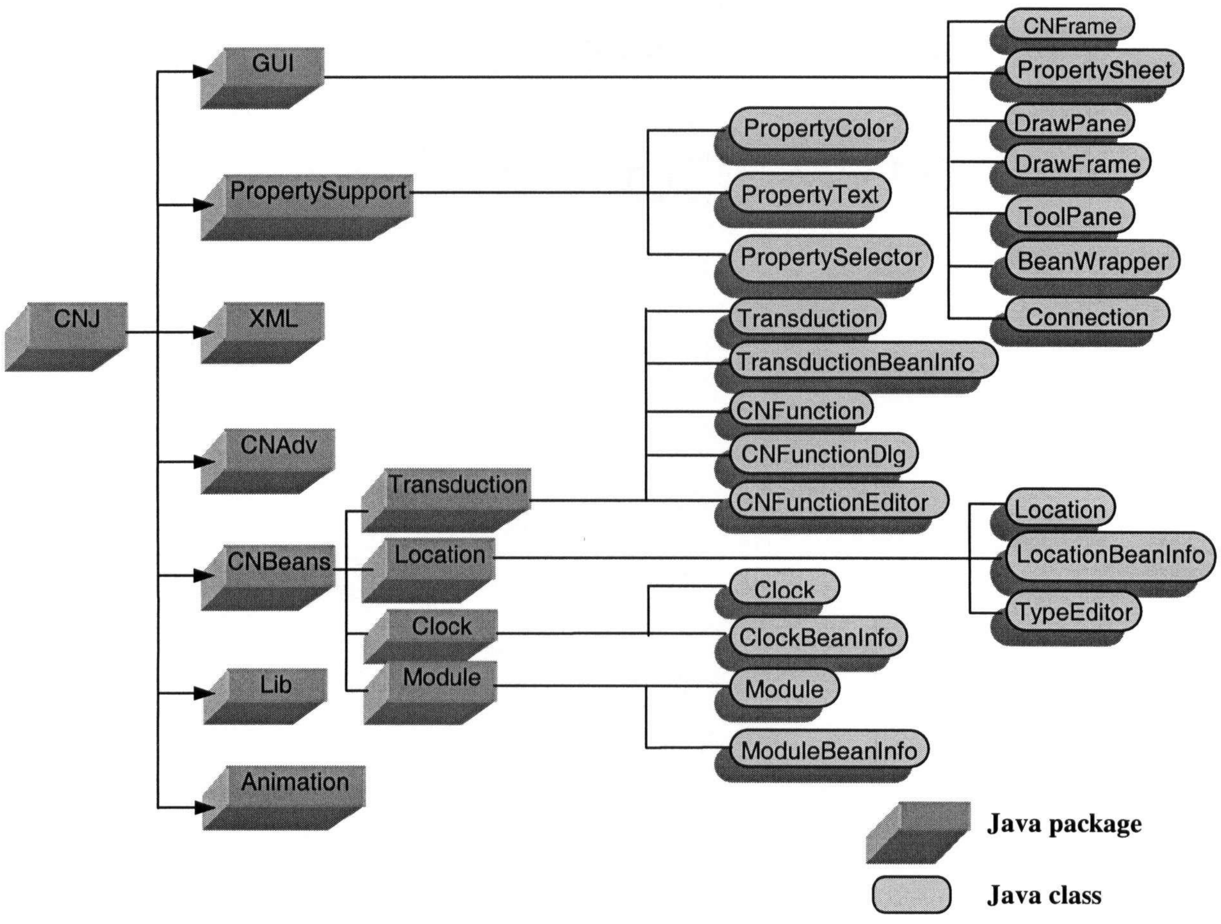


Figure 4.3. Class architecture graph

- **PropertySupport package**

By the means of JavaBeans Introspection, CNJ is able to detect a bean's properties. A property might have either a string, integer, float, or Color type. For each type, there is a PropertyEditor to allow for displaying and editing this type of data. PropertyColor.java supports the PropertyEditor for the *Color* property; PropertyText.java supports the

PropertyEditor for the *String* property; PropertySelector.java supports the PropertyEditor for the *Items* (multiple options) property.

- **xml package**

  After constraint nets are finished or partly completed, they are stored in an XML-based file format of CNML (Constraint Net Markup Language). To read and write a CNML file, the class of CNMLIO.java is implemented and put in the xml package.

- **CNAdv package**

  At the top right of the CNFrame window, there is an animation panel for advertising CNJ tool. Its classes are placed under this directory.

- **CNBeans package**

  All of the CN beans are placed in this package. It includes Java bean classes for transduction, location, clock, and module. These are inherited from one parent class named CNNode. Also, there are four subdirectories under the CNBeans: Clock, Transduction, Location, and Module. Since Transduction is realized as a Java bean, its bean supporting files are placed under its subdirectory, including:

  - Transduction.java
  - TransductionBeanInfo.java
  - CNFunction.java
  - CNFunctionDlg.java
  - PropertyCNFunction.java
  - FunctionCustomEditor.java

Similarly, the Location package has Location.java and LocationBeanInfo.java. The Clock package has Clock.java and ClockBeanInfo.java. Also, there are Module.java and ModuleBeanInfo.java in the package of Module.

4.4 Summary

This chapter gives the design idea, system requirements, and system design of CNJ. The requirement analysis determines that CNJ has the interactive drawing look-and-feel, while supporting visual modeling as well as real-time simulation. The utilization of component-based technology (JavaBeans) gives rise to the extensibility of CNJ and reusability of module components. Moreover, the class architecture and package composition of CNJ are also presented.

CHAPTER 5 GRAPHICAL USER INTERFACE DESIGN

This chapter illustrates the GUI design of CNJ. The tool is a component-based modeling and simulation environment where each CN node is a bean, and CNJ itself, is a JavaBeans introspecting environment. The design methodology is based on assembling or drawing a constraint net from a set of graphical building blocks (CN nodes), and then running or simulating that program. CNJ aims to provide users with numerous convenient features, such as drag-and-drop, zoom in, zoom out, move, automatic connection wiring, and so on. To avoid the slow response of GUI written in Java, it uses multi-threaded programming among most UI panels. The following sections describe the user interface architecture, UI threads, important supporting classes, and CN nodes as beans.

5.1 User Interface Architecture

In CNJ's user interface, there are dozens of panels with different purposes. To support human-computer interactive functions, CNJ's windows have several layers of Java components. First, we introduce the concepts and functions of those panels. Then we describe how the GUI is composed.

5.1.1 GUI Introduction

As described in Section 4.3, there are two main windows in CNJ: CNFrame and PropertyEdit. CNFrame is the left bigger window for supporting constraint net designing and simulation. PropertyEdit is the right one, supporting the dynamic customization of the focused object's properties. Figure 5.1 presents the composition graph of CNFrame.

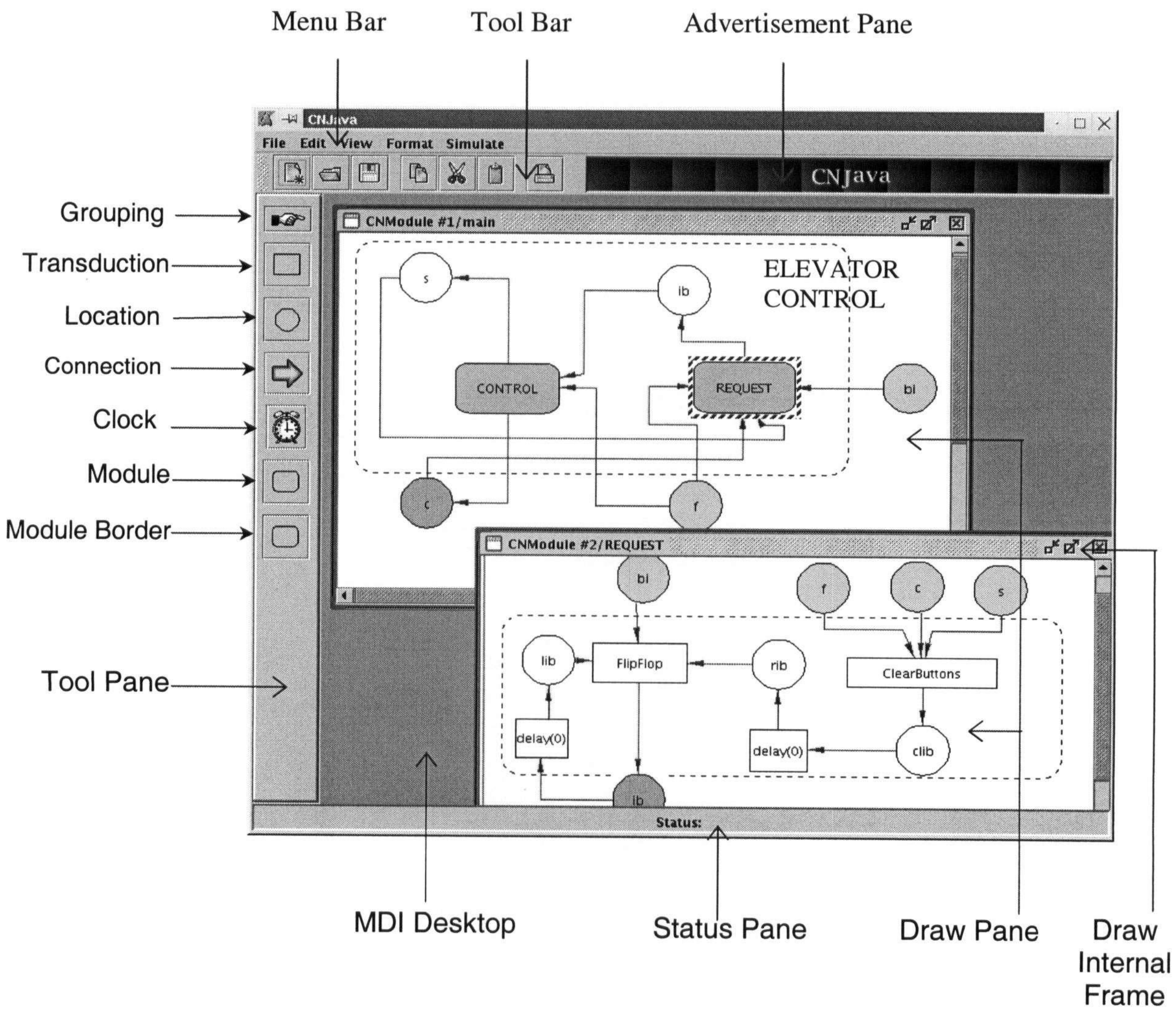


Figure 5.1. CNFrame composition

On the *Tool Pane*, there are seven icons. The five icons of *Transduction*, *Location*, *Connection*, *Clock*, and *Module* are CN nodes, which serve as building blocks for designing a constraint net. Whenever a node is needed, users click its corresponding icon and drag-and-drop it to the central *Draw Panes*. The "grouping" icon is used in the method of bottom-up modeling. That is, when a user wants to convert several components into a module, he or she uses the "grouping" function to circle those components, and then a new module is created from them at a higher level. "Module Border" is used for drawing a dashed-line round-corner rectangle. In a constraint net module, a border is required to separate input and output *interface locations* from other components.

In the center of the CNFrame is the MDI desktop. The MDI desktop is used as a container to create a multiple-document interface, or a virtual desktop. It manages potentially overlapping Draw InternalFrames, as there are probably dozens of Draw InternalFrames within it. A Draw InternalFrame extends JInternalFrame and can be added to the MDI desktop. A Draw InternalFrame displays a set of constraint net components. It is either a CN module or the highest level of a model. The Draw InternalFrame usually corresponds to a specific module at a higher level. When users double click a module component, its Draw InternalFrame window pops up accordingly.

Each Draw InternalFrame contains a Draw Pane. Draw Pane is the actual area for painting visual components and allowing them to be visually manipulated. Before being added to a Draw InternalFrame, Draw Pane is placed into a JScrollPane, which provides a scrollable view of Draw Pane. When users layout a constraint net larger than CNJ's window size, the drawing area is extended automatically. In CNJ, most of the work of constraint net modeling is done in the Draw Pane windows.

Status Pane provides users with interactive information, such as prompts, running results, and online help. *Advertisement Pane* is an animation window used to show lines of fancy words about CNJ with animated effects, such as contributions, authors, copyrights, and acknowledgements.

5.1.2 GUI Composition

The graphical user interface of CNJ is created using Java Swing components. It has a predefined containment hierarchy, which is hard to understand unless reading the source code. We introduce those Swing components, and show how they fit together into a multi-leveled containment hierarchy. The root of the containment hierarchy is a top-level container, which provides a place for its descendent Swing components to paint themselves.

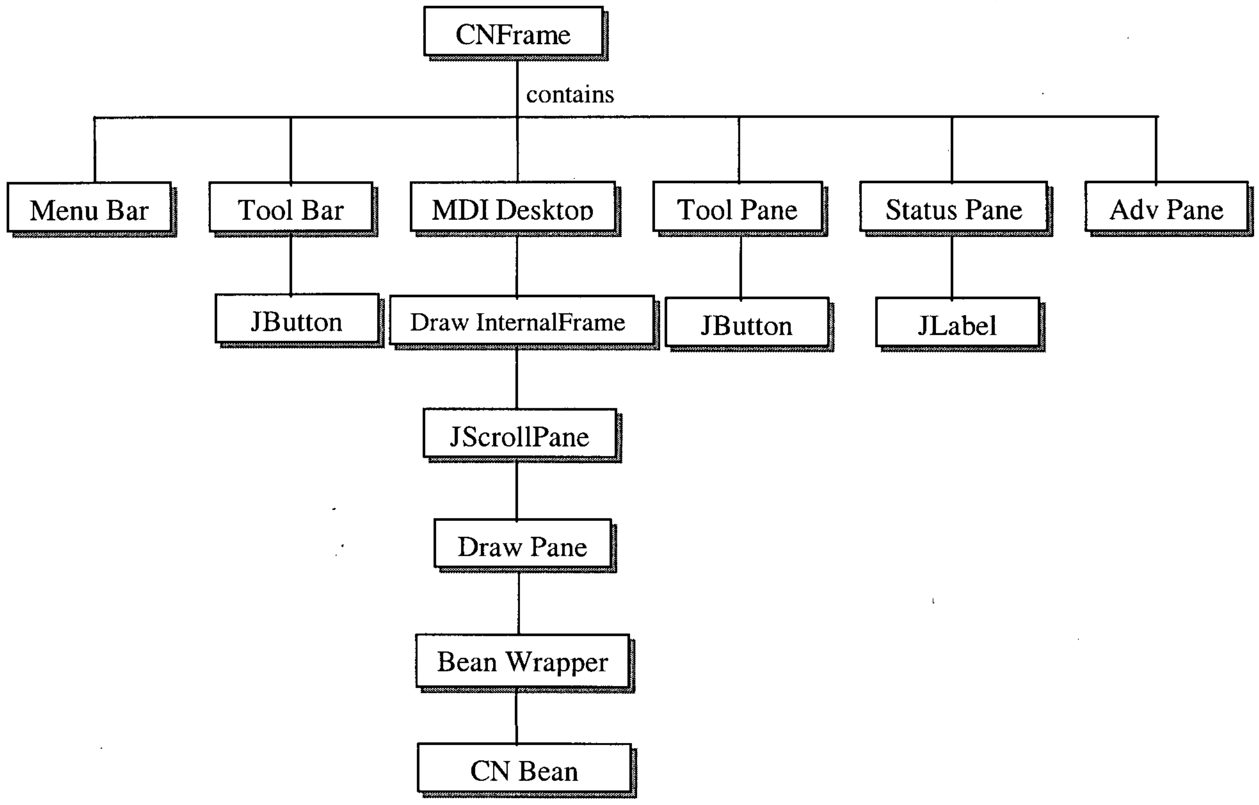


Figure 5.2. Containment hierarchy of CNJ GUI

Figure 5.2 is a diagram of the containment hierarchy for the CNFrame window. It shows each container created or used by CNJ, along with the components it contains.

CNFrame is the top-level container, which extends the class of JFrame. It contains components of Menu Bar, Tool Bar, MDI Desktop, Tool Pane, Status Pane, and Adv Pane. It is also composed of seven levels in its containment hierarchy. In Figure 5.2, an upper parent component is the container of its child components.

5.2 Multi-threaded Panels

Programs implemented in Java often involve slower speeds because of the Java virtual machine (JVM) that often interprets the bytecode at the runtime. To avoid this in CNJ, we implement multiple threads in programs to make the user interface respond as quickly as possible.

In the Draw Pane, users can choose one specific component, and focus on it with shaded hash bars around it. A thread called *focusThread* is created in CNFrame.java to handle the focus change. Whenever the focus has changed, the thread detects it immediately, and draws hash bars around the new focused object.

When designing a constraint net, users pick an icon from the Tool Pane, then put it into a Draw Pane. To respond to this picking action, CNJ creates an internal thread of *insertThread* in ToolPane.java. The thread detects that the user has clicked an icon and that it instantiates a bean object for the icon.

The *eventThread* is implemented in DrawPane.java. It is blocked by, and waiting for, a mouse click event. The thread distinguishes two kinds of mouse click events. One mouse click event is used for inserting a Java bean. The other one is used for drawing a CN connection. Two different processes are notified and executed for the two mouse click events.

Finally, the thread of *animationThread* is created in the *animation* package. It runs separately from the CNJ simulation. Depending on its internal clock, it runs in a variety of speeds. In addition, the animation thread is capable of setting a user's input to *input locations* of a constraint net, and at the same time, displaying the simulation result in an animation window.

5.3 Important UI Classes

In the GUI package, there are a dozen of classes. Among them, DrawPane is the most important one. Other classes such as BeanWrapper, CNFrame, and BeanPropertySheetPanel are also important. In this section, we introduce the classes of DrawPane (shown in Table 5.1) and PropertySheetPanel. The simpler methods in the DrawPane class are omitted from Table 5.1.

5.3.1 DrawPane Class

public class DrawPane	
Method Summary	
Public	DrawPane(CNFrame mainFrame) Constructs a new Draw Pane. The argument of mainFrame is the left CNFrame window which is unique for each CNJ running.
public CNFrame	getCNFrame() Returns the handle of the left CNFrame window..
public void	insertBean(Object bean, String beanName) Adds a Java bean to the Draw Pane.
public void	startMove(BeanWrapper w, int x, int y) Initializes a move operation after users have clicked the mouse button on the wrapper while move-cursor appears.
public void	startResize(BeanWrapper w, int x, int y, Cursor cursor) Initializes a resize operation after users have clicked the mouse button on the wrapper while resize-cursor appears.
public void	MousePressed(MouseEvent event) Starts the "grouping" operation if some condition is satisfied.
public void	MouseReleased(MouseEvent event) Handles the mouse release event.
public void	mouseClicked(MouseEvent event) Handles the mouse click event.
public void	mouseMoved(MouseEvent event) Draws a connection segment like a rubber band.
public void	MouseDragged(MouseEvent event) Handles the mouse drag event.
public void	SetDraggedHandle(int index) Records the index of the handle which is chosen by a user from a connection.

Continued

private rectangle	GetGroupingBox(int deltaX, int deltaY) Figures out the on-screen box to be drawn to represent the proposed new location of a grouping box.
private void	DrawModuleBox(Graphics g, Rectangle box) Draws a dashed line round-corner rectangle as a module border.
public void	run ( ) A thread's running body to handle users' mouse clicking events.
public void	DoEventHookup( ) Starts to draw a constraint net connection between two beans, and create the first segment of that connection.
public void	scrollLayout(Rectangle area, BeanWrapper wrapper) Checks whether a bean is out of the area of the original window. If it is, CNJ will adjust the size of the drawing area to fit it.
private void	GroupToModule(int x1, int y1, int x2, int y2) Groups the components inside the rectangle of (x1,y1) and (x2,y2) into a new module, which is created at a higher level.

Table 5.1. DrawPane class

There are a few other methods for supporting the creation of new CN beans based on the data read from a CNML file. These methods includes: *createTransductionFromElement*(), *createLocationFromElement*(), *createModuleFromElement*(), *createConnectionFromElement*(), and *createClockFromElement*().

Other methods provide support for drawing connections, while keeping connection segments horizontal or vertical, selecting a component, deleting a component, dragging a rubber line, scrolling a view, building the communication between two beans, compiling a designed constraint net, as well as running simulations, and so forth.

5.3.2 PropertySheetPanel Class

CNJ realizes the Bean Introspection mechanism which enables the identification of a bean's properties and methods. Using this method, we do not need to customize a different PropertyEdit window for each kind of bean, and CNJ can add appropriate components into the PropertyEdit to form a new edit window for the bean. This step is executed automatically by programs.

The PropertySheetPanel class mainly supports Bean Introspection. Whenever users click an object in the left CNFrame window, the setTarget( ) method in the PropertySheetPanel is called. The method first identifies the new focused object's properties, then creates and adds their corresponding visual components in the PropertyEdit window. Its code is shown in Figure 5.3.

```
/**********************************************
 *   setTarget( ) provides support for Bean Introspection.
 *   The Argument w is current focused bean in the left
 *   CNFrame window.   When the focus is changed, this
 *   method will be called. The visual components for
 *   that focused bean are then added into PropertyEdit.
 **********************************************/
public void setTarget(BeanWrapper w) {
        propertyFrame.getContentPane().removeAll();
        removeAll();

        wrapper = w;
        bean = w.getBean();

        try {
              BeanInfo bi = Introspector.getBeanInfo(bean.getClass());
              pds = bi.getPropertyDescriptors();
        } catch (IntrospectionException ex) {
              System.err.println("PropertyEdit Window couldn't introspect");
              return;
        }
        int len = pds.length;

        editors = new PropertyEditor[len];
        values =   new Object[len];
        views = new JComponent[len];
        names = new JLabel[len];

        // create event adaptor
        EditedAdaptor adaptor = new EditedAdaptor(propertyFrame);

        for (int i = 0; i < len; i++) {
              String name = pds[i].getDisplayName();
              Class type = pds[i].getPropertyType();
              Method getter = pds[i].getReadMethod();
              Method setter = pds[i].getWriteMethod();
```

Continued

```
            if (getter == null || setter == null) continue;
            JComponent view = null;
            try {
                Object args[] = {};
                values[i] = getter.invoke(bean, args);
                Class pec = pds[i].getPropertyEditorClass();
                PropertyEditor editor = null;
                if (pec != null) {
                    editor = (PropertyEditor) pec.newInstance();
                }
                if (editor == null) {
                    editor = PropertyEditorManager.findEditor(type);
                }
                if (editor == null) {
                    System.out.println("Can't find the PropertyEditor for: "+ name);
                    continue;
                }
                editors[i] = editor;
                editors[i].setValue(values[i]);
                editors[i].addPropertyChangeListener(adaptor);
                if (editor.getValue() instanceof Color) {
                    view = (JComponent)
                        new PropertyColor(propertyFrame,editor);
                    views[i] = view;
                }
                if (editor.getValue() instanceof String) {
                    view = (JComponent)
                        new PropertyText(editor);
                    views[i] = view;
                }
                if (name.equals("function")) {
                    view = (JComponent)
                        new FunctionCustomEditor(propertyFrame,editor);
                    views[i] = view;
                }
                if (name.equals("type")) {
                    view = (JComponent)
                        new PropertySelector(editor);
                    views[i] = view;
                }
             } catch (InvocationTargetException ex) {
                System.err.println("Can't invoke a getter method");
                ex.printStackTrace();
                continue;
             } catch (Exception ex) {
                System.err.println("Can't handle the property: "+name);
                System.err.println(ex);
                continue;
             }
             names[i] = new JLabel(name);
             add(names[i]);
             add(views[i]);
        }
        propertyFrame.getContentPane().add(this, BorderLayout.CENTER);
        propertyFrame.show();
}
```

Figure 5.3. Method of setTarget( ) in BeanPropertySheet

5.4 Constraint Net Nodes

The building blocks for constraint net modeling consist of *transductions*, *locations*, *modules*, *clocks*, and *connections*. Each of these is implemented as a Java bean, except for *connection*, which is implemented as a geometric polyline. Their classes are put separately into different packages. To implement the CN nodes as Java beans, some programming rules and Java classes are required.

The CN nodes of transduction, location, module, and clock have some common attributes and methods. We place these common attributes and methods into one parent class named CNNode. The CNNode is thus inherited by all the constraint net node classes shown in Figure 5.4.

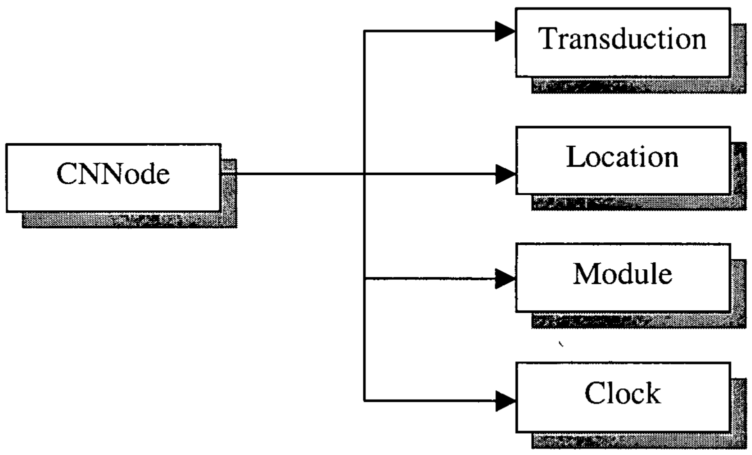


Figure 5.4: CN bean classes

5.4.1 Transduction Bean

The class of Transduction is a child class of the CNNode. Its related classes are put under the directory of Transduction (in one package). They are classes of Transduction, TransductionBeanInfo, CNFunction, PropertyCNFunction, FunctionCustomEditor, and CNFunction.

The Transduction class contains an important attribute of the *CN function*, which is to define its supported function. It implements a variety of methods for computing different functions, such as +, -, *delay*, *sine*, *cosine*, and so forth. The CNFunction class represents a constraint net function, and has the attributes of function name, input number, input data type, output data type, and input buffer. TransductionBeanInfo class provides explicit information about a transduction bean's properties, methods, and events. Without the BeanInfo class, all of its implicit properties, methods, and events from Java foundation classes are shown in CNJ.

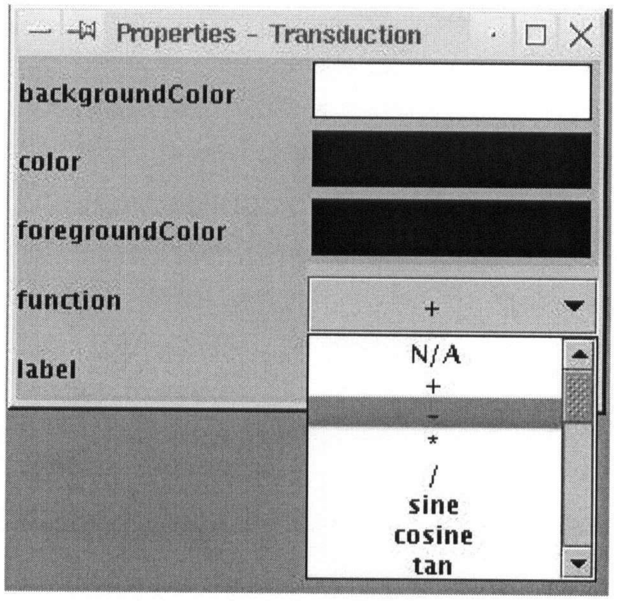
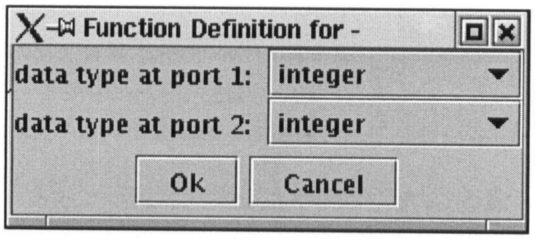


Figure 5.5. PropertyEdit of Transduction


Function Definition for -

data type at port 1: integer

data type at port 2: integer

Figure 5.6. Transduction definition for "minus"

The class of PropertyCNFunction inherits from PropertyEditorSupport the ability to allow for editing a CNFunction object. After finding the CNFunction type, CNJ displays a specific visual view for it. The visual view is provided in the class of FunctionCustomEditor which supports both the showing and choosing of a function name in a ComboBox (see Figure 5.5). After users choose a function name, a new dialog pops up to let them customize that function (see Figure 5.6). The implementation of function customization is in the class of CNFunctionDlg.

5.4.2 Location Bean

The Location class also inherits from CNNode. A location is used to store a *trace* of a constraint net. Its attributes include location type, trace clock, and trace value. A location type can be either *integer*, *real*, *boolean*, *event*, or *vector*. Its bean-related classes are put under the directory of *Location*. The class of LocationBeanInfo provides explicit information about Location bean's properties, methods, and events. Based on the LocationBeanInfo, the properties of Background Color, Foreground Color, Color, Label, Type, Trace Clock, and Trace Value are shown in the PropertyEdit window. Extended from PropertyEditorSupport, the LocationTypeEditor class allows CNJ to access a location's *type* value.

5.4.3 Clock Bean

The Clock contains classes similar to what Transduction and Location have: Clock and ClockBeanInfo. A clock bean has an important property of "cycle", which defines how many milliseconds it takes for a clock to tick. If its cycle is set to 10 milliseconds, the clock's frequency is 1000/10 = 100 Hz. To support the Clock's timing, CNJ uses Java Swing's *Timer* class. A *Timer* object is able to fire action events (clock triggering events) to each of its connected transductions in a specified frequency.

5.4.4 Module Bean

Module classes are put under the directory of Module. Similar to the other beans, the Module bean has related classes of Module and ModuleBeanInfo. The module itself does not have many properties, and it only has Background Color, Foreground Color, Color, and Label. Each module bean corresponds to a constraint net module, and has a child window to show its contents.

Modules are generally used for the methods of top-down, bottom-up, and middle-out modeling. They are hierarchically located in different levels. To provide support for the communication between modules in CNJ, some special methods are developed. A module bean's input location is relayed inward to the module's input *interface location*, and a module's output *interface location* is relayed outward to the module bean's output location. When the module is driven by a clock, its transductions must be triggered in a correct order because the transductions have a dependency relationship among them. CNJ uses the transduction scheduling algorithm and a heuristic method to find their dependency, and then lets the clock drive them in that order.

5.4.5 Connection

A connection is one of the building blocks for constraint net modeling, but it is not implemented as a Java bean. A connection is drawn as a red directed polyline to connect locations and ports of transductions. When a connection is created between one location and one transduction, there is an event transfer from the source to the destination. The Connection class is put under the directory of *GUI.* Methods in Connection.java deal with drawing a polyline, editing and moving handles of the polyline, as well as checking the compatibility of data types for the connected location and transduction.

5.5 Summary

This chapter concentrates on an overview of CNJ's graphical user interface. It introduces the composition and architecture of the user interface. To provide an easy-to-use, extensible and convenient user interface, CNJ creates multiple threads in program and utilizes the JavaBeans Introspection mechanism. Furthermore, it describes some important UI classes and the detailed implementation of CN nodes as Java beans: transductions, locations, clocks, modules, and connections.

CHAPTER 6 CNJ SIMULATION

After modeling a hybrid system in constraint nets, designers can run a simulation to verify the model. This chapter introduces how to support the real-time simulation in CNJ, which implements the object-oriented Java Event mechanism. Before running the simulation, a compilation step is used to detect the syntax errors. Since CN is a generalization of dataflow with multiple data and events, and contains a set of dependency relationships, CNJ has to find a correct order for the data flow using the Transduction Scheduling algorithm. Another issue on the real-time simulation of CNJ is also discussed in this chapter.

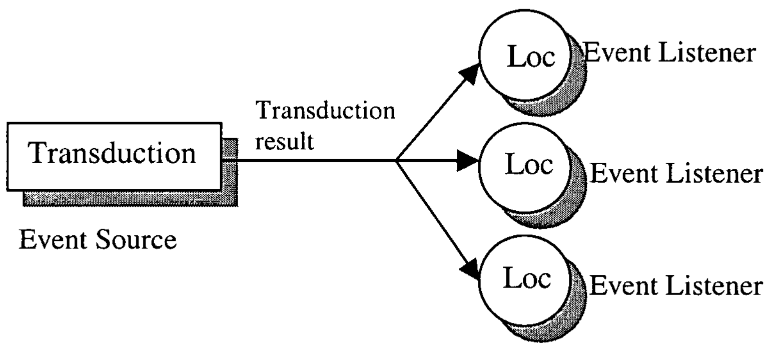
6.1 Java Event Mechanism

In the past, a program which tried to figure out what the user was doing, had to actively ask for such information itself. This method often implies an infinite loop which repeatedly checks a user's status, and then takes the appropriate action. This technique is known as *polling*. Polling is workable, but it tends to be unwieldy when used in modern-day applications because of its CPU waste and large code in one location.

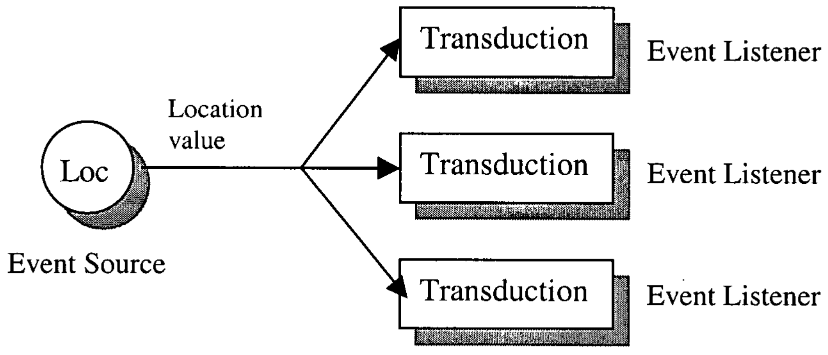
In Java, these problems are solved by event driven programming, which is also popular in Window systems. An *event* describes a specific kind of user action. Rather than the program collecting events, JVM notifies the program when an interesting event occurs. Programs that handle user interaction in this way are called event driven [19].

To receive an event, a Java object has to implement the appropriate interface, and be registered as an event listener on the appropriate event source. After hearing an event, the object performs some actions to handle the received event. Each event-handling code executes in the single thread of *event-dispatching* thread. This ensures that each event handler finishes executing before the next one executes.

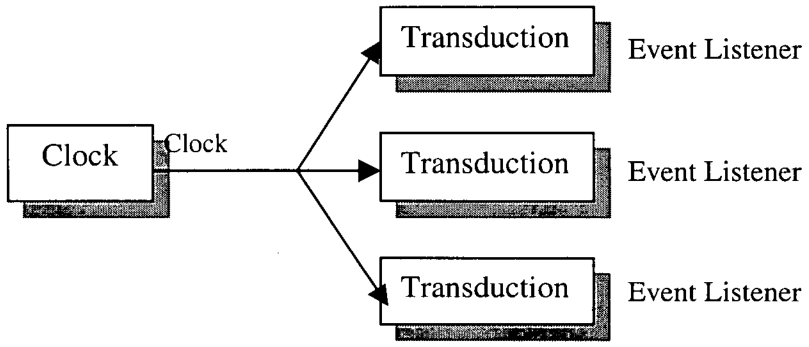
In CNJ, a connection's target node registers to listen to the events from the source node of the connection. After setting up a connection between a location and a port of transduction, the data flows from the location to the transduction, or from the transduction to the location, depending on the connection's direction. Different from event types used in the GUI implementation, CNJ utilizes the *PropertyChangeEvent* to support the CNJ simulation. In CNJ, the PropertyChangeEvent is applied in three cases: (a) Transduction works as the event source and Location as the event listener. (b) Location works as the event source and transduction as the event listener. (c) Clock works as the event source and transduction as the event listener. These three cases each carry different data objects. These are shown in Figure 6.1.



(a) Transduction as event source, Location as event listener



(b) Location as event source, Transduction as event listener



(c) Clock as event source, Transduction as event listener

Figure 6.1. Three types of PropertyChangeEvents in CNJ Simulation

To realize such a mechanism for supporting PropertyChangeEvent propagation between constraint net nodes, three bits of code are required:

1) In the event listener class, write the code to specify that the class implements the PropertyChangeListener interface.

```
public  class  CNBean implements  PropertyChangeListener {
```

2) In the event source class, write the code that registers an instance of the event listener class.

```
public  PropertyChangeSupport  pceListeners  =
                         new  PropertyChangeSupport(this);
pceListeners.addPropertyChangeListener(instanceOfCNBean);
```

3) In the event listener class, write the code that executes the event handler in the listener interface. For instance, a transduction bean calculates a new value after getting its inputs, and then fires an event to its connected output location.

```
public  void  propertyChange(PropertyChangeEvent evt) {
    ...  //  code that handles those heard events.

    // Transfer its new value to its registered event listeners.
    pceListeners.firePropertyChange("Event  Name",
                                    newvalue,
                                    oldvalue);
    }
```

After a constraint net is designed, the relationship between event sources and event listeners is built up. Whenever there is a data change, the event of data changing is propagated from the source to the connected listening nodes, which continues to propagate after some processing. During the process of propagation, CN clocks are indispensable. *Clock* is a special kind of *transduction* with no input *locations*. It has a specifiable frequency. A clock drives all of its connected *transductions* in each cycle to trigger them to compute their functions. Without clocks, data is not able to flow further, since transductions' function calculation and data propagation happen only if they get a clock event.

The following constraint net example in Figure 6.2 represents an equation of x(t) = x(t-1) + step. We assume x = 100, step = 2, at t = 0, and the time unit is equal to the clock's cycle (in milliseconds).

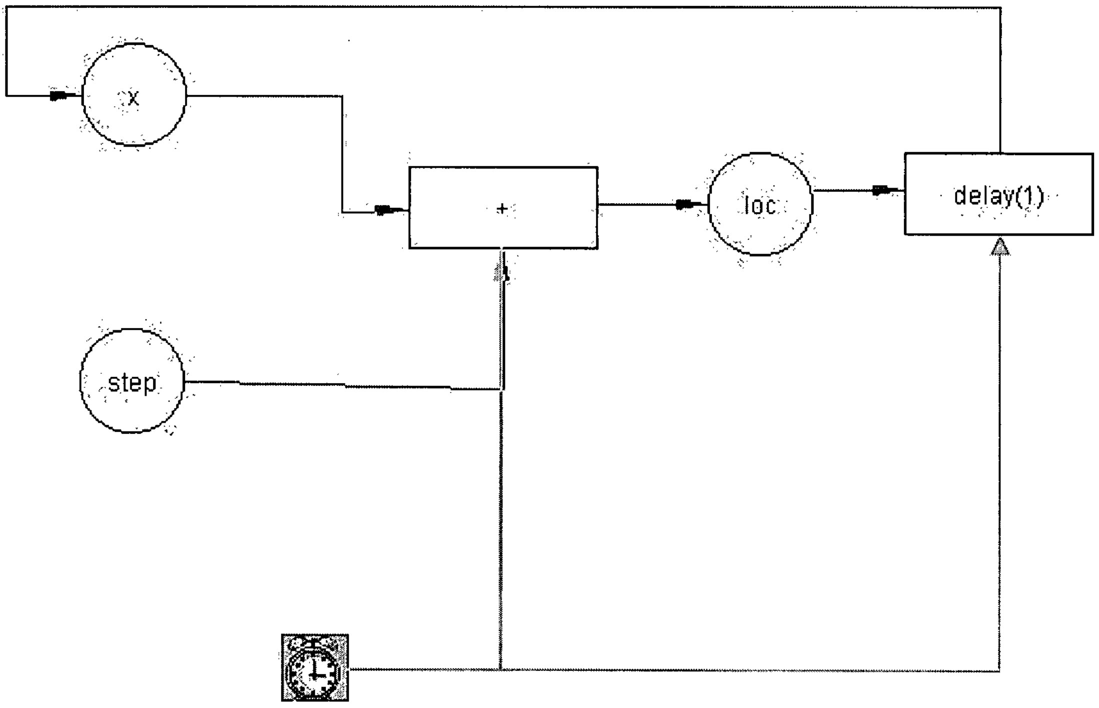


Figure 6.2. A constraint net to represent *x(t) = x(t-1) + step*

Users are able to run the simulation where transductions of "+" and "delay(1)" are driven by a clock. The simulation output is shown in Table 6.1.

	t=0	t+1	t+2	t+3	t+4	t+5	t+6	t+7
x	100	102	104	106	108	110	112	114
step	2	2	2	2	2	2	2	2

Table 6.1. Simulation output of a simple constraint net

6.2 The Compilation Step of CNJ

In the design of constraint nets, users might make some errors which are either syntactic or semantic. In CN, there is a set of syntax rules for designing a well-formed constraint net. Formally, a constraint net is a triple CN = <Locations, Transductions, Connections>. To design a *connection*, users have to follow restrictions:

(1) there is at most one output port connected to each location;
(2) each port of a transduction connects to a unique location; and
(3) no location is isolated.

The above restrictions also affect the design of transductions and locations. First, a transduction/location cannot be directly connected with another transduction/location, which means a transduction is always connected to a location. Second, each location can have only one output port. Third, a location's input should be dependent on a unique transduction's output. Fourth, a location's data type should be compatible with the data type of its connected transduction port.

Not obeying the rules results in syntax errors. In CNJ, we implement a compilation module to detect syntax errors. After a user completes designing a constraint net, he or she clicks the menu to start a simulation. However, the simulation cannot run unless it passes the compilation. The compilation module works as follows.

- First, it checks the data type compatibility for the connected locations and transduction ports. There are 5 kinds of data types in CNJ: integer, float, boolean, event, and vector. When finding such a kind of error, it displays the prompt

"Error: data type incompatibility from Location→ Transduction *type1*, *type2*" if data flows from a location to a transduction.

- Second, it checks to make sure that there is no case where two transductions are connected together directly.
- Third, it checks whether all the transduction's functions are defined completely and correctly. If not, a prompt "Error: undefined Transduction *name*" pops up.
- Fourth, it checks to ensure a location's input depends on a unique transduction.
- Finally, it checks whether each of the input locations in the top-level module has an initial value. If there is one input location not set, CNJ displays an error prompt "Should not start simulation since not all the input locations are set."

Whenever a syntax error is found, the simulation stops immediately. Users have to redesign their constraint nets to correct the syntax errors. CNJ's compilation step greatly reduces users' design errors. However, because it is still a prototype, it does not guarantee that it constrains users to make absolutely syntax-correct constraint nets.

Besides those syntax errors, there are semantic errors. A semantic error might occur while the simulation is running. When a Java exception is thrown out, a semantic error occurs. In our experiments, a semantic error might occur when a constraint net is very complex and has a lot of cycles (feedback) in its graph. This problem is related to the transduction scheduling algorithm, which sorts out transductions and then drives them one by one. Such an error can be avoided if users know how to break the cycles through defining some special locations in non-white color ("heuristic tips"). The transduction scheduling algorithm is introduced, in detail, in Section 6.3.2.

6.3 Dataflow Issue

CN is a dataflow with multiple data and events. Data and events produced and consumed by CN nodes are carried by Java objects, which flow along the connections from tails of connections to their heads. The dataflow computing model is based on the flow of data, not on the flow of control. Unlike the control-flow computing model, it assumes that a program is a data dependency graph whose nodes denote operations, and whose edges denote dependencies between operations. A dataflow computing model executes any operation denoted by a node as soon as its incoming edges have the necessary operands [11]. There are two kinds of dataflow models: data-driven dataflow model and demand-driven dataflow model.

6.3.1 Data-Driven and Demand-Driven Models

The dataflow model described above can be more accurately characterized as a data-driven dataflow model. It often implies the qualification of "data-driven". The data-driven dataflow model was invented as an architectural abstraction, based on which dataflow computers are built up. An *Operand set* is used as the basis for the firing rules in data-driven systems. The firing rules may be *strict* or *non-strict* [11]. A strict firing rule requires a complete operand set to exist before a node can fire; a non-strict firing rule triggers execution as soon as a specific proper subset of the operand set is formed. The latter rule gives rise to more parallelism.

Another important branch of the dataflow model is demand-driven dataflow. It was invented as a computing model to efficiently evaluate programs in non-strict dataflow languages. The basic idea behind demand-driven dataflow execution is that "an operation of a node will be performed only if there are tokens on all the input edges and there is a demand for the result of applying the operation" [11]. Its main motivation is to deal efficiently with non-strictness, something that data-driven cannot deal with efficiently.

6.3.2 Dataflow in CNJ

The language of constraint nets is a graphical programming model whose programs can be given mathematical semantics under a particular data algebra. Constraint nets appear similar to dataflow graphs, but they do not make any assumptions about how they are evaluated. It is possible to be either data-driven or demand-driven. It also often denotes a semantic model, which is a composition of other modules corresponding to subnets. To execute a constraint net program, CNJ regards it as a specific computation given a set of sequences of input values and a sequence of demands for output values of the constraint net. At the initial step, only the values of the input locations are defined. At each subsequent step, some of the undefined locations are computed and eventually, the constraint net outputs are computed.

In CNJ, a CN module component is often both connected with and driven by a clock. To evaluate the module correctly, the transductions contained in the module have to be triggered in proper sequence. CNJ uses the *transduction scheduling algorithm* to figure out a right dependency relationship within the transductions. Then the clock triggers those transductions one by one in that order. This approach allows the computation of constraint net modules to work as demand-driven dataflow, and it works well.

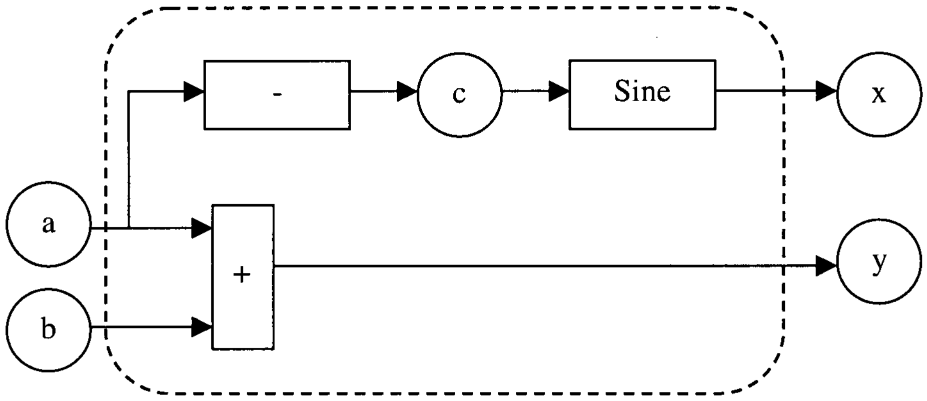
6.3.3 Transduction Scheduling Algorithm

In general, a module component is triggered by one outside clock. It consists of a set of connections, locations and transductions. The clock's driving events are transferred to the module's inside transductions, but the problem is how to drive the transductions and in what order. To find out a correct order to drive the transductions, we implement a scheduling algorithm based on the topological sorting algorithm.

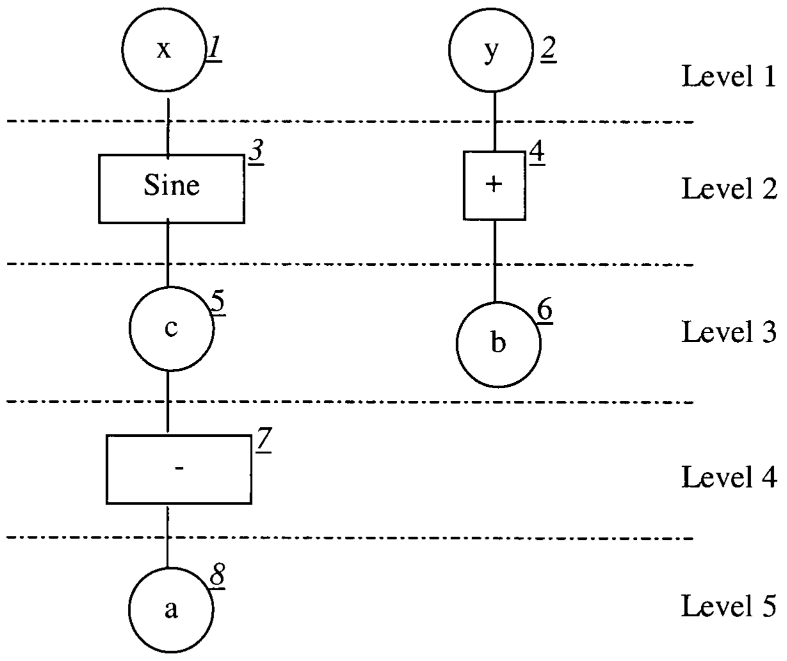
Topological sorting is a natural problem in many algorithms on directed acyclic graphs (DAG). Topological sorting orders the vertices and edges of a DAG in a simple and consistent way. It can be used to schedule tasks under dependency constraints. The problem of topological sorting is described as follows:

Input: A directed, acyclic graph G=(V, E) (also known as a partial order).

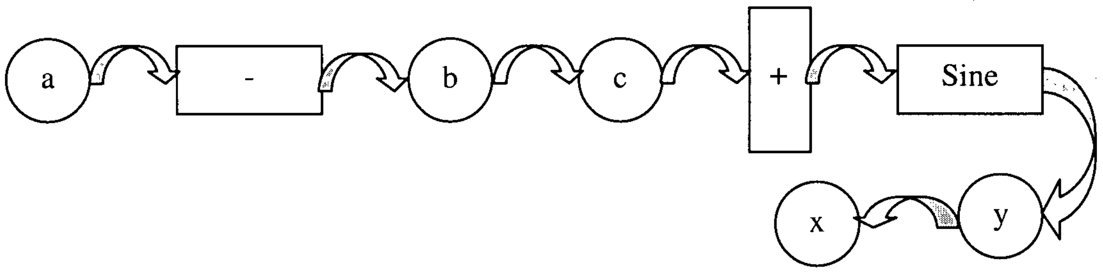
Problem: Find a linear ordering of the vertices of V such that for each edge (i,j) in E, vertex i is to the left of vertex j.



a) A constraint net module



b) The hierarchical tree after sorting



c) The sorted order to drive the nodes

Figure 6.3. A transduction scheduling example

The topological sorting problem is also applicable to constraint net graphs. Suppose we have a set of transductions to be driven in a CN module, and certain transductions must be computed before other transductions. These dependency constraints thus form a constraint net (also a directed graph). The transduction scheduling algorithm searches for an order to execute the transductions, such that each is performed only after all of its previous transductions are executed. In the implementation, it utilizes the breadth-first algorithm to transverse the constraint net graph, but in a backward way (from output *interface locations* to input *interface locations*). The algorithm picks vertices in hierarchical levels with the output *interface locations* as roots. That is, if a vertex has an *out-degree-coun*t 0 it can be next in the topological order. Then, the algorithm removes this vertex and looks for another vertex with an out-degree-count 0 in the resulting DAG. It repeats this until all vertices are added to the topological order. Figure 6.3 represents a constraint net example to illustrate the algorithm.

Graph a) shows a constraint net module. After applying the breadth-first algorithm to transverse the graph from the end to the beginning, together with an *out-degree-count* in each node, a linear order is reached in Graph b), where the numbers denote the order of the node in the breadth-first transverse. However, the order in Graph b) is not the final result yet, since the sequence number is calculated with the roots of the output locations instead of the input locations. Therefore in Graph c), a correct order is finally acquired after reversing the order in Graph b). Based on the final order, the execution of the module works correctly. Although the ordered sequence includes both transductions and locations, the clock only needs to trigger the transductions.

The transduction scheduling algorithm, however, does not work without the condition that the constraint net has to be a *directed acyclic graph* (DAG). Sometimes constraint nets have feedback connections resulting in a few cycles in the graph. In the complex case of a directed cyclic graph, those cycles have to be broken up, and then it becomes an acyclic graph.

In CN modules, a cycle forms when there is a backward connection for creating a feedback. To run the simulation, the particular location in that feedback cycle has to be assigned an initial value (or else, the involved transduction can never get inputs to compute). Such a special kind of location is regarded as a "heuristic tip" for breaking up cycles. When designing a constraint net and confronting a feedback cycle, designers are required to paint the special location in a non-white background color. It also reminds designers to assign an initial value to that location before starting the simulation. The Java method for the algorithm is listed in Figure 6.4.

```
public void driveTransductionsInOrder(long timestamp) {
        DrawPane drawP = drawFrame.getDrawPane();
        Vector vector;
        Transduction transduction;
        BeanWrapper wrapper;

        Vector orderVector = new Vector();
        int number = drawP.getComponentCount();

        HashMap visitedMap = new HashMap();
        HashMap dependentCountMap = new HashMap();
        ObjectQueue queue = new ObjectQueue(number); // a FIFO queue class.

        // First, add the output interface locations to Queue.
        vector = drawP.getOutputInterfaceLocationWrappers();
        for (int i = 0; i < vector.size(); i++) {
            queue.addToQueue(vector.elementAt(i));
            visitedMap.put(vector.elementAt(i),new Boolean(true));
        }

        while (!queue.isEmpty()) {
            Object current = queue.getFirst(); //Wrapper
            orderVector.addElement(current);
             vector = queue.getConnectedNodes((BeanWrapper) current);
```

Continued

```
            for (int i = 0; i < vector.size() ; i++) {
                wrapper = (BeanWrapper) vector.elementAt(i);
                if (!visitedMap.containsKey(wrapper)) {
                    /*  isLegalToPush( ) decides whether a node should be added
                     *  to the tail of the Queue, also breaks up cycles.
                    */
                    if (isLegalToPush(wrapper,dependentCountMap)) {
                        queue.addToQueue(wrapper);
                        visitedMap.put(wrapper,new Boolean(true));
                    }
                }
            }
            queue.removeFirstFromQueue();
        }
        BeanWrapper element;
        Object bean;

        // reverse the order to drive in the orderVector.
        for (int i = orderVector.size() - 1; i >= 0; i--) {
            element = (BeanWrapper) orderVector.elementAt(i);
            // trigger transductions to compute.
            bean = element.getBean();
       if (bean instanceof Transduction) {
                transduction = (Transduction) bean;
                transduction.computeResult(timestamp);
            }
             // trigger modules to execute.
            if (bean instanceof Module) {
                Module m = (Module) bean;
                m.driveTransductionsInOrder(timestamp);
            }
        }
    }
```

Figure 6.4. Transduction scheduling algorithm

6.4 Discussion of "Real-time"

Users might run CNJ's real-time simulation to see the system's result and verify their designed models. A constraint net has one (global) or more (local) clocks with different frequencies (Hz). The clocks are distributed in different areas or modules. By setting a clock to an infinitely high frequency, the continuous time structure can be virtually realized. It also depends on the simulation speed that the system requires.

In general, CNJ's real-time system is composed of a number of Java threads which execute concurrently, and compete or cooperate to fulfill the requirements placed on the system. Because it runs on digital computers, it cannot be of absolutely real-time and have no time delay. If those threads' *timing* (completed before their deadlines) is correct, we can say that the real-time simulation is correct. There are four types of real-time systems [26]:

- A hard real-time system where a missed deadline causes the failure of the system.
- A firm real-time system where there is no associated value if finishing after the deadline, but the system can continue to run.
- A soft real-time system where missed deadlines are not ideal, but can be tolerated.
- An interactive system where there is just the "adequate response times".

We develop the *transduction scheduling algorithm* to schedule the transductions for execution. Since the algorithm produces the correct order that obeys the transductions' dependency relationship, the calculation result should be correct. In our environment where we have a Linux Redhat 7.3 box with a PIII 1G CPU and 256M RAM, the CNJ's simulation can run correctly in the fastest speed of 50 Hz if the CN model is not too big. Its performance heavily depends on the hardware speed and its specific operating system. Moreover, the correctness of the real-time system depends not only on the logical result of its computation, but also on the time at which the results are produced.

CNJ simulation runs on Linux boxes. Unix supports a time-slicing (preemptive) multi-threaded system and it ensures that multiple threads of the same priority share CPU time. When running CNJ in Linux, there are a group of processes created for the threads to run. Also, the Linux scheduler latency affects any process's speed. If the scheduler latency is big, it is impossible for a real-time process to respond faster than the latency. Most Real Time Operating Systems (RTOS) have kernel code that is written to be preemptive. However, the Linux kernel is not a preemptive kernel where all interrupts are created equal [38]. This means that once the kernel has been entered from a system call, the current process cannot be unexpectedly changed.

Linux is not designed to be a RTOS. If users need a hard real-time system in the 10's of microseconds, a modified kernel such as RTLinux, or a dedicated RTOS, is probably more suitable than Linux as a solution.

6.5 Summary

CNJ simulation is realized by the means of Java event mechanism. CN clocks drive the transductions to compute results, and drive the data flow along the graph. To make sure a designed constraint net is well-formed, CNJ calls the compilation module before starting the simulation. Also, the transduction scheduling algorithm is introduced to compute a proper order to drive transductions correctly. On the discussion of CNJ's real-time issue, the real-time simulation depends not only on the model's logical computation, but also on its running environment, which changes with time.

CHAPTER 7 CONSTRAINT NET MARKUP LANGUAGE

The Constraint Net Markup Language (CNML) serves as an interchange format for CNJ to store CN models. It is a preliminary proposal of an XML-based interchange format for constraint nets. This chapter presents, in the first version, the concepts and terminology of the CNML, as well as its syntax, which is based on XML. In the current version, constraint nets' general features are included in the CNML. It also provides a starting point for the development of a standard format for Constraint Nets.

7.1 XML

XML stands for Extensible Markup Language, which is a set of rules for defining semantic tags that break a document into parts and identify the different parts of the documents [27]. Different from HTML or Tex, it is a meta-markup language, in which developers make up the tags they need as they go along. It means they do not have to wait for browser vendors to catch up with what they want to do. They can invent the tags when they need to.

XML describes a document's structure and meaning. It does not describe the formatting of the elements on the page. The document itself only contains tags that say what is in the document, not what the document looks like. Formatting can be added to a document with a *style sheet.*

XML documents are most commonly created with an editor. This may be a basic text editor that does not understand XML at all, such as vi, emacs, or Notepad. Moreover, an XML document does not absolutely have to be a file on some hard disk. It can be a record or a field in a database, or it may be a stream of bytes received from the network [27]. An XML parser reads the document and verifies whether it is well formed. If the document passes the tests, the parser converts the document into a tree of elements. Finally, the parser passes the tree to the end application. If this application is a web browser such as Mozilla, the browser formats the data and shows it to the user.

However, other programs may also receive the data. For instance, CNJ might view the XML as a constraint net that is composed of transductions, locations, and connections, then display it in front of users. XML is extremely flexible, and can be used for many different purposes. Figure 7.1 shows its whole process.

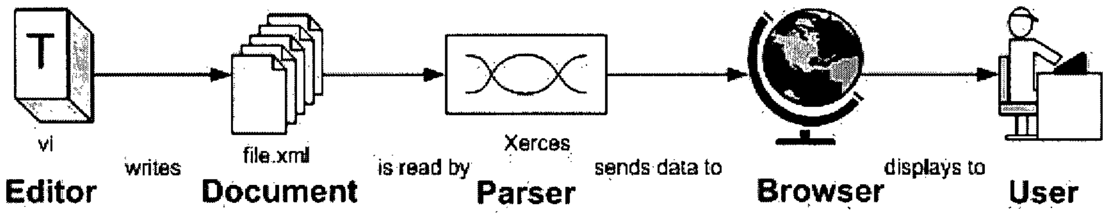


Figure 7.1. XML document life cycle

Using XML involves several related technologies and standards. These include the following:

- HTML for backward compatibility with popular browsers. It takes about two years from the initial release before most users have upgraded to a new browser version [27].
- The CSS (Cascading Style Sheets) and XSL (Extensible Stylesheet Language) languages to define the appearance of XML documents.
- URLs (Uniform Resource Locators) and URIs (Uniform Resource Identifiers) to specify the locations of documents.
- Xlinks to connect XML documents to each other.
- The Unicode character set to encode the text of XML documents.

XML brings a lot of advantages to program developers. It enables the design of field- specific markup languages. Individual professionals can develop their own markup languages. It is a self-describing and simple data format. It can be written in 100 percent pure ASCII text. In other formats (compressed data or serialized objects), the loss of a single byte can result in the entire file being unreadable. XML is designed to be easy for both human and programs to read and write. Moreover, since it is structured and integrated, XML is ideal for large and complex documents.

7.2 CNML

CNML supports all constraint net models. Each CN node has a corresponding Element defined in CNML. A file that satisfies the requirements of the interchange file format of CNML is called a *constraint net* file. Each constraint net consists of locations, transductions, connections, and modules. It represents the graph structure of the constraint net.

Conforming to XML, in CNML, a model and its constraint net nodes are represented by XML *elements* [22][27][35]. For convenience, the tags of CNML elements are named after the *constraint net* terminology (e.g., <model>, <module>, <transduction>, <location>, <connection>, and <clock>). When a constraint net model is saved to a file, each node in the file is assigned a unique ID, which can be used to refer to the object.

To give further meaning to a CN node, each node has a label. Typically, a label represents the name of an object, or the marking of a place. For instance, transductions, locations, and modules have labels.

Each CN node includes some graphical information. For a transduction, location, connection, or module, its position information must be recorded. In addition, its specific geometry shape, size, background color, and foreground color are indispensable information. For a connection, the graphical information is the polyline's coordinates and its connected nodes' IDs.

To represent the graphical information of a constraint net, CNML uses the SVG (Scalable Graphics Vector) 1.0 specification [31]. SVG is a standard language developed by W3C for describing two-dimensional graphics in XML1.0. Moreover, in order to specify a transduction's function, CNML uses MathML (Mathematics Markup Language) 2.0 specification [32]. CNML leverages and integrates with other W3C specification and standards. By conforming to other specifications, CNML becomes more powerful and makes it easier for users to transfer to other applications.

CNJ uses JDOM beta 7 [24] to deal with the CNML files, to parse, to read or write, as well as export to or import from another file format, that is, HTML, XHTML and Simulink's MDL. In CNJ, the CNMLIO class is implemented to support the handling of CNML files.

The following a), b), and c) in Figure 7.2 are Elements of *transduction*, *location*, and *connection*, described by CNML. Also, a complete constraint net for the Car Dynamics model in CNML is presented in Appendix A.

```
<transduction id = "0">
        <graphics>
                <rect x = "    " y = "    "
                 width = "    " height = "    "
                 fill = "color"
                 stroke = "color" />
        </graphics>
        <label color = "blue">
                +
        </label>
        <function>
                <input number = "2" />
                <math>
                        <apply>
                                <plus />
                                <ci type = "integer"> input </ci>
                                <ci type = "integer"> input </ci>
                        </apply>
                </math>
        </function>
</transduction>
```

a) CNML element for the "plus" transduction

```
<location id = "1">
        <graphics>
             <ellipse cx = "    " cy = "    "
                   rx = "    " ry = "    " fill = "color"
                   stroke = "color" />
        </graphics>

        <label color = "blue">
                input__x
        </label>
        <ci type = "integer" />
</location>
```

b) CNML element for a location

```
<connection id = "2">
      <polyline points = " x1,y1 x2,y2 x3,y3 ..." />
   <source> id </source>
   <target> id </target>
</connection>
```

c) CNML element for a connection

Figure 7.2. CNML Elements

7.3 Summary

Constraint Net Markup Language (CNML) is defined for Constraint Nets for the first time. It works as an XML-based interchange format to support the storing of constraint nets and the representation of them. Its XML compliant definition brings a lot of advantages to store constraint nets, and facilitates the development and sharing of constraint net models.

CHAPTER 8 A REALISTIC CN MODEL

To test and demonstrate the performance of CNJ, we modeled a complete elevator system in Constraint Nets, and simulated its running behavior. The elevator system is a typical hybrid system that works as a good example to demonstrate the power of constraint nets. In this chapter, Section 8.1 introduces the elevator system and its designed model in a constraint net. Section 8.2 presents the constraint net model generated by CNJ for the elevator system. Section 8.3 briefly introduces the simulation and animation result of the modeled elevator system.

8.1 The Elevator System

Elevator systems are used in various communities as examples of methodologies for software engineering and real-time systems. An elevator system is a typical hybrid dynamic system with continuous motion following Newtonian dynamics and discrete event control responding to users' request. In Zhang and Mackworth's paper [21], one *n*-floor elevator system is modeled using constraint nets, including the specification and verification for that CN model. Based on their model, a 3-floor elevator system is designed and simulated successfully in CNJ.

The 3-floor elevator system is used in a three-floor building. Inside the elevator there is a board of buttons to indicate floor numbers (*1*, *2*, and *3*). Outside the elevator there are two direction buttons (up, down) on each floor, except for the first and the third floor, where only one button is needed (see Figure 8.1). Any button can be pushed at any time. After being pushed, a floor button is on until the elevator reaches that floor, and a direction button is on until the elevator reaches that floor and is moving in the same direction.

The elevator system is modeled in two levels using constraint nets. At the lower lever, the continuous dynamics is modeled. At the higher level, the abstraction of the desired discrete system is modeled. Therefore, the model has a continuous elevator body and a discrete controller.

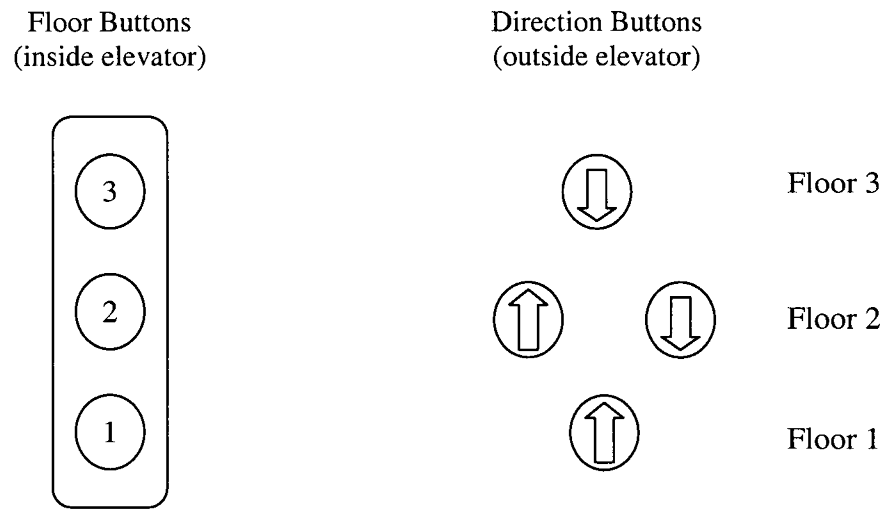


Figure 8.1. 3-floor elevator system

The rest of this section describes the constraint net modules for the elevator system. The system was first introduced by Zhang and Mackworth in [21].

8.1.1 Top-level Hybrid Model

The 3-floor elevator system is modeled as a hybrid constraint net model with its body running in a continuous time domain, and its controller running in a discrete time domain. The top-level model is shown in Figure 8.2 [21].

The EVENT module implements the event logic that produces the events for triggering the discrete CONTROL 1 module. It realizes the *or* function of the following three events:

1) $(s = idle) \wedge \bigcup_{1 \le k \le 3} (Ub_s(k) \vee Db_s(k) \vee Fb_s(k))$. A user pushes a button at the elevator's idle state.

2) $|d_S| <= \varepsilon$. The elevator comes to a home position.

3) $(Com = 0) \wedge \neg(Fb_s(f) \vee (Ub_s(f) \wedge s = up) \vee (Db_s(f) \wedge s = down))$. A user request has been served for a certain time. If it takes τ seconds to serve a user, a delay of τ will be waited for.

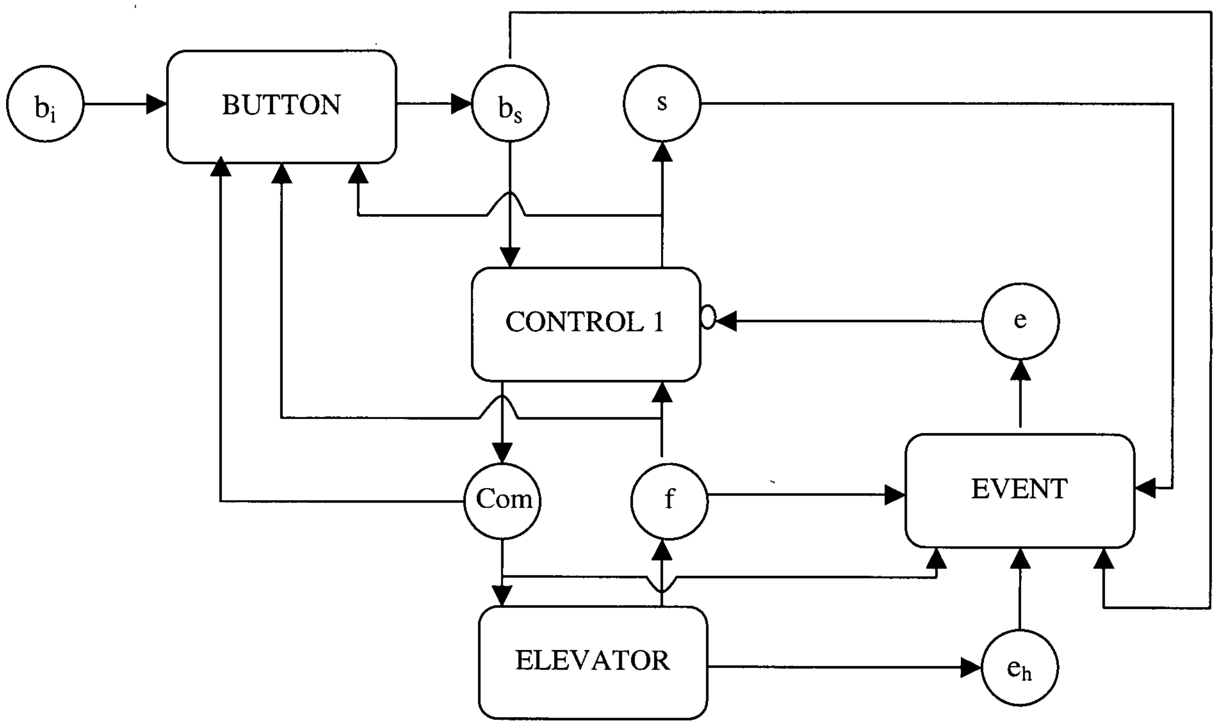


Figure 8.2. A hybrid model of the Elevator system

ELEVATOR is a continuous module where the elevator body is modeled by a second order differential equation following Newton's Second Law. It is depicted in Section 8.1.2. From Figure 8.2, we can see that the continuous part of ELEVATOR is linked seamlessly together with the discrete part of CONTROL1 by the event-driven module of EVENT.

CONTROL1 is a discrete control module with the current floor number *f* and the current button states b_S (includes Ub_S, Db_S, Fb_S for Up buttons, Down buttons, and Floor number buttons) as inputs. The command *Com*, and serving state *s*, are its outputs. Its module composition is described in Section 8.1.3.

The BUTTON module is discrete, consisting of an array of flip-flops to simulate pushing buttons. Interested readers might refer to [21] for more detail.

8.1.2 The Continuous ELEVATOR Module

The continuous ELEVATOR module is depicted in Figure 8.3 [21]. *Com* is a command from a higher level (CONTROL1), which can be 1, -1, 0 representing up, down, and stop respectively. *f* is the current floor number. When the elevator body is within a small range of a floor, e_h is set to *true*, or else *false*.

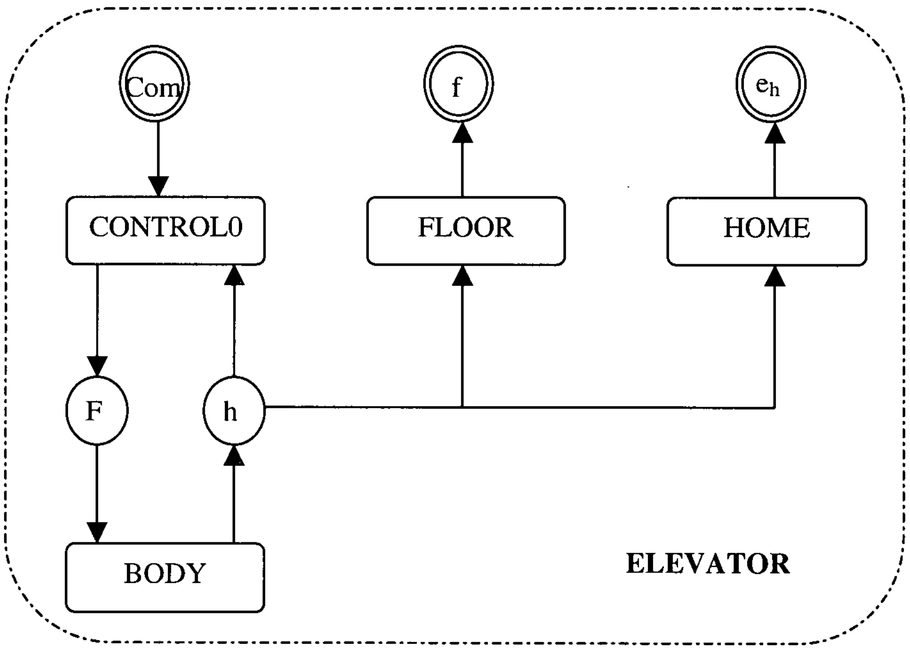


Figure 8.3. The continuous ELEVATOR module

In the FLOOR module, given the separation of floors H, and the current elevator height h, the current floor number f can be obtained by the following:

$$f = [h / H] + 1 \qquad (1),$$

where [x] denotes the closest integer to x. In the HOME module, the distance to the nearest floor is as follows:

$$d_S = h - (f-1)H \qquad (2),$$

and the home position e_h is as follows:

$$e_{h} = (| d_s | <= \varepsilon) \quad (3).$$

Unlike the discrete controller of CONTROL1, CONTROL0 is an analog controller that generates force to drive the elevator body. In our model, it is a linear *proportional and derivative* (PD) controller. The driving force F is computed by the following:

$$F = \begin{cases} F_0 & \text{if Com} = 1 \quad (4) \\ -F_0 & \text{if Com} = -1 \quad (5) \\ -K_p d_s - K_v \dot{d}_s & \text{if Com} = 0 \quad (6) \end{cases}$$

where d_s is the distance to the nearest floor, F_0 is a positive constant, K_p is a proportional gain, and K_v is a derivative gain. The F_0, K_p, and K_v should satisfy the following conditions [21]:

- Continuous Stability. The continuous control is stable.
- Hybrid Consistency. The interface to the discrete control is consistent.

To satisfy the conditions, F_0, K_p, and K_v have to be chosen based on the following equations:

$$K + K_v > 0, \quad K_p > 0 \quad (7),$$

$$F_0 \leq \min(K\sqrt{4\varepsilon d / 3}, a) \quad (8),$$

$$K_v' = F_0 / (K\varepsilon) \quad (9),$$

$$K_v = K_v' - K \quad (10), \text{ and}$$

$$K_p = K_v^2 / 4 \quad (11),$$

where F_0 is the motor driving force, *K* is the fiction coefficient, *a* is the maximum acceleration of the motor, and *d* is the maximum deceleration of the motor. For detailed deduction and calculation, readers may refer to the paper [21].

In the model of the elevator system, we set $K = 1.0$, $a = d = 0.5$, and $\varepsilon = 0.15$, $F_0 = 0.33$, $K_v = 1.2$, and $K_p = 1.21$ to satisfy the above equations.

The elevator BODY module is modeled by a second order differential equation following Newton's Second Law, as follows:

$$F - K\dot{h} = \ddot{h} \quad (12)$$

Here we assume the mass of the body is 1, since it can be scaled by F and *K*. We also ignore gravity since it can be added to F to compensate for its effect.

8.1.3 The Discrete CONTROL1 Module

CONTROL1 is a discrete controller. It receives buttons' state of *Ubs*, *Dbs*, *Fbs*, the current floor number *f*, and the current serving state *s*, and then determines the next motion *Com*, and the serving state *s* of the elevator. It is described in Figure 8.4 [21].

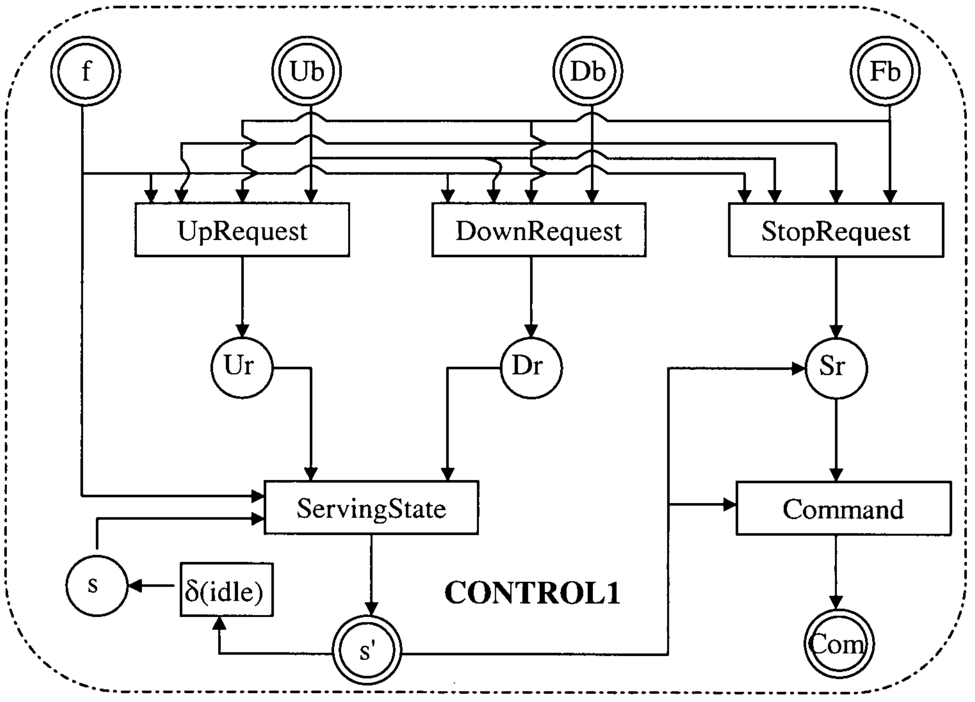


Figure 8.4. The discrete CONTROL1 module

We assume there are three kinds of serving states: up, down, and idle. In addition, we assume that the elevator is always parked (idle) at the first floor. The elevator moves persistently in one direction until there is no request in that direction. The complete logic functions of *UpRequest*, *DownRequest*, *StopRequest*, *ServingState*, and *Command* can be found in the paper [21].

8.2 Elevator System Modeled in CNJ

The 3-floor hybrid elevator system is also practically modeled and simulated in CNJ. However, it is different from the one described in Section 8.1. The model in Section 8.1 is developed at a higher level than this one modeled in CNJ.

As described in Chapter 4, CNJ provides users with a set of elementary transductions for designing a constraint net, such as *plus*, *minus*, *times*, *division*, and so forth. Before starting to design the elevator system, we already modeled a few simple control systems for which those elementary transductions are sufficient.

However, to model the much more complicated hybrid elevator system, the basic transduction set is not enough. Therefore, while modeling the elevator system, we wrote code to add new transductions and new supports to allow for elevator system modeling. In addition, some complex transductions have to be developed as modules, based on basic transductions. In other words, the implementation of CNJ is driven by models' new requirements. For instance, to support the elevator system, we added the *vector* data type and modified some transductions to make them accept *vector* data input, added the *ElementAt* transduction to access a specific element from vectors, and added the *isChanged* transduction, as well as the *wait* transduction.

The elevator system modeled in CNJ is composed of three levels. There are in total, 17 modules, 169 transductions, 2 clocks, and hundreds of locations. The top-level model in CNJ is shown in Figure 8.5. The other module graphs are given in Appendix B, since they have similar designs to those depicted in Section 8.1.

8.3 Elevator System Simulation and Animation in CNJ

The simulation of the modeled elevator system is able to run in two ways. First, users input data directly into locations *Ubi*, *Dbi*, and *Fbi*. They are boolean vectors of three elements with the initial value {false,false,false}. To simulate a button's pushing action (down-up), CNJ requires users to restore the button's value to *false* after setting it to *true*. For instance, if the Down button on the third floor is pushed once, the Dbi[3] element should be first set to *true*, then set to *false* right away. This value change is considered as button pushing by CNJ. After the CNJ simulation starts, the elevator's height value is continually displayed at the bottom StatusPane.

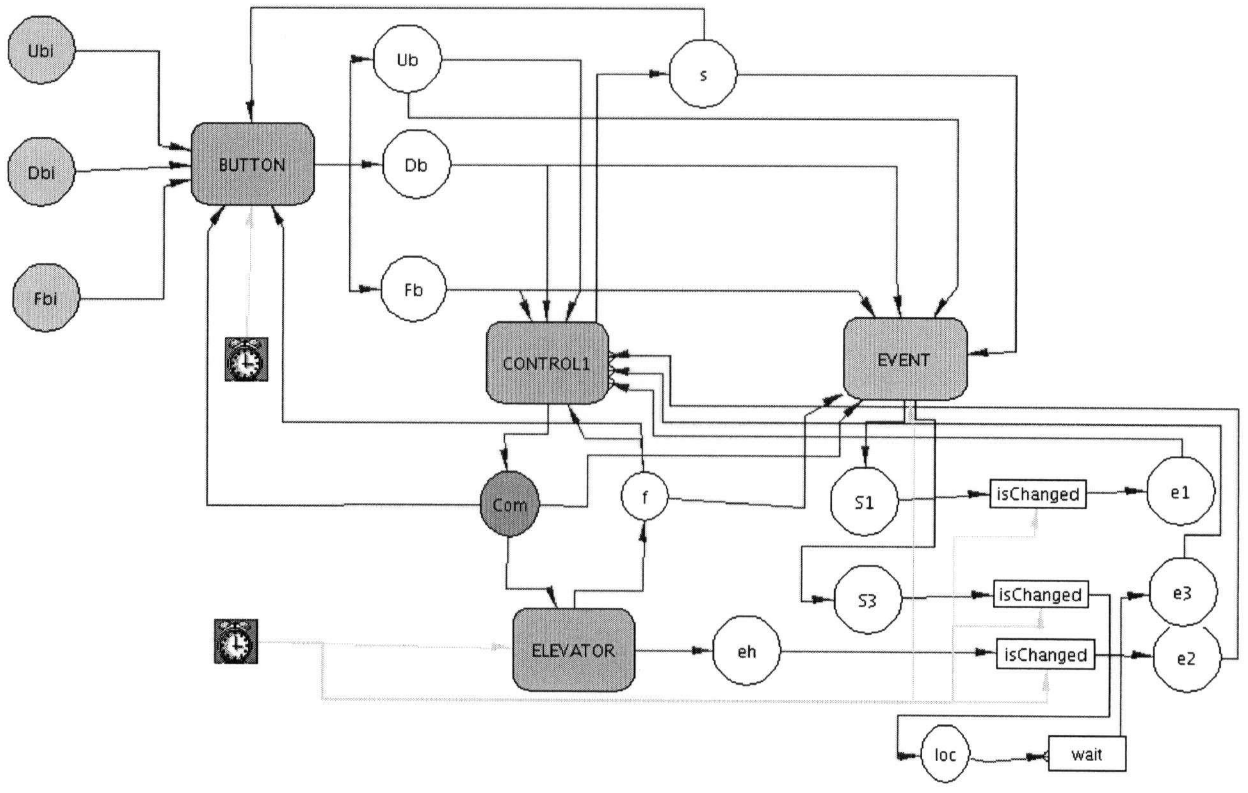


Figure 8.5. The Elevator system modeled in CNJ

Another way for the simulation is to utilize a two-dimensional animation window to display the simulation result. For the Elevator animation, we implemented a dedicated software package. It also works as a human-computer interface for interacting with the CNJ simulation. From the animation window in Figure 8.6, we can see that the left three buttons are the floor buttons inside the Elevator; each floor has two direction buttons (Up and Down). The floor buttons and the direction buttons can be pushed by clicking the mouse on them.

The animation window is running in a separate thread. Whenever a button is pushed, it is lighted as on, until the request is served. The states of buttons are transferred to the CNJ simulation. Then the CNJ simulation computes new results for every *location* while data is flowing along *connections*. For each clock tick, there is a newly computed value for the height of the elevator body. Meanwhile, the height of the elevator body is obtained by the animation thread in a specified frequency. Depending on the animation thread's frame speed, the elevator body is shown and refreshed frequently (generally, 5Hz).

8.4 Summary

The n-floor hybrid elevator system is modeled in constraint nets and works as a good example to demonstrate the performance of CNJ. It is a typical hybrid system with the elevator body as the continuous part and a discrete controller. The model designed by CNJ is based on the one depicted in Section 8.1, but is at a lower level. During the process of modeling, the elevator system, new functions and features are added into the environment. Furthermore, CNJ provides an animation package as a human computer interface to execute the simulation. It has successfully realized the modeling and simulation of the complex 3-floor hybrid elevator system, and works correctly.

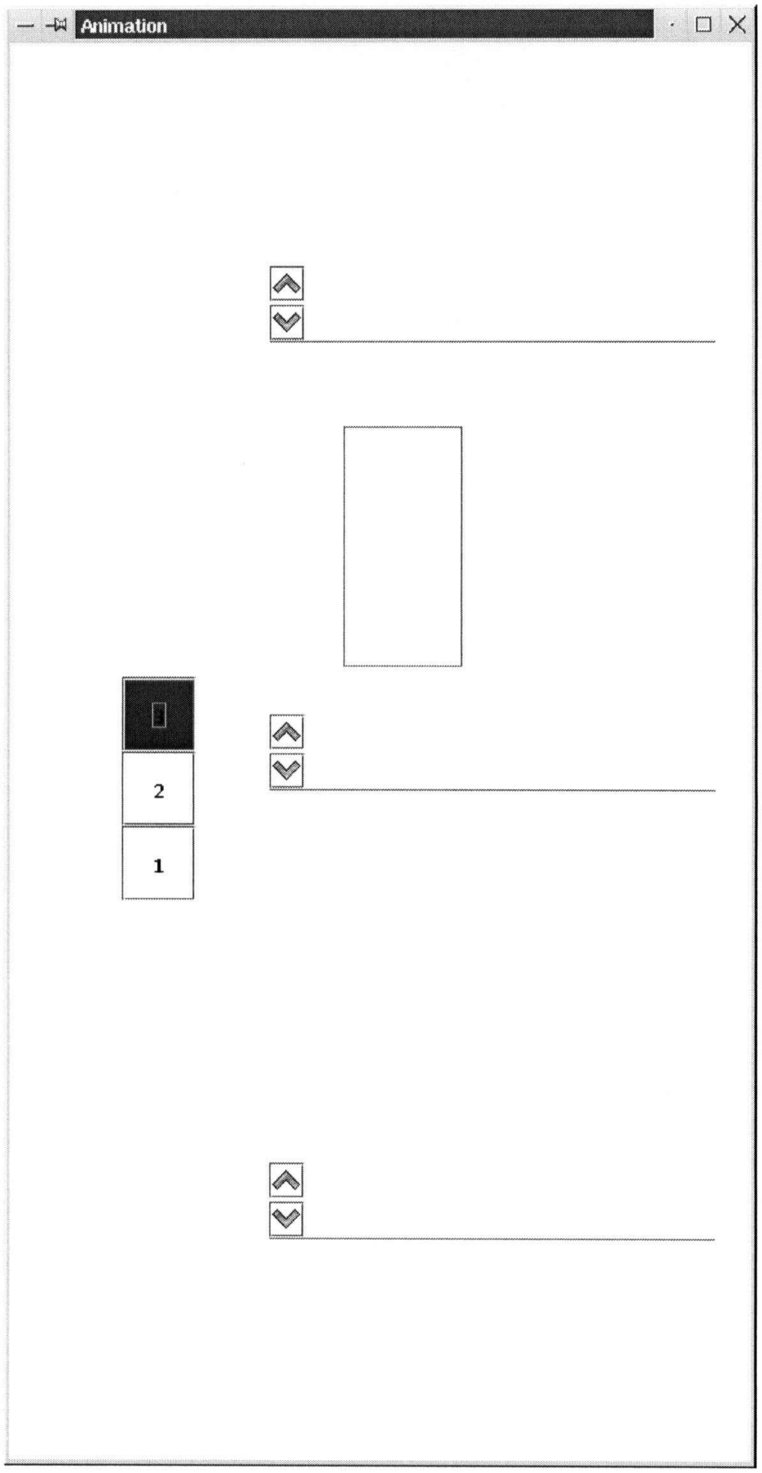


Figure 8.6 The Elevator animation window

CHAPTER 9 EXPERIMENTAL RESULTS

9.1 Methodology

Several experiments are carried out on the CNJ environment to analyze its performance. There are four examples developed. The first example is the simplest one, which has only one transduction (see Figure 9.1). The second one has two levels, and the lower level module is very simple (see Figure 9.2). The third one has dozens of nodes at one level (see Figure 9.3) [35]. It represents the dynamics of a car which is used as a radio-controlled mobile base for a robot soccer player. The last one is the most complicated model (Hybrid Elevator Control System illustrated in Chapter 8), which has 17 modules distributed in three levels, with 169 transductions and 2 global clocks (see Figure 8.5 in Chapter 8).

We generated different shapes and sizes of models in order to find CNJ's scalability. The shape and size of models can be defined by the parameters *depth* and *width*. *Depth* determines how many levels there are in the model. *Width* determines the number of nodes in each level. The above four examples have different *shapes* and *sizes* for conducting the experiments.

The experiments run under the environment of Linux Redhat 7.1 with PIII 1 G Hz CPU, 256M SDRAM, and Java Hotspot^{TM} Client VM (1.4.0, mixed mode). The code is programmed and compiled using J2SDK1.4.0. The performance of the CNJ simulation depends on the size of the model, clock frequency, and number of clocks in the model. Also it is dependent on its operating system's scheduling algorithm. These are called *factors*. CNJ is able to run simulations in speed from an arbitrarily low frequency (e.g., 0.1 Hz), up to what your computer is capable of (e.g., 50 Hz in our environment).

9.2 Results

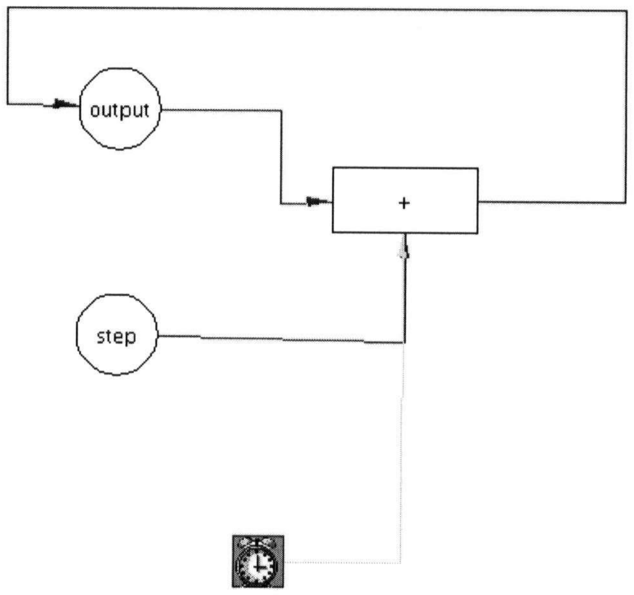


Figure 9.1. Example 1 (a simple adder)

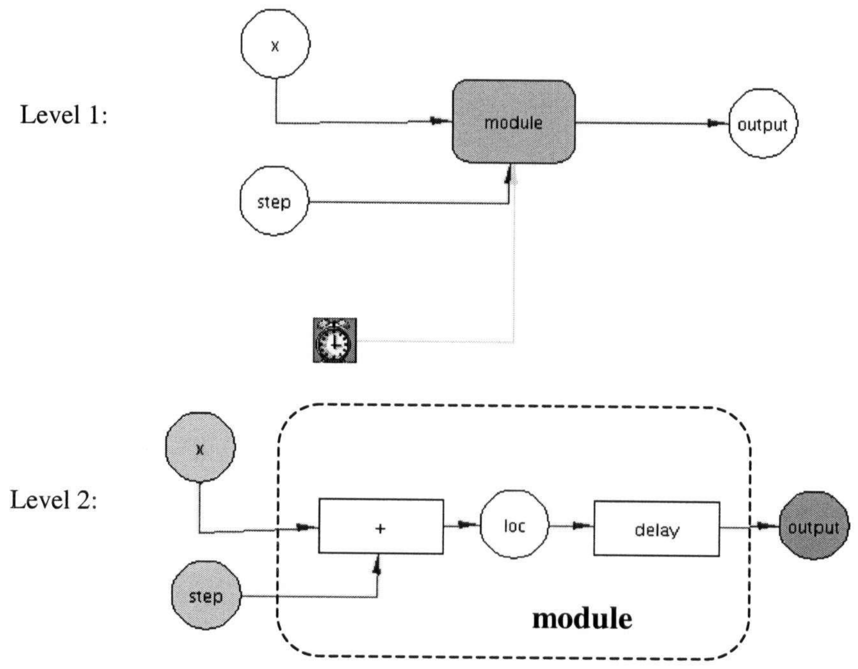


Figure 9.2. Example 2 (a two-level adder)

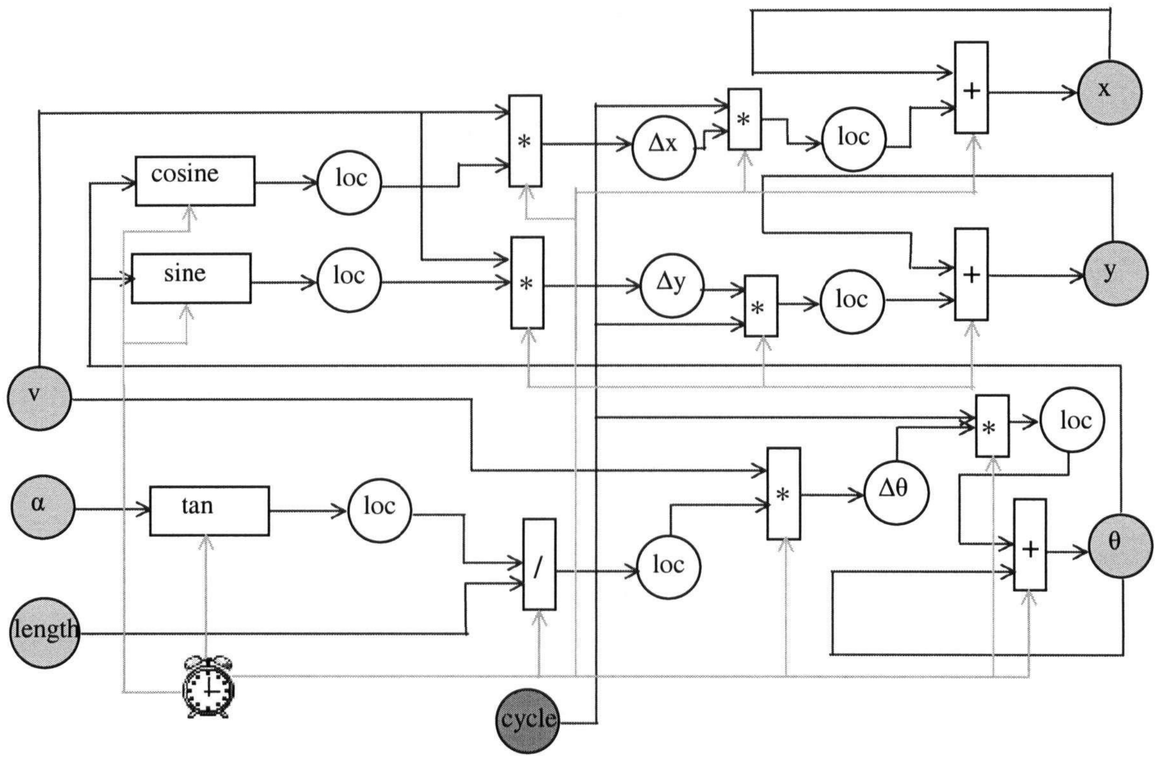


Figure 9.3. Example 3 (the Car Dynamics)

Since there is only one global CN Clock in the models of Example 1, Example 2, and Example 3, we put them in one table (Table 9.1) for comparison. Depending on the Clock's specified frequency, the output values are computed, and flow out in different speeds.

Clock frequency (Hz)	Average time cost per output (millisecond)		
	Example 1	Example 2	Example 3 (Car)
1	1010	1010	1010
10	110	110	110
20	60	60	60
50	20	20	20
100	20	20	20
1000	20	20	20

Table 9.1. Simulation results for Example 1, 2, 3

Chapter 8 described the hybrid 3-floor Elevator system modeled in CNJ. The Elevator model is used as Example 4, which has two abstract levels. One is the higher-level discrete controlling part. The other is the lower-level elevator body, running in a continuous domain. To let the elevator body virtually run in a continuous time structure, we used a faster Clock to drive the ELEVATOR and EVENT modules, and a slower one to drive the discrete BUTTON module. Assuming the slower clock is *Clock 1*, and the faster one *Clock 2*. Unlike other modules, the CONTROL1 module is event-based. Whenever there is a state change outside, it is triggered to run once. In Table 9.2, we assigned to the two clocks a variety of frequencies to test how more than one clock affects the CNJ's performance. This time the *output* specifically refers to the elevator's height from its body to the ground. BODY is a continuous module driven by Clock 2.

Clock frequency (Hz)		Average time cost per output of Height (millisecond)
Clock 1	Clock 2	Example 4 (The Elevator Control System)
1	10	110
1	20	60
1	50	20
10	10	110
10	20	60
10	50	25
20	20	60
20	50	30
50	50	40

Table 9.2. Simulation results for the Elevator system (Example 4)

From Table 9.1, we can see that the maximum speed that the simulation can run in our environment is 50 Hz. When the clock is set to 50 Hz, it fires clock events to its transductions to make them execute once per 20 ms (1000 / 50). However, when the clock's frequency is greater than 50 Hz, the output cannot be computed in a time slot less than 20 ms. The reason is that the operating system's scheduler has its own predefined latency for scheduling the running processes.

In Table 9.2, we find that if Clock 1 is not fast, the output speed is still dependent on Clock 2's frequency. However, when Clock 1's frequency is high enough, it greatly slows down the output speed. For instance, compared with Table 9.1 (Clock = 50 Hz, output speed = 20 ms), when Clock 2 = 50 Hz and Clock 1 = 10 Hz, the output speed becomes 25 ms (slower than 20 ms). Another example is that when Clock 2 = 50 Hz, Clock 1 = 50 Hz, the speed = 40 ms, which is much slower than 20 ms, as in Table 9.1. Therefore, we can draw the conclusion that the number of clocks affects the simulation speed more than the size of a CN model. The larger scalability results from the *Clock* class using the Java Swing *Timer* class.

In Java, the Swing *Timer* object automatically shares a thread to avoid spawning too many threads. It utilizes the same thread used to make cursors blink, tool tips appear, and so on. This method might lead to an event queue which may be too long for the CPU to process. To solve this problem, we need to implement a new Timer class which has its own dedicated thread. Through this means, different timers' clock events will be distributed in different threads, instead of one thread. Although it involves much more threads when running a CNJ simulation, a faster computer can easily solve this problem.

9.3 Summary

In our environment, CNJ is able to run at the speed of 50 Hz, close to real-time. From the experiments, the number of clocks gives a larger scalability than the model's size. The calculation of transductions is also executed after the correct dependency order is obtained by the transduction scheduling algorithm. Although CNJ's GUI is implemented in Java Swing, the UI response lag is small enough that it cannot be detected by users. Also, the performance of CNJ can be greatly improved if real-time operating systems and high-performance computers are adopted.

CHAPTER 10 CONCLUSIONS AND FUTURE WORK

10.1 Conclusions

CNJ is a visual programming environment for hybrid dynamic system modeling and simulation with constraint nets. Component-based technology (JavaBeans) is deployed to design and implement such an environment. Using a visual programming tool, users can easily model control systems by assembling a number of CN nodes together without knowing any programming languages, such as C/C++ and Java. In addition, CNJ is capable of developing programs with features of modularity, hierarchy, and reusability, saving designers a great deal of time and effort. It is useful for users to adopt the visual programming technique to model and simulate control systems.

The environment is written in pure Java, and the user interface is implemented by Java Swing. However, it runs fast, and its response lag is too low to be detected by users. Based on Java's Event mechanism, CNJ real-time simulation is built up successfully. Discrete, continuous, and event-based hybrid time structures are provided in CNJ. Also, the Transduction Scheduling Algorithm ensures that data flow satisfies the transductions' dependency relationship. Experimental results indicate that for clocks running at 50 Hz, the designed models can run close to real-time. This work demonstrates that Java language is useful for virtual real-time programming.

Constraint Net Markup Language (CNML) is developed for the first time as an XML-based interchange format for storing constraint nets. It makes translation to and from other file formats feasible and convenient. CNJ is the first practical specific environment to support constraint net modeling and simulation. We have modeled a variety of control systems successfully using CNJ. It works as proof of the feasibility of the efficient and useful implementation of the concept of Constraint Nets.

The current version of CNJ is definitely a prototype. In many ways, it is an experiment which uses component-based technology to build a visual programming environment. It is implemented in Java, and able to execute real-time simulations for hybrid models. The tool can also work as a useful practical programming environment for constraint net users and learners.

10.2 Future Work

The prototype of CNJ could be improved in many aspects, and there might be opportunities for future work. Besides the constraint net modeling language, CN includes timed $\forall$-automata as the requirement specification language, and a verification method to verify whether a constraint net satisfies the requirements. Future work, therefore, includes the extension of the support for graphical requirement specification and verification tools.

Until now, CNJ provides a limited set of basic functions to define transductions. Sometimes, a complex model may require new functions that are not provided by the function set. In addition, developers have to read code and modify some methods in a few classes. We propose to provide a functionality which allows users to add new functions automatically. During our future work of designing more hybrid systems, new useful functions and reusable modules would then be incorporated.

To improve real-time performance, much work needs to be done. This includes research on Real Time Specification Java (RTSJ 1.0 currently), adoption of real-time operating systems, real-time Java Virtual Machines [36], and better transduction scheduling algorithms.

Regarding the detailed implementation of CNJ, we hope to improve the compilation step to restrict users to design constraint nets with no syntax errors, and to generate better and clearer runtime error reports.

The graphical drawing of connections and transductions needs to be refined. In particular, transduction components need explicit ports with them. We would like better ways to draw CN connections more efficiently and easily.

Some operations would be helpful for future users, such as *copy*, *paste*, *undo*, *redo*, and *export to / import from* other file formats (HTML, Adobe Illustrator, Matlab MDL, ...). Also we hope to add additional support for storing each location's history traces in CNJ simulations.

For Constraint Net Markup Language (CNML), a DTD (Data Type Definition) or XSD (XML Schema Definition) can be added to define the legal building blocks of CNML files. We can also use XSL (eXtensible Stylesheet Language) to express CNML files in different styles in web browsers.

In this way CNJ could provide wider coverage and be more robust. Finally, there is an opportunity to design and simulate more challenging hybrid dynamic systems in constraint nets with CNJ.

BIBLIOGRAPHY

[1] S. Chang, *Principles of Visual Programming Systems.* New York: Prentice Hall, 1990.

[2] Brad A. Myers and Mary Beth Rosson, "Survey on User Interface Programming," presented at SIGCHI'92: Human Factors in Computing Systems, 1992.

[3] B. Sharhian and M. Hassul, *Control System Design Using Matlab*: Prentice-Hall Inc., 1993.

[4] Y. Zhang and A. K. Mackworth, "Constraint Programming in Constraint Nets," presented at First Workshop on Principles and Practice of Constraint Programming, 1993.

[5] John Lygeros, Datta N. Godbole, and Shankar S. Sastry, "Simulation as a Tool for Hybrid System Design," presented at Fifth IEEE conference on AI, Simulation and Planning in High-Autonomy Systems, 1994.

[6] Y. Zhang and A. K. Mackworth, "Specification and Verification of Constraint-Based Dynamic Systems," in *Principles and Practice of Constraint Programming, No. 874 in Lecture Notes in Computer Science*, A. Borning, Ed.: Springer-Verlag, 1994, pp. 229-242.

[7] Ying Zhang, "A Foundation for the Design and Analysis of Robotic Systems and Behaviors, Ph.D. thesis," in *Department of Computer Science.* Vancouver: University of British Columbia, 1994.

[8] Margaret Burnett, "Scaling Up Visual Programming Languages," presented at IEEE Computer, 1995.

[9] E. Jackson, *ACE For Windows NT Prototype Design Document*: International Submarine Engineering Ltd, 1995.

[10] Y. Zhang and A. K. Mackworth, "Synthesis of Hybrid Constraint-Based Controllers," in *Hybrid Systems II, Lecture Notes in Computer Science 999*, W. K. P. Antaklis, A. Nerode, S. Sastry, Ed.: Springer Verlag, 1995, pp. 552-567.

[11] R. Jagannathan, "Dataflow Models," in *Parallel and Distributed Computer Handbook*, A. Y. Zomaya, Ed.: McGraw-Hill, Inc., 1996.

[12] Goddard Space Flight Center of NASA, "User-Interface Guidelines," 1996.

[13] Marat Boshernitsan and Michael Downes, "Visual Programming Languages: A Survey," University of California, Berkeley 1997.

[14] Sun Microsystems, *JavaBeans specification version 1.01*, 1997.

[15] Y.Fukazawa, "Cooperation Between Methodology and its Support Tools in Component-based Software Development," *Waseda University Journal of Science and Engineering*, vol. 1, 1997.

[16] Hessam S. Sarjoughian and Bernard P. Zeigler, "DEVSJava: Basis for a DEVS-based Collaborative M&S Environment," presented at INTERNATIONAL CONFERENCE ON WEB-BASED MODELING & SIMULATION, 1998.

[17] John A. Miller, Y. Ge, and J. Tao, "Component-Based Simulation Environments: JSIM As a Case Study Using Java Beans," presented at Winter Simulation Conference, 1998.

[18] Wim Groenendaal, "Component Technology," in *Xootic Magazine*, 1999.

[19] H. Praehofer, J. Sametinger, and A. Stritzinger, "Discrete Event Simulation using the JavaBeans Component Model," presented at International Conference On Web-Based Modelling & Simulation 1999, 1999.

[20] M. Pidd, N. Oses, and R. J. Brooks, "Component-Based Simulation On The Web," presented at 1999 Winter Simulation Conference, 1999.

[21] Y. Zhang and A. K. Mackworth, "Modeling and Analysis of Hybrid Systems: An Elevator Study," in *Logical Foundation for Cognitive Agents*, F. P. H. Levesque, Ed. Berlin: Springer, 1999, pp. 370-396.

[22] M. Jungel, E. Kindeler, and M. Weber, "The Petri Net Markup Language," presented at Workshop Algorithm for Petri Net, 2000.

[23] A. K. Mackworth, "Constraint-Based Agents: The ABC's of CBA's," presented at 6th Int. Conf. On Principles and Practice of Constraint Programming - CP2000, Singapore, 2000.

[24] "JDOM Beta 7," 2001.

[25] A. K. Mackworth and Y. Zhang, "Constraint-Based Agents: A Formal Model for Agent Design," in *UBC Computer Science Technical Report TR 2001-09*, 2001.

[26] Guillem Bernat and Alan Burns, "Weakly Hard Real-Time Systems," *IEEE Transactions on Computers*, vol. 50, 2001.

[27] E.R. Harold, *XML Bible 2nd edition*: IDB Books, 2001.

[28] K. Verschaeve, B. Wydaeghe, and F. Westerhuis, "Visual Composition with SDL Beans," presented at ECBS 2001, Washington, USA, 2001.

[29] Sun Microsystems, "Java Look and Feel Design Guidelines (2nd edition)," 2001.

[30] Andy Quinn, "Trail: JavaBeans (TM)," 2001.

[31] W3C, *Scalable Vector Graphics (SVG) 1.0 Specification*, 2001.

[32] W3C, *Mathematical Markup Language (MathML) Version 2.0*, 2001.

[33] Y. Wang and S. Ho., "Implementation of a DEVS-JavaBean Simulation Environment," presented at Advanced Simulation Technologies Conference, 2001.

[34] "Petri Net Kernel," Research Group Petri Net Technology, 2002.

[35] Fengguang Song and Alan K. Mackworth, "CNJ: A Visual Programming Environment for Constraint Nets," presented at AI, Simulation and Planning in High Autonomy Systems, Lisbon, Portugal, 2002.

[36] M. Pfeffer, S. Uhrig, Th. Ungerer, and U. Brinkschulte, "A Real-Time Java System on a Multithreaded Java Microcontroller," presented at Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2002.

[37] Sun Microsystems, "The Only Component Architecture for Java Technology," 2002.

[38] Clark Williams, "Linux Scheduler Latency," Red Hat Inc., 2002.

APPENDIX A CNML FILE FOR THE CAR DYNAMICS MODEL

```
<?xml version="1.0" encoding="UTF-8"?>
<cn_model id="cn">
  <name>Car Dynamics</name>
  <transduction id="0">
    <graphics>
      <rect x="150" y="326" width="90" height="40" fill="-1" stroke="-
16777216" />
    </graphics>
    <label color="-16776961">tan</label>
    <function>
      <input number="1" />
      <math>
        <apply>
          <tan />
          <ci type="1">input</ci>
        </apply>
      </math>
    </function>
  </transduction>
  <transduction id="1">
    <graphics>
      <rect x="383" y="347" width="19" height="86" fill="-1" stroke="-
16777216" />
    </graphics>
    <label color="-16776961">/</label>
    <function>
      <input number="2" />
      <math>
        <apply>
          <divide />
          <ci type="1">input</ci>
          <ci type="1">input</ci>
        </apply>
      </math>
    </function>
  </transduction>
  <transduction id="2">
    <graphics>
      <rect x="537" y="300" width="19" height="98" fill="-1" stroke="-
16777216" />
    </graphics>
    <label color="-16776961">*</label>
    <function>
      <input number="2" />
      <math>
        <apply>
          <times />
          <ci type="1">input</ci>
          <ci type="1">input</ci>
        </apply>
      </math>
    </function>
  </transduction>
  <transduction id="3">
    <graphics>
      <rect x="739" y="373" width="22" height="78" fill="-1" stroke="-
16777216" />
    </graphics>
    <label color="-16776961">+</label>
    <function>
      <input number="2" />
      <math>
        <apply>
          <plus />
          <ci type="1">input</ci>
          <ci type="1">input</ci>
        </apply>
      </math>
    </function>
  </transduction>
  <transduction id="4">
    <graphics>
      <rect x="152" y="185" width="90" height="40" fill="-1" stroke="-
16777216" />
    </graphics>
    <label color="-16776961">sine</label>
    <function>
      <input number="1" />
      <math>
        <apply>
          <sine />
          <ci type="1">input</ci>
        </apply>
      </math>
    </function>
  </transduction>
  <transduction id="5">
    <graphics>
      <rect x="153" y="128" width="90" height="40" fill="-1" stroke="-
16777216" />
    </graphics>
    <label color="-16776961">cosine</label>
    <function>
      <input number="1" />
      <math>
        <apply>
          <cosine />
          <ci type="1">input</ci>
        </apply>
      </math>
    </function>
  </transduction>
  <transduction id="6">
    <graphics>
      <rect x="373" y="82" width="19" height="63" fill="-1" stroke="-
16777216" />
    </graphics>
    <label color="-16776961">*</label>
    <function>
      <input number="2" />
      <math>
        <apply>
          <times />
          <ci type="1">input</ci>
          <ci type="1">input</ci>
        </apply>
      </math>
    </function>
  </transduction>
  <transduction id="7">
    <graphics>
      <rect x="377" y="171" width="21" height="66" fill="-1" stroke="-
16777216" />
    </graphics>
    <label color="-16776961">*</label>
    <function>
      <input number="2" />
      <math>
        <apply>
          <times />
          <ci type="1">input</ci>
          <ci type="1">input</ci>
        </apply>
      </math>
    </function>
  </transduction>
  <transduction id="8">
    <graphics>
      <rect x="667" y="15" width="15" height="63" fill="-1" stroke="-
16777216" />
    </graphics>
    <label color="-16776961">+</label>
    <function>
      <input number="2" />
      <math>
        <apply>
          <plus />
          <ci type="1">input</ci>
          <ci type="1">input</ci>
        </apply>
      </math>
    </function>
  </transduction>
  <transduction id="9">
    <graphics>
      <rect x="674" y="134" width="13" height="66" fill="-1" stroke="-
16777216" />
    </graphics>
    <label color="-16776961">+</label>
    <function>
      <input number="2" />
      <math>
        <apply>
          <plus />
          <ci type="1">input</ci>
          <ci type="1">input</ci>
        </apply>
      </math>
    </function>
  </transduction>
  <transduction id="10">
    <graphics>
      <rect x="561" y="84" width="20" height="56" fill="-1" stroke="-
16777216" />
    </graphics>
    <label color="-16776961">*</label>
    <function>
      <input number="2" />
      <math>
        <apply>
          <times />
          <ci type="1">input</ci>
          <ci type="1">input</ci>
        </apply>
      </math>
    </function>
  </transduction>
  <transduction id="11">
    <graphics>
      <rect x="561" y="174" width="20" height="55" fill="-1" stroke="-
16777216" />
    </graphics>
    <label color="-16776961">*</label>
    <function>
      <input number="2" />
      <math>
        <apply>
          <times />
          <ci type="1">input</ci>
          <ci type="1">input</ci>
        </apply>
      </math>
    </function>
  </transduction>
  <transduction id="12">
    <graphics>
      <rect x="690" y="327" width="21" height="48" fill="-1" stroke="-
16777216" />
    </graphics>
    <label color="-16776961">*</label>
    <function>
      <input number="2" />
      <math>
        <apply>
          <times />
          <ci type="1">input</ci>
          <ci type="1">input</ci>
        </apply>
      </math>
    </function>
  </transduction>
  <location id="13">
```

```
    <graphics>
      <ellipse cx="91" cy="343" rx="25" ry="25" fill="-3355393"
stroke="-16777216" />
    </graphics>
    <label color="-16776961">alpha</label>
    <ci type="1" />
  </location>
  <location id="14">
    <graphics>
      <ellipse cx="311" cy="347" rx="25" ry="25" fill="-1" stroke="-
16777216" />
    </graphics>
    <label color="-16776961">loc</label>
    <ci type="1" />
  </location>
  <location id="15">
    <graphics>
      <ellipse cx="243" cy="431" rx="25" ry="25" fill="-6684928"
stroke="-16777216" />
    </graphics>
    <label color="-16776961">length</label>
    <ci type="1" />
  </location>
  <location id="16">
    <graphics>
      <ellipse cx="476" cy="389" rx="25" ry="25" fill="-1" stroke="-
16777216" />
    </graphics>
    <label color="-16776961">loc</label>
    <ci type="1" />
  </location>
  <location id="17">
    <graphics>
      <ellipse cx="90" cy="272" rx="25" ry="25" fill="-3355393"
stroke="-16777216" />
    </graphics>
    <label color="-16776961">V</label>
    <ci type="1" />
  </location>
  <location id="18">
    <graphics>
      <ellipse cx="631" cy="350" rx="33" ry="23" fill="-1" stroke="-
16777216" />
    </graphics>
    <label color="-16776961">delta-theta</label>
    <ci type="1" />
  </location>
  <location id="19">
    <graphics>
      <ellipse cx="851" cy="407" rx="23" ry="20" fill="-26368"
stroke="-16777216" />
    </graphics>
    <label color="-16776961">theta</label>
    <ci type="1" />
  </location>
  <location id="20">
    <graphics>
```

```
      <ellipse cx="314" cy="205" rx="25" ry="25" fill="-1" stroke="-
16777216" />
    </graphics>
    <label color="-16776961">loc</label>
    <ci type="1" />
  </location>
  <location id="21">
    <graphics>
      <ellipse cx="315" cy="149" rx="25" ry="25" fill="-1" stroke="-
16777216" />
    </graphics>
    <label color="-16776961">loc</label>
    <ci type="1" />
  </location>
  <location id="22">
    <graphics>
      <ellipse cx="501" cy="113" rx="25" ry="25" fill="-1" stroke="-
16777216" />
    </graphics>
    <label color="-16776961">delta-x</label>
    <ci type="1" />
  </location>
  <location id="23">
    <graphics>
      <ellipse cx="503" cy="204" rx="25" ry="25" fill="-1" stroke="-
16777216" />
    </graphics>
    <label color="-16776961">delta-y</label>
    <ci type="1" />
  </location>
  <location id="24">
    <graphics>
      <ellipse cx="753" cy="50" rx="25" ry="25" fill="-26368" stroke="-
16777216" />
    </graphics>
    <label color="-16776961">output</label>
    <ci type="1" />
  </location>
  <location id="25">
    <graphics>
      <ellipse cx="759" cy="168" rx="21" ry="22" fill="-26368"
stroke="-16777216" />
    </graphics>
    <label color="-16776961">y</label>
    <ci type="1" />
  </location>
  <location id="26">
    <graphics>
      <ellipse cx="481" cy="470" rx="35" ry="17" fill="-10027213"
stroke="-16777216" />
    </graphics>
    <label color="-16776961">clock cycle</label>
    <ci type="1" />
  </location>
  <location id="27">
    <graphics>
```

```
      <ellipse cx="624" cy="93" rx="13" ry="12" fill="-1" stroke="-
16777216" />
    </graphics>
    <label color="-16776961">loc</label>
    <ci type="1" />
  </location>
  <location id="28">
    <graphics>
      <ellipse cx="625" cy="201" rx="13" ry="11" fill="-1" stroke="-
16777216" />
    </graphics>
    <label color="-16776961">loc</label>
    <ci type="1" />
  </location>
  <location id="29">
    <graphics>
      <ellipse cx="684" cy="397" rx="15" ry="12" fill="-1" stroke="-
16777216" />
    </graphics>
    <label color="-16776961">loc</label>
    <ci type="1" />
  </location>
  <clock id="30">
    <graphics>
      <rect x="178" y="489" width="32" height="32" fill="-16711936"
stroke="-65536" />
    </graphics>
    <delay>1000</delay>
  </clock>
  <connection id="31">
    <polyline points="91,343 195,346" />
    <source>13</source>
    <target>0</target>
  </connection>
  <connection id="32">
    <polyline points="195,346 311,347" />
    <source>0</source>
    <target>14</target>
  </connection>
  <connection id="33">
    <polyline points="311,347 353,347 353,381 392,390" />
    <source>14</source>
    <target>1</target>
  </connection>
  <connection id="34">
    <polyline points="243,431 352,430 352,397 392,390" />
    <source>15</source>
    <target>1</target>
  </connection>
  <connection id="35">
    <polyline points="392,390 476,389" />
    <source>1</source>
    <target>16</target>
  </connection>
  <connection id="36">
    <polyline points="90,272 494,272 494,335 546,349" />
    <source>17</source>
    <target>2</target>
  </connection>
  <connection id="37">
    <polyline points="476,389 516,389 516,356 546,349" />
    <source>16</source>
    <target>2</target>
  </connection>
  <connection id="38">
    <polyline points="546,349 631,350" />
    <source>2</source>
    <target>18</target>
  </connection>
  <connection id="39">
    <polyline points="194,505 195,346" />
    <source>30</source>
    <target>0</target>
  </connection>
  <connection id="40">
    <polyline points="194,505 392,505 392,390" />
    <source>30</source>
    <target>1</target>
  </connection>
  <connection id="41">
    <polyline points="194,505 546,505 546,349" />
    <source>30</source>
    <target>2</target>
  </connection>
  <connection id="42">
    <polyline points="750,412 851,407" />
    <source>3</source>
    <target>19</target>
  </connection>
  <connection id="43">
    <polyline points="194,505 194,538 750,538 750,412" />
    <source>30</source>
    <target>3</target>
  </connection>
  <connection id="44">
    <polyline points="851,407 849,265 849,245 109,245 109,146 198,148"
/>
    <source>19</source>
    <target>5</target>
  </connection>
  <connection id="45">
    <polyline points="851,407 849,235 130,235 130,204 197,205" />
    <source>19</source>
    <target>4</target>
  </connection>
  <connection id="46">
    <polyline points="198,148 315,149" />
    <source>5</source>
    <target>21</target>
  </connection>
  <connection id="47">
    <polyline points="197,205 314,205" />
    <source>4</source>
    <target>20</target>
  </connection>
  <connection id="48">
    <polyline points="90,272 90,86 358,86 358,101 382,113" />
    <source>17</source>
    <target>6</target>
  </connection>
  <connection id="49">
    <polyline points="315,149 315,113 382,113" />
    <source>21</source>
    <target>6</target>
  </connection>
  <connection id="50">
    <polyline points="90,272 90,103 342,103 342,179 358,179 358,190
387,204" />
    <source>17</source>
    <target>7</target>
  </connection>
  <connection id="51">
    <polyline points="314,205 387,204" />
    <source>20</source>
    <target>7</target>
  </connection>
  <connection id="52">
    <polyline points="382,113 501,113" />
    <source>6</source>
    <target>22</target>
  </connection>
  <connection id="53">
    <polyline points="387,204 503,204" />
    <source>7</source>
    <target>23</target>
  </connection>
  <connection id="54">
    <polyline points="194,505 120,505 120,183 204,183 198,148" />
    <source>30</source>
    <target>5</target>
  </connection>
  <connection id="55">
    <polyline points="194,505 138,505 138,295 193,295 197,205" />
    <source>30</source>
    <target>4</target>
  </connection>
  <connection id="56">
    <polyline points="194,505 425,505 425,157 381,157 382,113" />
    <source>30</source>
    <target>6</target>
  </connection>
  <connection id="57">
    <polyline points="194,505 413,505 413,299 387,299 387,204" />
    <source>30</source>
    <target>7</target>
  </connection>
  <connection id="58">
    <polyline points="674,46 753,50" />
    <source>8</source>
    <target>24</target>
  </connection>
```

```
    <connection id="59">
      <polyline points="194,505 210,511 734,511 734,97 670,97 674,46" />
      <source>30</source>
      <target>8</target>
    </connection>
    <connection id="60">
      <polyline points="194,505 721,505 721,220 677,220 680,167" />
      <source>30</source>
      <target>9</target>
    </connection>
    <connection id="61">
      <polyline points="753,50 801,47 801,7 608,7 608,46 674,46" />
      <source>24</source>
      <target>8</target>
    </connection>
    <connection id="62">
      <polyline points="759,168 808,173 808,120 609,120 609,166 680,167"
  />
      <source>25</source>
      <target>9</target>
    </connection>
    <connection id="63">
      <polyline points="680,167 759,168" />
      <source>9</source>
      <target>25</target>
    </connection>
    <connection id="64">
      <polyline points="851,407 849,466 631,466 631,411 750,412" />
      <source>19</source>
      <target>3</target>
    </connection>
    <connection id="65">
      <polyline points="501,113 571,112" />
      <source>22</source>
      <target>10</target>
    </connection>
    <connection id="66">
      <polyline points="503,204 571,201" />
      <source>23</source>
      <target>11</target>
    </connection>
    <connection id="67">
      <polyline points="631,350 700,351" />
      <source>18</source>
      <target>12</target>
    </connection>
    <connection id="68">
      <polyline points="481,470 601,470 601,385 671,385 671,356 700,351"
  />
      <source>26</source>
      <target>12</target>
    </connection>
    <connection id="69">
      <polyline points="481,470 575,470 575,255 539,255 539,122 571,112"
  />
      <source>26</source>
      <target>10</target>
  </connection>
  <connection id="70">
    <polyline points="481,470 565,470 565,280 546,280 546,213 571,201"
/>
    <source>26</source>
    <target>11</target>
  </connection>
  <connection id="71">
    <polyline points="571,112 592,112 592,92 624,93" />
    <source>10</source>
    <target>27</target>
  </connection>
  <connection id="72">
    <polyline points="624,93 642,93 642,51 674,46" />
    <source>27</source>
    <target>8</target>
  </connection>
  <connection id="73">
    <polyline points="571,201 625,201" />
    <source>11</source>
    <target>28</target>
  </connection>
  <connection id="74">
    <polyline points="625,201 648,201 648,174 680,167" />
    <source>28</source>
    <target>9</target>
  </connection>
  <connection id="75">
    <polyline points="700,351 724,351 724,378 611,378 611,396 684,397"
/>
    <source>12</source>
    <target>29</target>
  </connection>
  <connection id="76">
    <polyline points="684,397 724,397 750,412" />
    <source>29</source>
    <target>3</target>
  </connection>
  <connection id="77">
    <polyline points="194,505 589,505 589,154 571,154 571,112" />
    <source>30</source>
    <target>10</target>
  </connection>
  <connection id="78">
    <polyline points="194,505 588,505 588,250 571,250 571,201" />
    <source>30</source>
    <target>11</target>
  </connection>
  <connection id="79">
    <polyline points="194,505 704,505 704,402 703,390 700,351" />
    <source>30</source>
    <target>12</target>
  </connection>
</cn_model>
```

APPENDIX B THE ELEVATOR SYSTEM MODELED IN CNJ

This appendix contains the complete graphical modules for the 3-floor Elevator control system modeled by CNJ. The top level is shown in Figure 8.5.

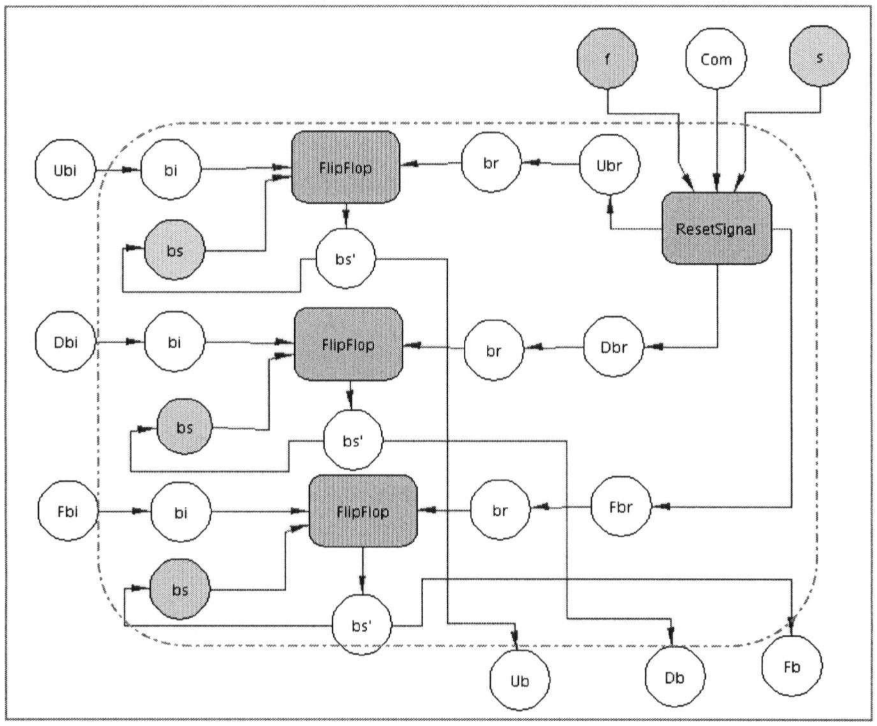


Figure B-1. BUTTON Module

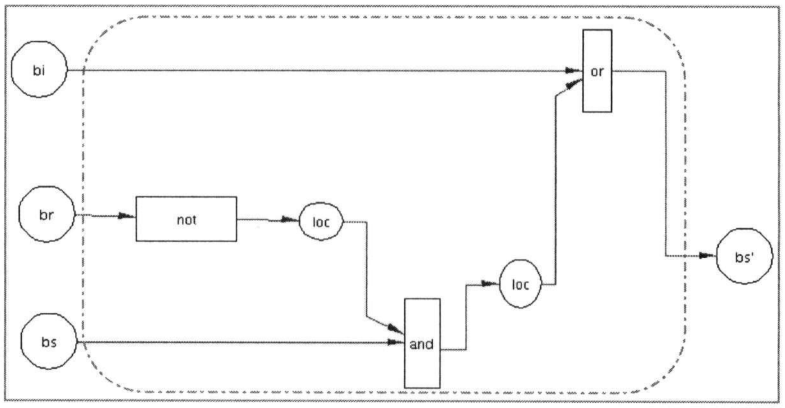
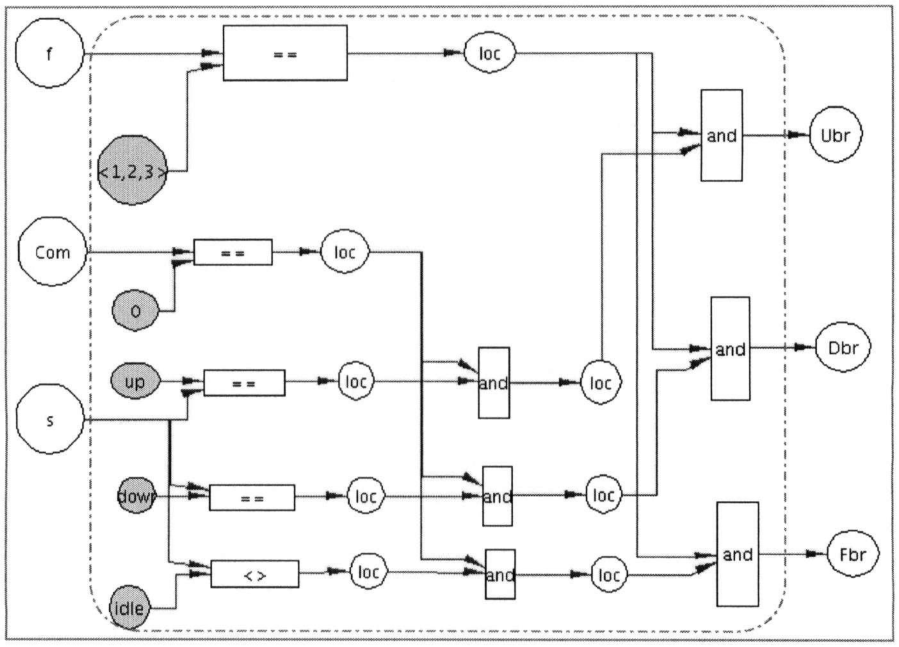


Figure B-2. FlipFlop Module



Fugure B-3. ResetSignal Module

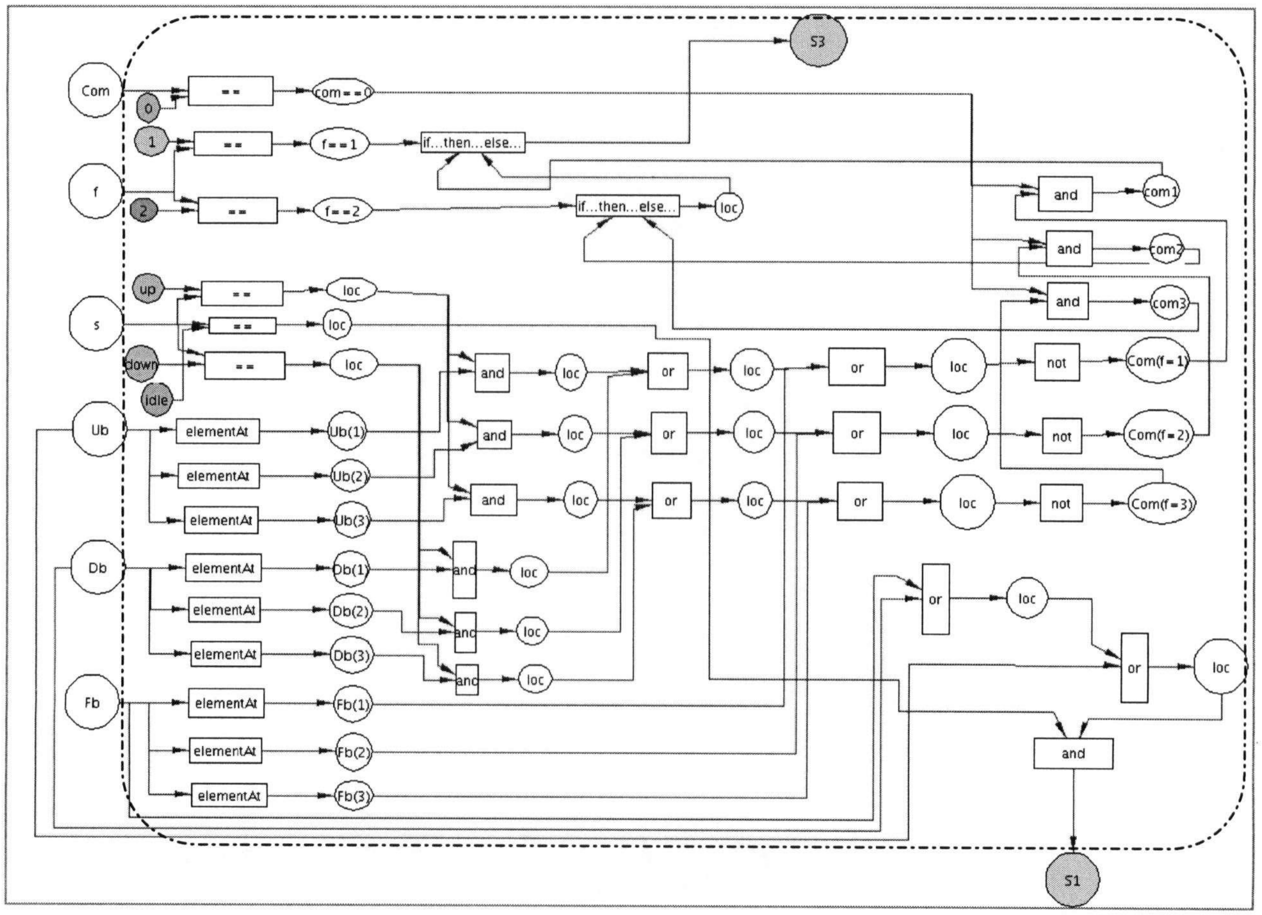


Figure B-4. EVENT Module

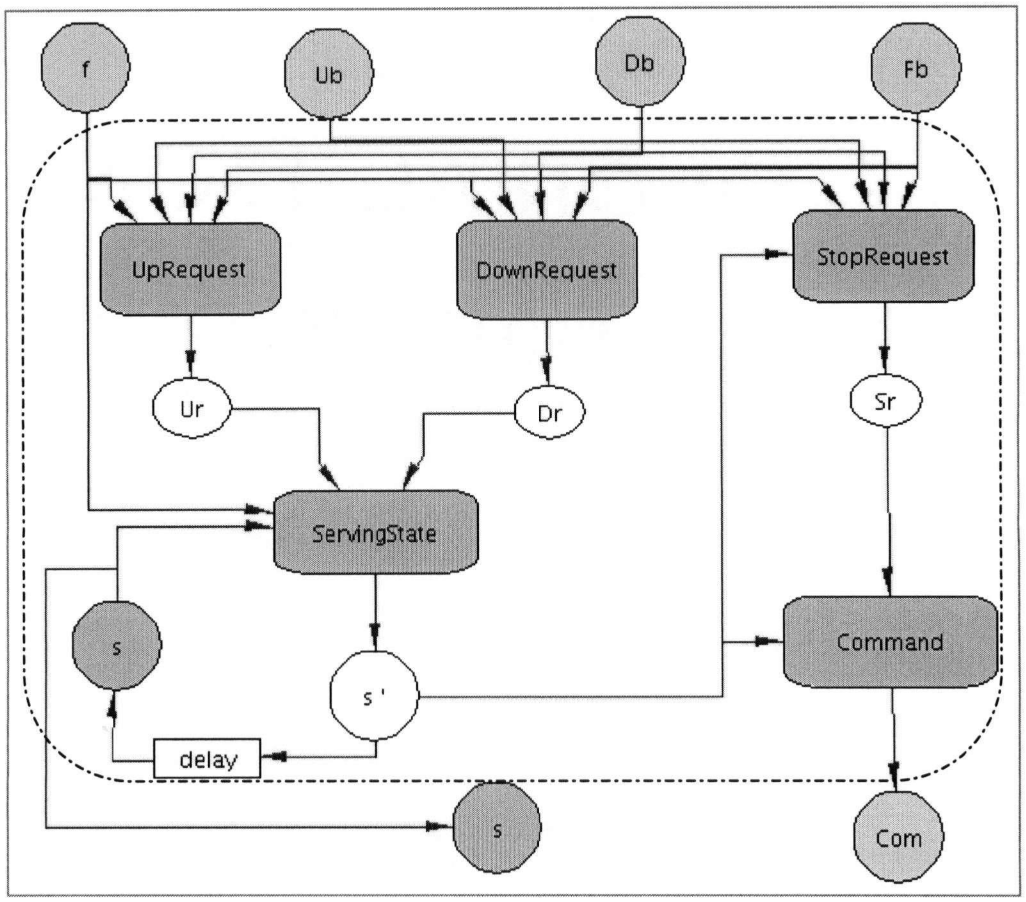


Figure B-5. CONTROL1 Module

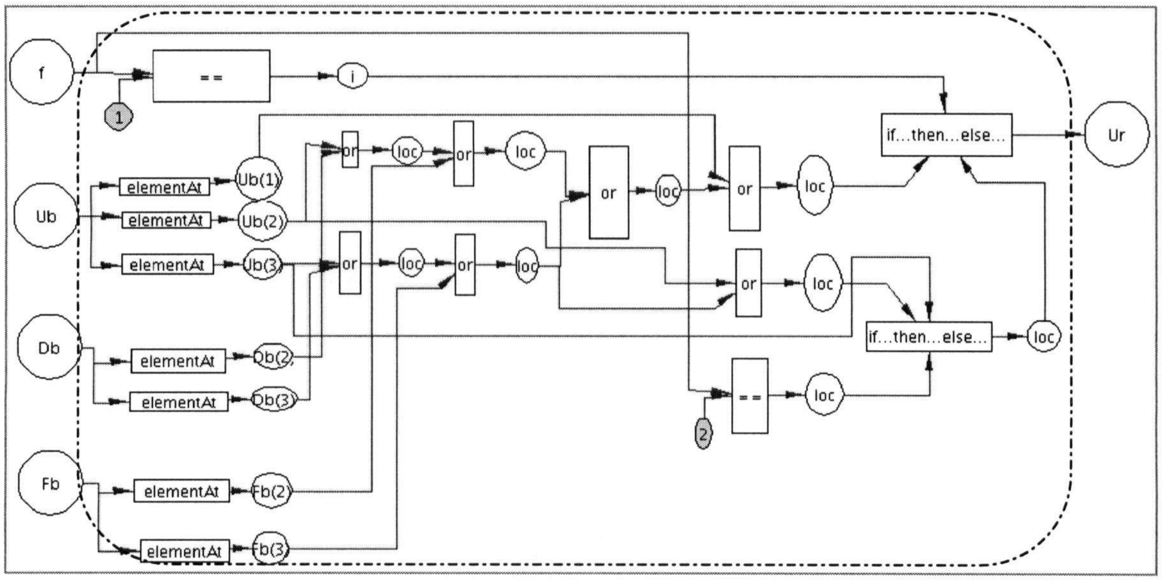


Figure B-6. UpRequest Module

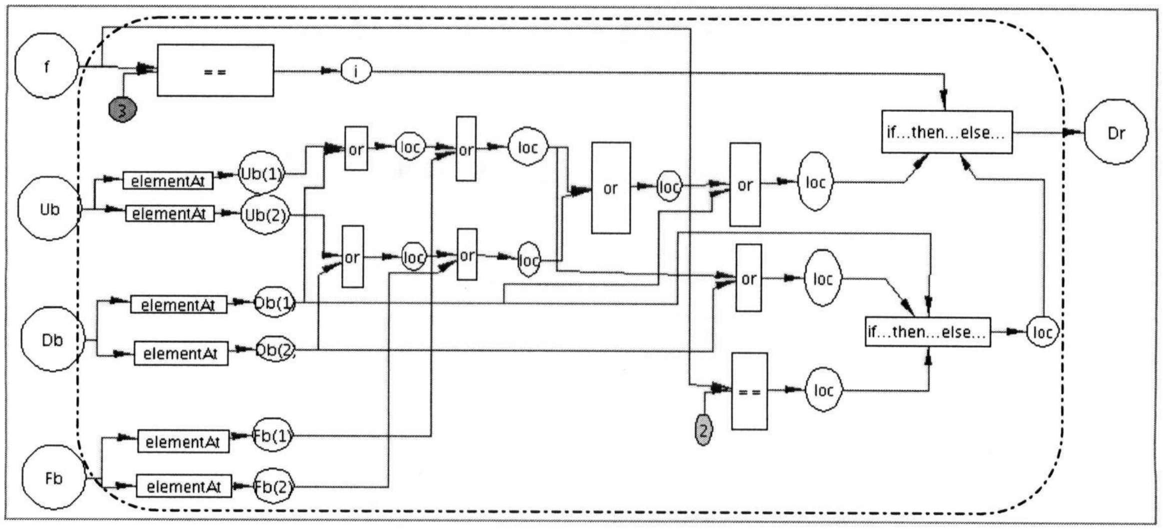


Figure B-7. DownRequest Module

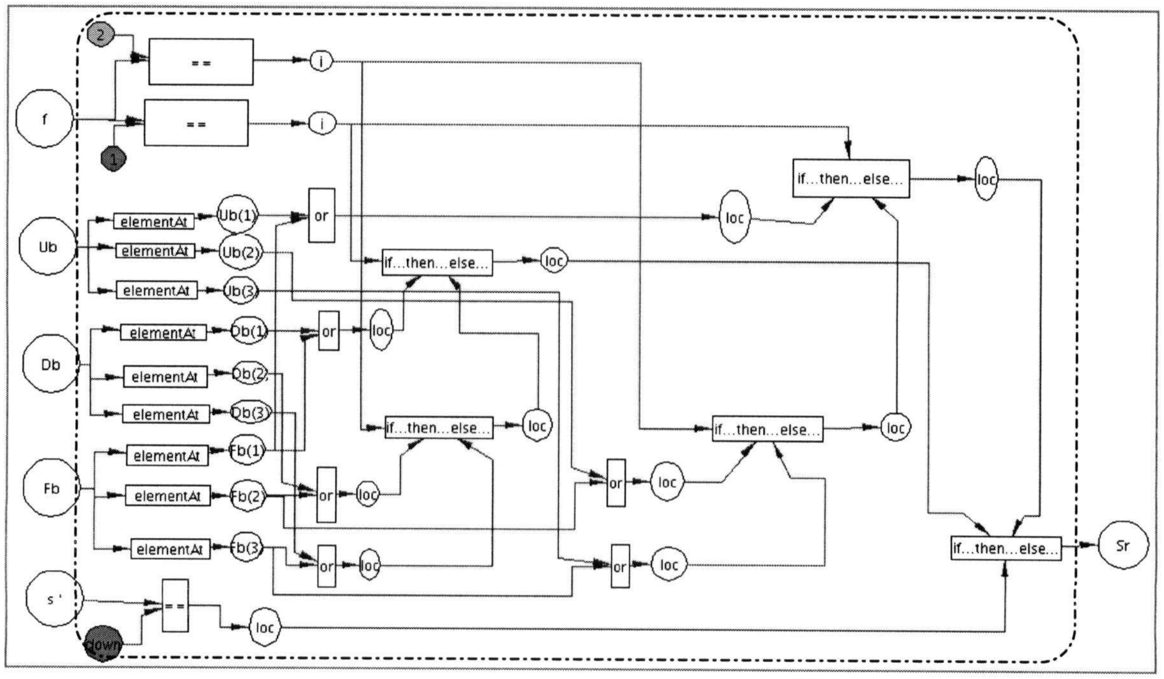


Figure B-8. StopRequest Module

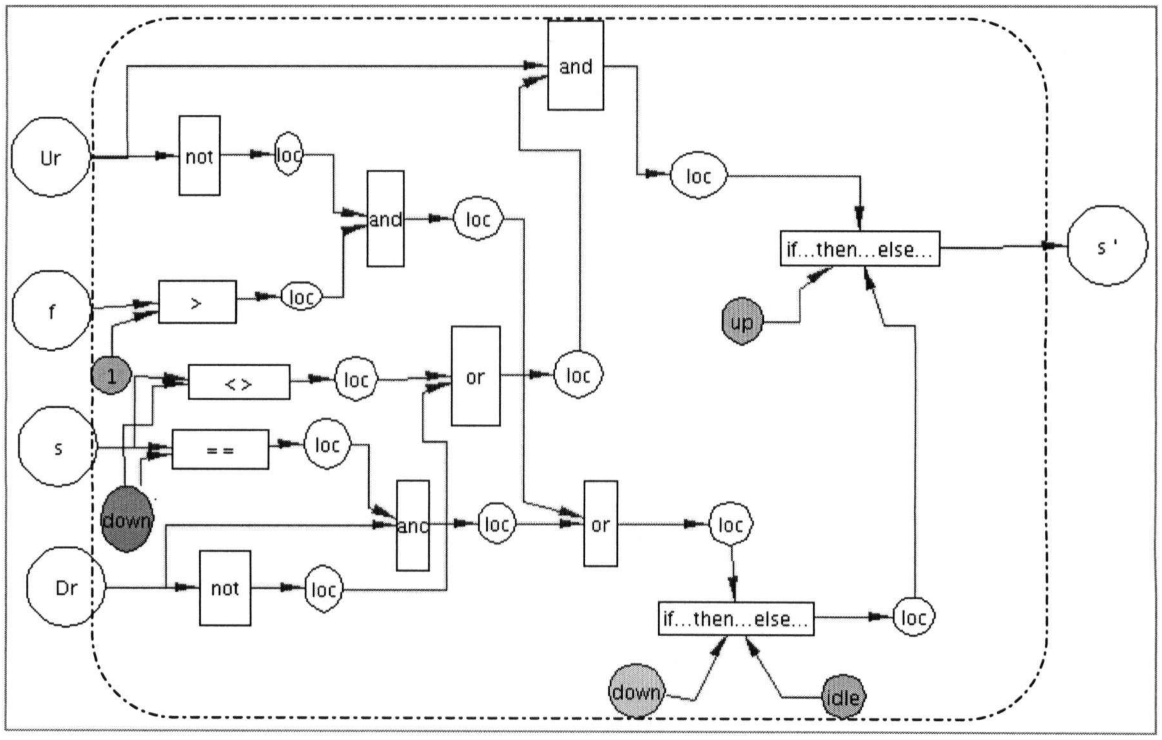


Figure B-9. ServingState Module

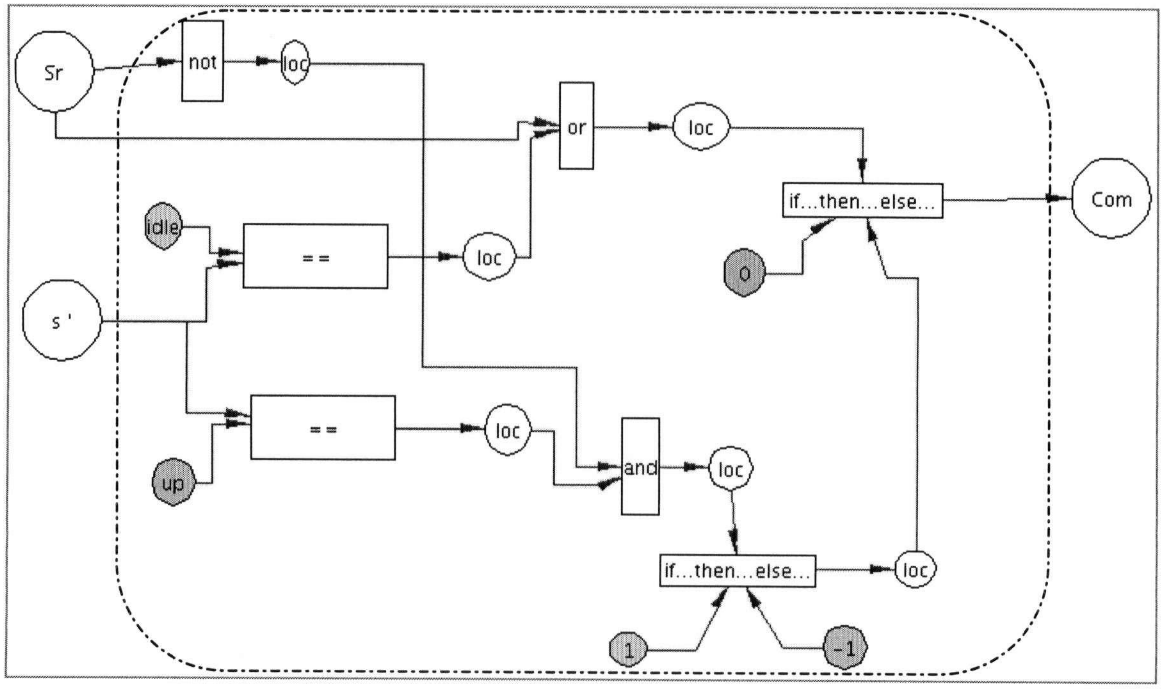


Figure B-10. Command Module

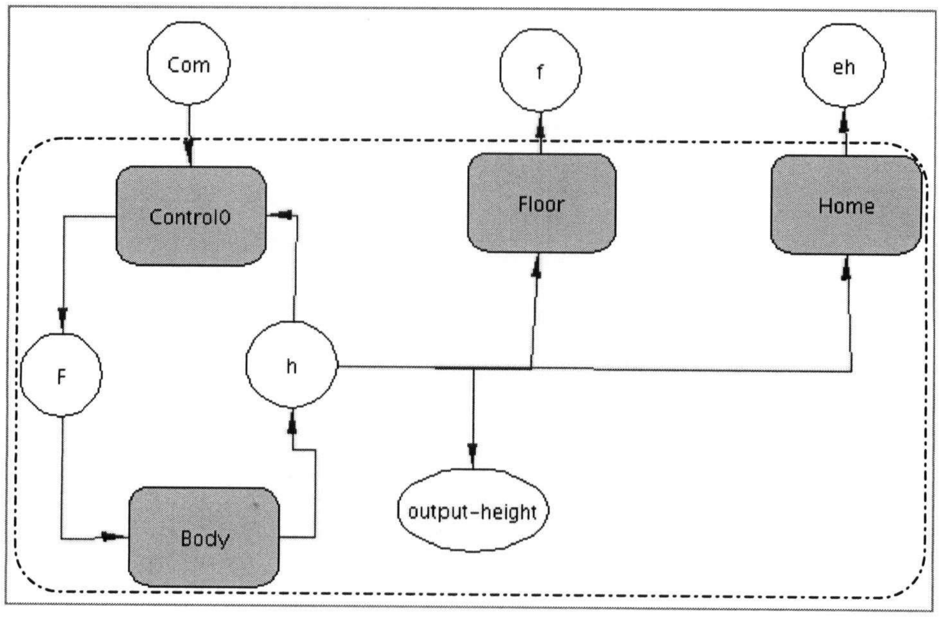


Figure B-11. ELEVATOR Module

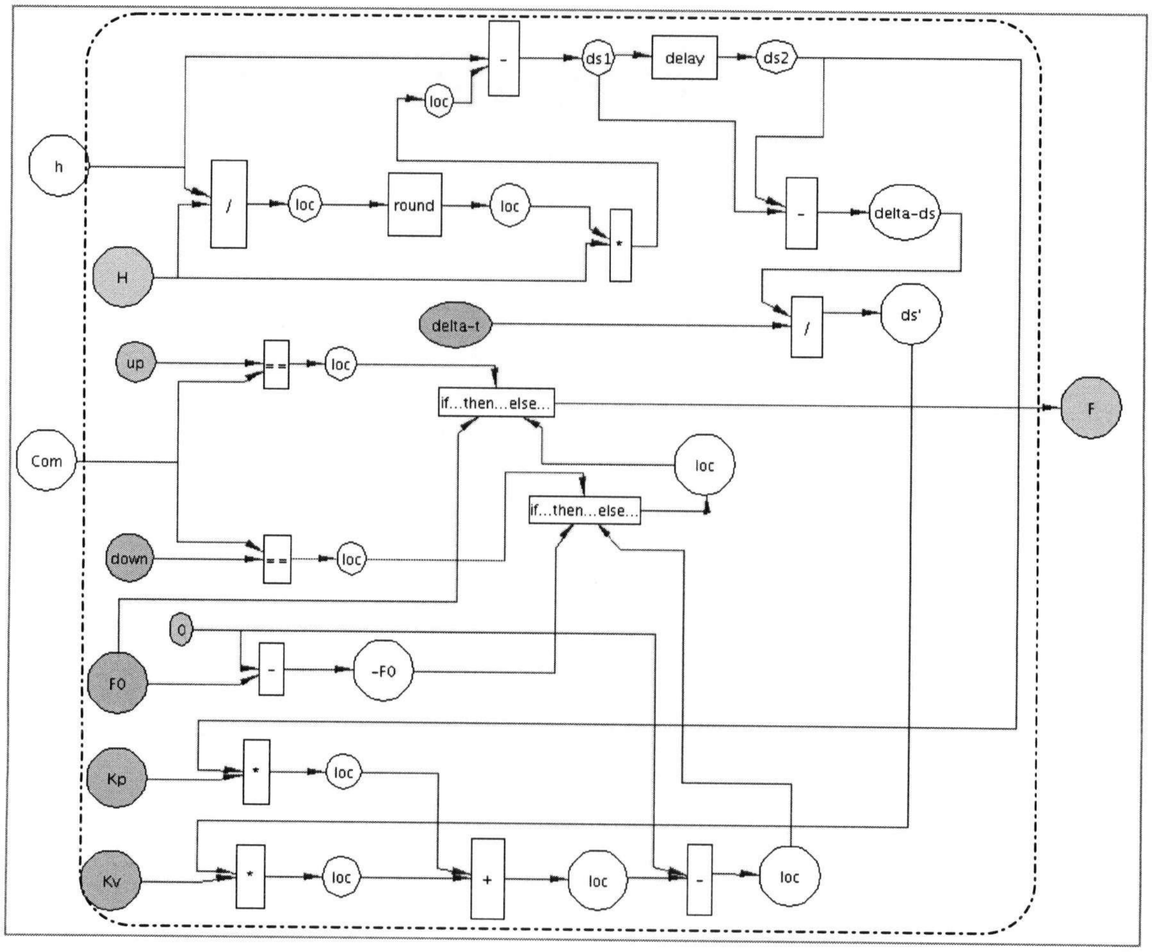


Figure B-12. CONTROL0 Module

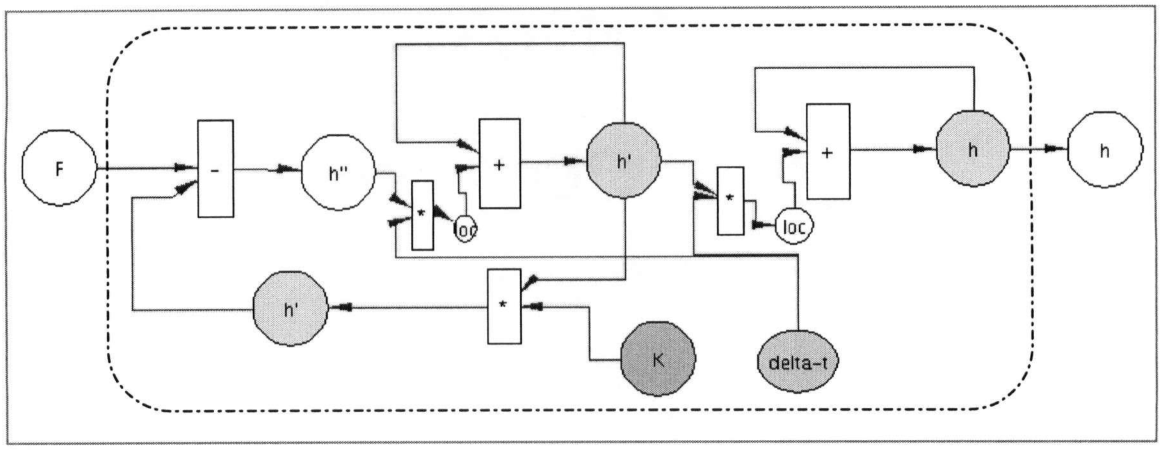


Figure B-13. BODY Module

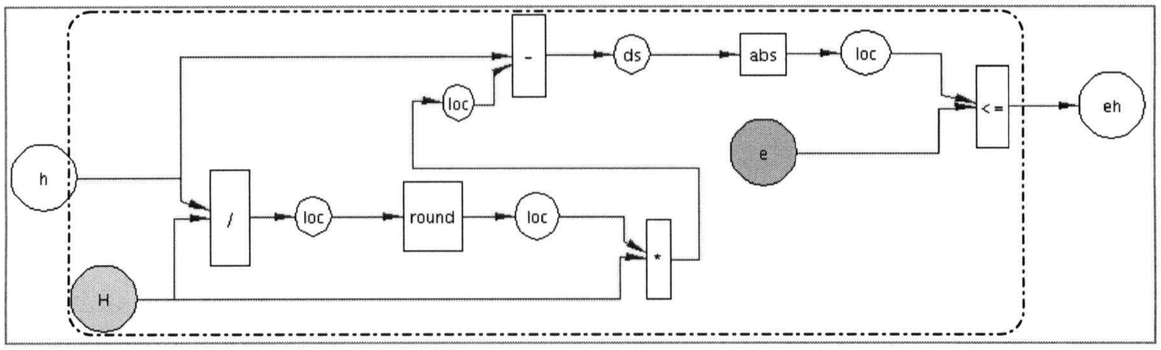


Figure B-14. HOME Module

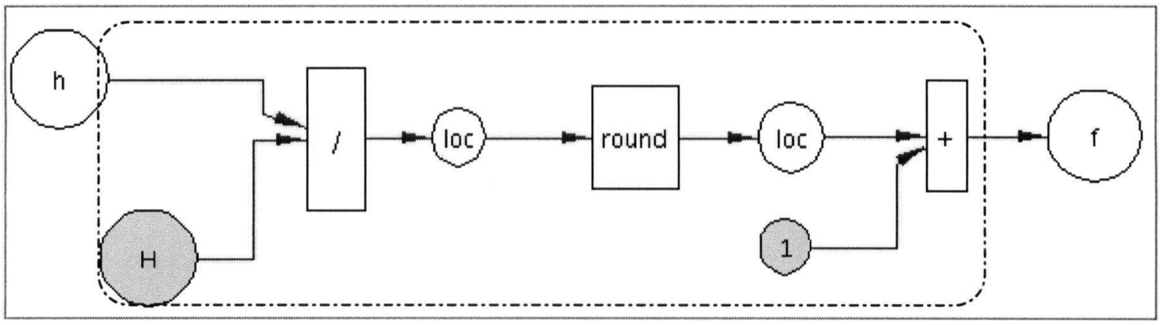


Figure B-15. FLOOR Module