An In-Vehicle Embedded System for Intelligent Transport System : Modeling and Simulation

By

Ravisaheb Rushikumar R.

(05MCE013)

Department of Computer Science & Engineering

Institute of Technology

Nirma University of Science & Technology

Ahmedabad 382481

May 2007

An In-Vehicle Embedded System for Intelligent Transport System : Modeling and Simulation

By

Ravisaheb Rushikumar R.
(05MCE013)

Guide

Dr. (Prof.) S. N. Pradhan

A Dissertation
Submitted to

Nirma University of Science and Technology
in partial fulfillment of the requirement
for the degree of

Master of Technology

Department of Computer Science & Engineering
Institute of Technology
Nirma University of Science & Technology
Ahmedabad 382481
May 2007

This is to certify that Dissertation entitled

An In-Vehicle Embedded System For Intelligent Transport System : Modeling and Simulation

Submitted by

Ravisaheb Rushikumar R.

has been accepted toward fulfillment of the requirement
for the degree of
Master of Technology in Computer Science & Engineering

Prof. S. N. Pradhan	Prof. D. J. Patel
Professor In Charge	Head of The Department

Prof. A. B. Patel
Director, Institute of Technology

CERTIFICATE

This is to certify that the Major Project entitled “An In-Vehicle Embedded System For Intelligent Transport System : Modeling and Simulation” submitted by Mr. Rushikumar R. Ravisaheb (05MCE013), towards the partial fulfillment of the requirements for the degree of Master of Technology in Compute Science & Engineering of Nirma University of Science and Technology, Ahmedabad is the record of work carried out by him under my supervision and guidance. In my opinion, the submitted work has reached a level required for being accepted for examination. The results embodied in this major project, to the best of my knowledge, haven’t been submitted to any other university or institution for award of any Master degree.

Project Guide

Dr. (Prof.) S. N. Pradhan
P. G. Coordinator,
Department of Computer Engineering,
Institute of Technology,
Nirma University,
Ahmedabad

Date :-

ACKNOWLEDGEMENT

The success that I have got in the accomplishment of the project work is not only due to my efforts. In fact, I stand on the shoulders of many people. So, with great pleasure I take this opportunity to express my gratitude towards all the individuals who have helped and inspired me in my project work. It was a great experience working on this project, about a totally new concept. The project taught me many things and added knowledge to my memory bank.

First of all, I would like to express my earnest gratitude to my internal project guide **Dr. S. N. Pradhan,** M.Tech In-Charge, Department of Computer Science and Engineering, Nirma University for their constant guidance, encouragement and moral support which helped me to accomplish the project.

I would like to thank our Head of the Department **Prof. D. J. Patel** for helping me in all possible ways.

I am heartily thankful to **Prof. Jaladhi Joshi**, for giving new direction to my system's model and helping in getting to know the system.

I extend my sincere gratitude to my **parents** and **all family** members for their moral support. I would also like to be grateful to my **fiancée** for helping me in all possible ways whenever I looked for her help and always being on my side.

Finally, I would like to thank all my friends and classmates for their coordination and constant help, without which project wouldn't be done.

Last but not the least, I am thankful to **God** for giving me the light and strength to work and making this project a success.

Ravisaheb Rushikumar R.
(05MCE013)

ABSTRACT

Computers & communications are becoming integral part of intelligent transport systems. Such system aims to provide better amenity for passengers, vehicle condition monitoring, driver performance and vehicle fleet management. The thesis discusses the development of such a system using modern embedded system design approach.

Embedded system described here automates all functionalities including the timing information, ticketing process; station information, passenger information etc., which are packaged with minimal manual entry required by bus-driver. Development cycle of an embedded system includes modeling as the foremost step. Modeling is the process which reduces the development time and also helps in testing the system before manufacturing. That way modeling reduces market window of an embedded system by great amount. For modeling Ptolemy 6.0.2, which is open source tool from Berkeley University is used.

All the functions are understood and defined with separate modules, and models have been developed that simulates the functionality. The Ptolemy tool refers each individual entity as an actor that operates on tokens. During project some actors in original tool have been modified and found more efficient later. Also some actors' are developed, patched and then used for this project. Simulation needed cross compilation of the generated code for specific target system and then checking its performance.

Different modules have been implemented to develop entire design. System statecharts, model design and working of modules are included. After implementation of individual units, they have been integrated. Core Java and Cygwin were used for adding actors and code generation.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS AND ABBREVIATIONS

API	—	Application Programming Interface
BTS	—	Bus Transport System
CHESS	—	the Center for Hybrid and Embedded Software Systems
CT	—	Continuous Time
DDE	—	Distributed Discrete Event
DE	—	Discrete Event
DLL	—	Dynamic Link Library
DT	—	Discrete Time
ES	—	Embedded System
FSM	—	Finite State Machine
GCC	—	GNU Compiler Collection
GUI	—	Graphical User Interface
HDF	—	Heterogeneous DataFlow
HDL	—	Hardware Description Language
ITS	—	Intelligent Transportation System
JDK	—	Java Development Toolkit
JVM	—	Java Virtual Machine
LCD	—	Liquid Crystal Display
MoC	—	Model of Computation
SDF	—	Synchronous DataFlow
SDL	—	Simple DirectMedia Layer
UML	—	Unified Modeling Language

1. INTRODUCTION

This chapter includes overview of the project. The most important thing about project is “why the project is required?” To answer of this question, a separate section named “Need of Project” is included. This chapter also includes aim of the project and scope of the project. “Scope of Project” explains which kind of domains can use this project.

1.1 PROJECT OVERVIEW

Modeling and simulation are very key phases of embedded system manufacturing process. Modeling is necessary as it reduces market window of an ES by great amount. Also it is always better to verify the design offline, i.e. without manufacturing it and then testing the same. For these and also many other purpose modeling is very much required and helpful too. Simulation is just other side of a coin in which one side is modeling. They are indifferent from each other. Simulation tests the model on particular target processor. Here the project is carried out to learn modeling of an Embedded System.

For the same need concept of an ES is taken, BTS is the system used to model. An instant question that arises is like what is BTS then? BTS stands for Bus Transport System. It is a kind of ITS – Intelligent Transport System. BTS is emerging concept in metro cities. The concept of the complete system has set of functions, assumptions, pros and cons. The complete system is made for better amenity of passengers and for easier and efficient administration. Details of the system are very much required to understand prior knowing modeling and other details. ITS have been developed and deployed at many places. They are a good example of such transportation systems. There is extreme need of such systems in many countries. The needs arise because of many reasons. Few of the reasons for need for such systems are,

1. Traffic congestion due to population.
2. Low awareness of traffic rules and regulations.
3. Low level of Public Transport Services.

There are transport systems found in city and also for interstate traveling. These transport system has every single function done manually. That is bus conductor notifies passengers about the stations, routes and etc. Conductor also issues tickets to the passengers and collects charges for that. Concept of BTS in brief is to do bus conductor’s operations automatically. In addition to these many other functions can be achieved with BTS, functions which helps not only passengers but also the administration to improve the system. Functions which this system can provide are as below.

Functions for passengers:

1. Alert passengers about next coming stations on route.
2. Alert passengers about current time and time when next station comes.
3. Issue tickets to passengers according to destination selected.
4. Fare collection from the passenger for the ticket.
5. Inform users waiting outside bus, enroute forth coming station about the bus timing and route.

Function for Administrators:

1. Note practical value of time taken, for complete journey and for each stations enroute.
2. Check upon driver performance.
3. Check traffic situations and update journey according to that.
4. Communicate with other buses to send data for traffic situations and etc. details.
5. Communicate with payment system (e.g. Card Company).

The list may go on. These are some of the functionalities BTS can provide to the system users. Only some functions are modeled for project and that details are covered in next chapters.

ITS or BTS may still have not been found if there are no advanced technologies available. Such systems are found and are used with support from communication technologies, computers and etc. There are several factors that promote deployment of these systems, use of credit card for payment, increased use of mobile phones and etc.

To achieve some of the functionality stated above, many approaches have been used so far at European and Asian countries. Approaches that were used so far lacks some or the other functionality from the list above. Approaches used were using,

1. Informing users via SMS.
2. Measuring a system for performance periodically.
3. Electronic Toll collection for tickets.
4. Vehicle tracking using GPS.
5. Public transport management systems.

In this project, aim is to prepare a working model of BTS. Then simulate it for a target processor family. For the same purpose of modeling Ptolemy II is used, which is a GUI based modeling tool from Berkeley University, California.

1.2 NEED OF PROJECT

Need of an embedded system in today's world is still in demand. And so is the need of modeling the system. Modeling improves system's market window, development time and also testing of the system. Modeling is key phase of any manufacturing process and its crucial one also. So it is very much necessary to model and test a system offline with modeling tools and methods available. The reason behind it is the speed of the development cycle of ES. There are lots of options for one to choose the software to perform application like modeling of an ES. GUI based modeling is better compared to other ways of modeling. As in GUI based model every single detail of model is graphical and can be easily understood by even naïve users also. Here Ptolemy tool which is GUI based tool to model system is used which serves good example of model based approach of modeling an ES.

As stated in above section, the system modeled is BTS. Such systems are developed and deployed. Yet there is scope of adding more features to make it more useful. To provide these functions a new system is required which may serves users in best ways. So there is desperate need to come out with a new embedded system that has additional functionality. For this purpose here an ES is proposed for BTS. The proposed ES is modeled and verified using modeling tools known as Ptolemy II.

1.3 AIM OF PROJECT

Ultimate aim of this project is to model the BTS (Bus Transport System) with Ptolemy II. System here has a limited set of functionality. It is to be modeled to provide following functional specification,

1. Alert passengers about next coming stations enroute.
2. Alert passengers about current time and time when next station comes.
3. Charge fares from passengers according to destination selected.
4. Collect charge from the passenger for the ticket.
5. Record time taken, for complete journey and also for each station along the route.

All these functionality are required to be modeled and tested for the desired outcomes. Model should generate desired outcomes. Once they are tested for performance, the system needs to be integrated. After complete system is available, it can be simulated on target platform.

1.4 SCOPE OF PROJECT

Modeling is basic requirement of any embedded system. Here a complete different approach of modeling is used. This way of GUI based modeling can be used in many places and in many ways. This helps users and developers to understand the model more intuitively. And development can be much faster.

Model of BTS if fits some target processor family, then design can also be implemented on hardware level. Also hardware software co-simulation can be done once a model is giving desired outcomes. In all case this will surely help other developers in modeling an ES with Ptolemy II.

2. MODELING AND SIMULATION

Here the needs of modeling are explained in brief. All embedded system needs to be modeled and then simulated to verify the functional behavior. Once everything goes well, the system can be manufactured. So simulation is also focused in this section. Also the complete cycle of modeling and simulation used in here is explained.

2.1 INTRODUCTION

Embedded systems interact with the physical world through sensors and actuators. These days, most include both hardware and software designs that are specialized to the application. Conceptually, the distinction between hardware and software, from the perspective of computation, has only to do with the degree of concurrency and the role of time. An application with a large amount of concurrency and a heavy temporal content might as well be thought of as using hardware abstractions, regardless of how it is implemented. An application that is sequential and has no temporal behavior might as well be thought of as using software abstractions, regardless of how it is implemented. The key problem becomes one of identifying the appropriate abstractions for representing the design [5].

Unfortunately, for embedded systems, single unified approaches to building such abstractions have not, as yet, proven effective. HDLs with discrete-event semantics are not well-suited to describing software. On the other hand, imperative languages with sequential semantics are not well-suited to describing hardware. Neither is particularly good at expressing the concurrency and timing in embedded software. Another approach is to increase the expressiveness of the languages in use. VHDL, for example, combines discrete-event semantics with a reasonably expressive imperative subset, allowing designers to mix hardware abstractions and software abstractions in the same designs. To attempt to unify these design styles, the VLSI design community has made heroic efforts to translate imperative VHDL into hardware with only limited success. A significantly different direction has been to develop domain-specific languages and synthesis tools for those languages [5].

Primarily, actor-oriented design allows designers to consider the interaction between components distinctly from the specification of component behavior. Model-based design is simply the observation that if one uses a modeling language to state all the important properties of a design, then that model can and should be refined into an implementation. To accomplish model-based design, therefore, one needs a design framework [5].

2.2 NEED OF MODELING

A model is a pattern, plan, representation, or description designed to show the structure or workings of an object, system, or concept. The model is built by considering all aspects of the systems. According to the need of the system, every single detail is considered and the model is generated for desired outcome.

These day modeling is much of a necessity. Modeling reduces market window by great amount. Also it reduces as well as improves testing of the system before getting manufactured. If and only if the desired outcome is achieved then system can be manufactured so the time and cost is also reduced. Modeling is done with special care so as to represent the complete system identically. Modeling advantages the manufacturer at time of modification also, as the upgrade can be tested before getting patched to the design.

As models are good tools for humans in understanding and creating complex structures, they also have definite advantages if the system itself is expected to be reflective, i.e. to be able to supervise its own operation. Eventually this facilitates the creation of self-adaptive computing architectures, where automatic adaptation itself is based and carried out on the well-understood models of the embedded system. A significant part of embedded systems are also required to be fault-tolerant, manageable, or even externally serviceable, both in the hardware and the software sense. The model-based approach is definitely helpful in providing these features, since the decomposition boundaries usually also identify standardized access points for those operations.

2.3 MODELING CYCLE

The complete modeling process followed here is shown in fig 2.1. The complete cycle consists of phases starting with requirement study till the simulation of the system to get desired outcome. Once desired outcome is achieved then the system can be manufactured. Based on available options with tools used, the model will behave. Sometimes if some actors are not available then in that case it has to be developed. Here the modeling cycle shows how to generate and use actors with the available ones.

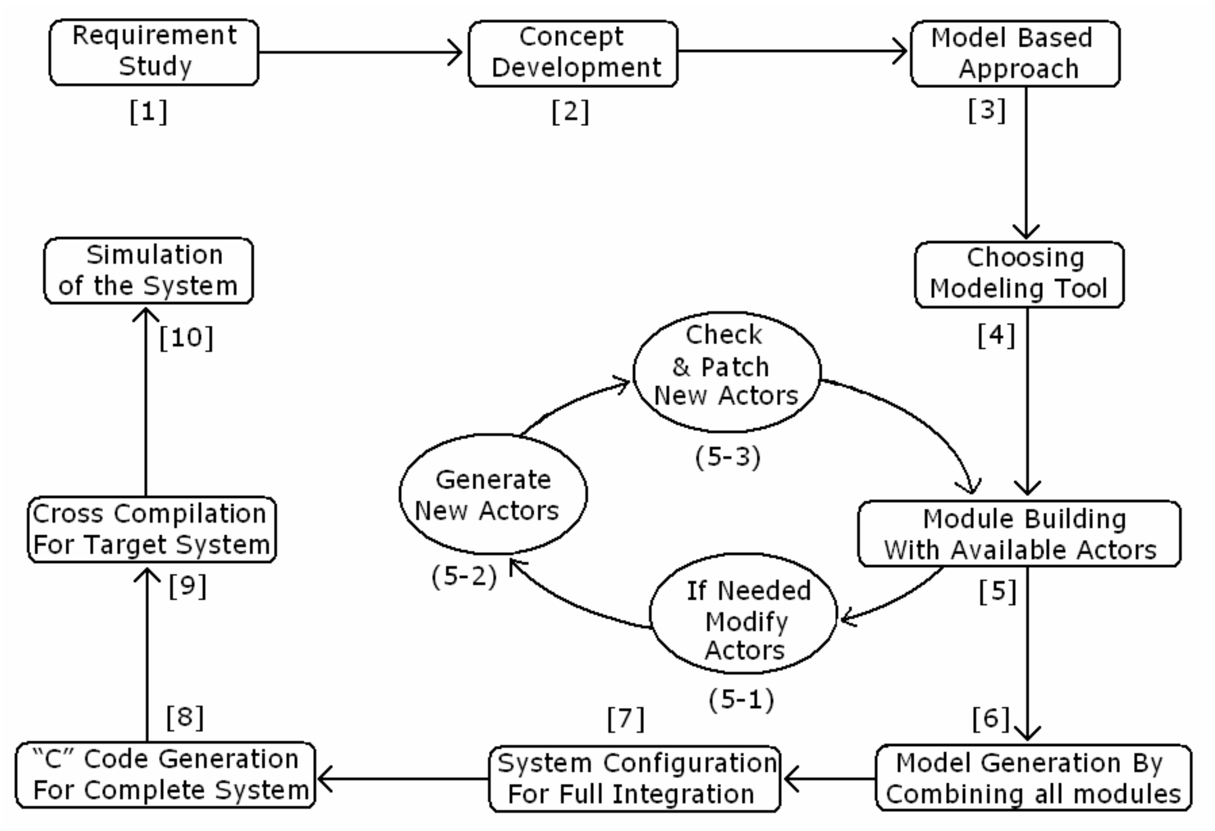


Fig 2.1 Modeling Cycle

Starting with requirements the need and scope is firstly decided and studied. Then the complete concept of the system development is understood and planed. As per the approach to be used the system has to be devised and designed. The approach can be model based or any other that includes model and simulation linked together. Once the approach is decided and outline of the design is generated then tool has to be chosen for modeling. Here Ptolemy [7] tool from Berkeley University is used.

Once the tool is chosen then model building has to be started. Here in Ptolemy the models are built in terms of actors. The complete system is divided in several modules, and so design is done module wise. The modules are built with available actors. If the actors are not available then for that actors have to be generated and then used. The actors that are generated have to be checked and patched to the current environment and then it can be used. Once all the modules are ready then they can be combined together to generate model of the system. Once the desired model is readily available then it has to be simulated. Prior to simulating the system, it has to be configured for the target system. According to the target environment the model has to be configured, the configuration includes processor family details and other hardware information. Once the system is configured then "C" language code is generated for the system. The "C" code is generated as it is compatible for all processor families. Finally the "C" code is cross compiled for target system and then it can be simulated. The cross compilation also requires configuration based on the target platform.

2.4 ACTOR ORIENTED MODELING

Here actor oriented models [1] are used for modeling purpose. There is other several modeling methodology available, the choice depends upon the need and existing situations. In actor-oriented design, components called actors execute and communicate with other actors in a model. Actors have a well-defined component interface. This interface abstracts the internal state and behavior of an actor, and restricts how an actor interacts with its environment. The interface includes ports that represent points of communication for an actor, and parameters that are used to configure the operation of an actor. Often, parameter values are part of the *a* priori configuration of an actor and do not change when a model is executed. The configuration of a model also contains explicit communication channels that pass data from one port to another. The use of channels to mediate communication implies that actors interact only with the channels that they are connected to and not directly with other actors.

Like actors, which have a well-defined external interface, models which are compositions of interconnected actors may also define an external interface.

External interfaces allow for hierarchical abstraction. This interface consists of external ports and external parameters, which are distinct from the ports and parameters of the individual actors in the model. The external ports of a model can be connected by channels to other external ports of the model or to the ports of actors that comprise the model. Taken together, the concepts of models, actors, ports, parameters and channels describe the abstract syntax of actor-oriented design [2].

In actor-oriented design, actors are the primary units of functionality. Actors have a well defined interface, which abstracts internal state and execution of an actor and restricts how an actor interacts with its environment. Externally, this interface includes ports that represent points of communication for an actor and parameters which are used to configure the behavior of an actor. Actors are composed with other actors to form composite actors or models. Connections between actor ports represent communication channels that pass data tokens from one port to another. The semantics of composition, including the communication style, is determined by a model of computation. When necessary, the model of computation will be shown explicitly as an independent director object in model. Models often export an external actor interface, enabling them to be further composed with other models.

A central concept in actor-oriented design is that internal behavior and state of an actor are hidden behind the actor interface and not visible externally. This property of strong encapsulation separates the behavior of a component from the interaction of that component with other components. System architects can design at a high level of abstraction and consider the behavioral properties of different models of computation independently from the behavioral properties of components. Furthermore, different models of computation can be used at different levels of hierarchy, enabling hierarchically heterogeneous design. By emphasizing strong encapsulation, actor-oriented design addresses the separation of concerns between component behavior and component interaction. In addition to supporting hierarchically heterogeneous models, strong encapsulation allows primitive or atomic actors to be specified in a variety of ways. For instance, actors are often specified by drawing finite-state machines where each transition corresponds to a particular sequence of operations.

Another technique is to use a special purpose textual language that specifies what tokens to consume and what operations to compute on that data. However, one of the most flexible ways to specify actor behavior is to embed the specification within a traditional programming language, such as Java or C, and use special purpose programming interfaces for specifying ports and sending and receiving data. This technique has been widely used in actor-oriented systems since it allows for existing code to be integrated into an actor-oriented design tool and for programmers to quickly start using actor-oriented methodologies.

2.5 SIMULATION

A simulation is an imitation of some real thing, state of affairs, or process. The act of simulating something generally entails representing certain key characteristics or behaviors of a selected physical or abstract system. Simulation is used in many contexts, including the modeling of natural systems or human systems in order to gain insight into their functioning. Other contexts include simulation of technology for performance optimization, safety engineering, testing, training and education. Simulation can be used to show the eventual real effects of alternative conditions and courses of action. Key issues in simulation include acquisition of valid source information about the referent, selection of key characteristics and behaviors, the use of simplifying approximations and assumptions within the simulation, and fidelity and validity of the simulation outcomes.

A computer simulation is an attempt to model a real-life situation on a computer so that it can be studied to see how the system works. By changing variables, predictions may be made about the behavior of the system. Here once the system is modeled it is simulated for the outcomes. The simulations are done under all the certain conditions under which the system will be working. The results are noted down and then it can be verified. If all the things are desirable then system can be manufactured. In here the simulation is yet a task to be carried out. There are certain conditions that have to be meet prior simulating the system; those conditions are defined for the design. Hardware dependencies has to be resolved if any, before simulation can take place.

3. TOOLS & TOOLKIT

This chapter explains details about the Ptolemy tool used throughout for the development. Out of many available details few important ones are referred in brief. Also how modeling and simulation can be carried out with the tool is explained here. Also it covers Cygwin toolkit's details which are used in system development process as a backend to Ptolemy for developing actors and modules.

3.1 PTOLEMY TOOL

Ptolemy Project is an informal group of researchers that is part of CHESS at U.C. Berkeley. This project conducts foundational and applied research in software based design techniques for embedded systems. Ptolemy II is the current software infrastructure of the Ptolemy Project. Ptolemy II is the third generation of design software to emerge from this group, with each generation bringing a new set of problems being addressed, new emphasis, and a new group of contributors.

The Ptolemy project studies heterogeneous modeling, simulation, and design of concurrent systems. The focus is on embedded systems, particularly those that mix technologies.

3.1.1 Modeling

Ptolemy project helps in modeling embedded system with many available options with its Vergil tool [10]. There are many actors available with the tool that can be used, and other actors have to be generated. As referred in section 2.2 ptolemy project uses actor oriented modeling.

Modeling is the act of representing a system or subsystem formally. A model might be mathematical, in which case it can be viewed as a set of assertions about properties of the system such as its functionality or physical dimensions. A model can also be constructive, in which case it defines a computational procedure that mimics a set of properties of the system. Constructive models are often used to describe behavior of a system in response to stimulus from outside the system. Constructive models are also called executable models. Design is the act of defining a system or subsystem. Usually this involves defining one or more models of the system and refining the models until the desired functionality is obtained within a set of constraints. Design and modeling are obviously closely coupled. In some circumstances, models may be immutable, in the sense that they describe subsystems, constraints, or behaviors that are externally imposed on a design. For instance, they may describe a mechanical system that is not under design, but must be controlled by an electronic system that is under design. Executable models are sometimes called simulations, an appropriate term when the executable model is clearly distinct from the system it models. However, in many electronic systems, a model that starts as a simulation mutates into a software implementation of the system. The distinction between the model and the system itself becomes blurred in this case. This is particularly true for embedded softwares [1].

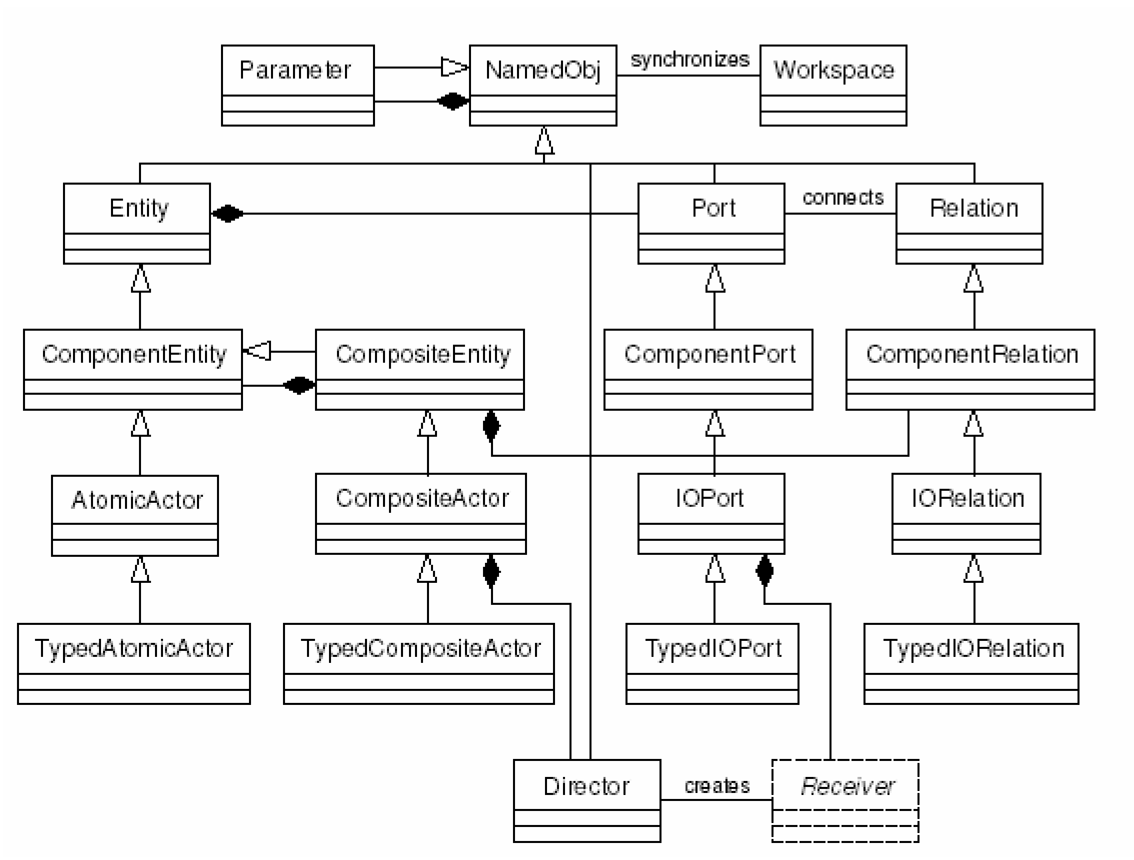


Fig 3.1 Ptolemy's Ptolemy.actor package with supporting classes

As shown in figure 2.1, the complete modeling cycle can be implemented here. Modeling starts from step 4, i.e. after choosing Ptolemy' vergil as the tool to model the system. Then all the modules in which system is divided can be developed using available actors. An actor is any entity in the system which consumes and produces token at each step, and does desired functions. So in here, for actor oriented modeling every single function has particular actor for that. There are many inbuilt actors available under vergil. Rest actors can be generated with help of Cygwin.

For the need of generating system's "C" code, there is code generation environment available under vergil. This environment limits number of actors available. In here very few of the actors are available which are there in normal vergil environment. For each actor available, there is C code available. So that finally while integrating full system, the C code will combine and then generate full system's code. That code can be cross compiled, simulated and then tested for the target system. Steps 4 to 6 of the modeling cycle of fig. 2.1 are executed for modeling of the system.

3.1.2 Vergil and its Elements

There are many ways to use Ptolemy II. It can be used as a framework for assembling software components, as a modeling and simulation tool, as a block-diagram editor, as a system-level rapid prototyping application, as a toolkit supporting research in component-based design, or as a toolkit for building Java applications [1]. Vergil is GUI to construct models graphically. Vergil comes with many editors for constructing models. Editors like, FSM editors, graph editor, icon editor and etc. are used for model building. There are many things to be considered for vergil, like actors, directors, tokens, expressions and etc.

Actors are very basic components of Vergil. They have well defined functionalities to achieve, they consume or/and produce token at every step of execution. There are many actors for functions like, audio, source, sinks, signal processing, mathematics, logical functions and many more. Actors can communicate with other actors of the model.

Directors are the elements which handles execution of the model. It has configuration for timing of module, steps, period of execution, details of hierarchy and etc. that will control execution of models. In all there are many directors or model of computation like, SDF (Synchronous Dataflow), CT (Continuous Time), HDF (Heterogeneous Dataflow), FSM (Finite State Machine), DT (Discrete Time), DE (Discrete Events), DDE (Distributed DE) and etc. [1]. Whereas only some of the directors are available under code generation environment like SDF, HDF and FSM.

The input and output values of the actors are encapsulated as tokens. Those tokens are consumed and produced by all or some of the actors. There are certain limitations on the token generated as per the data types of the actors. To have compatibility among the tokens either data types or the coding has to be changed.

There are other options like entities of the system, relations of the actors and models and hierarchy. These all details have to be considered at the time of modeling the system.

3.1.3 Simulation

To simulate the code under vergil, the tool will generate the code of the complete system once it is modeled. Generated models can be executed and verified for the desired result. Once model is ready then C code can be generated, according to the steps shown in fig 2.1. For simulation the C code is cross compiled for target system and then it can be mounted on the target system. Then it can be checked for the performance.

3.1.4 Code Generation Environment

The need for code generation is to have the model run under any processor family according to need. It is well known that very few processors are compatible for Java. As here the Ptolemy environment is built under java and it generates the java code so it becomes a necessity to have the code generated for the Model. Normally every processor family have cross compiler for 'C' language and so here the code generation is done in C.

The Ptolemy has code generation facility available in it. This facility is still under development phase, so very few actors and directors are supported here. The environment is to have Static Code Generator only with no timing details entertained.

This is a highly preliminary code generator facility, with many limitations. It is best viewed as a concept demonstration [4].

1. Only SDF, FSM and HDF domains are supported
2. Only IntToken, DoubleToken, StringToken and ArrayToken are supported. Other tokens are not supported at this time.
3. A limited number of actors have supporting helper code.

For code generation, the Cygwin is a must thing to have. The code generation is done with make and gcc facility with Cygwin. Also the JVM and JDK is needed as per the models to be built. The actors here have C files already pre-generated so at time of code generation using the files and design of the model code is generated.

Very few actors are supported with codegen environment, so there is need to change the complete modules designs. According to the environment and need for the ES, each module is changed. Currently due to limitations of codegen environment the design of modules is restricted.

3.2 CYGWIN

Cygwin is a Linux-like environment for Windows. It consists of two parts, a DLL which acts as a Linux API emulation layer providing substantial Linux API functionality and a collection of tools which provide Linux look and feel [8]. Cygwin is a collection of free software tools originally developed by Cygnus Solutions to allow various versions of Microsoft Windows to act similar to a UNIX system. It aims mainly at porting software that runs on POSIX to run on Windows with little more than a recompilation. While Cygwin provides header files and libraries that make it easier to recompile or port UNIX applications for use on Windows, it does not directly make UNIX binaries compatible with Windows [8].

Cygwin has many or all software tools used for source coding in C in GCC and for kernel compilation and editing of system files. It comes with all functionalities packaged to use with windows. The version of GCC that comes with Cygwin has various extensions for creating Windows DLLs, specifying whether a program is a windowing or console mode program, adding resources, etc.

Cygwin is used here for actor generation and then adding that to Ptolemy environment. The details of actor adding and source recompilation and etc. are covered in appendix B. Other than Cygwin for actor adding under Ptolemy eclipse is also used, but here Cygwin is used. Appendix B covers details on Cygwin installation and use particularly for Ptolemy.

4. BUS TRANSPORT SYSTEM

This chapter discusses the details about main system modeled and simulated here. Main goal of thesis is to focus modeling and simulation process and the system described here is like test case. It also covers all the details regarding phases, modules of the system, hardware that can be designed and etc. All desired functionalities with assumptions are described and then functions achieved and their limitations are listed.

4.1 INTRODUCTION

The system as name says is made for easy state level or city level transport. Bus transports with the functionalities listed below may result in better amenity for users. The system will facilitate passengers in better way. Also it will make controlling easier on the driver's part. Such systems are already developed and are in use. Here the system improves earlier system by taking into account the history of each journey. Such details will help in monitoring the performance and later improving the system in all possible ways. Complete system will be mounted on the Bus only. Driver will be able to do the controlling of the system with the device present; also that device will help in generating automatic system alerts for the passengers. The device for passengers is mainly for the ticketing purpose.

4.2 FUNCTIONALITIES

Below listed are the functions that the system is designed to perform. All the listed functions have been developed with Ptolemy. Various functions need support by one or more states. These functions are for better amenity of passengers and easy maintenance and control by the driver.

Route Information

1. Alert about all station on route.
2. Alerting passengers for coming stations.
3. Alerts about stoppages.

This will alert the users about the route. The information generated will be in audio and visual both form. In here for every station there will be fixed display messages that will be continuously guide passengers. These messages will be shown on LCD screen display. Other form of information is through audio alerts. Here also a list of messages will be announced periodically for user’s information. Both of the informations are controlled by the driver. Here the driver will choose the current station and according to that the messages will be loaded to the system’s memory.

Timing Detail

1. Timing alerts for arrival and departure.
2. Timing for each station on route.

As the name and functions says, this is particularly for timing information. Again this is also visual form of alert. This particular function also helps system administrator to update the system if needed. This will record the timing for each station and so that this time can be verified with the ideal timing. Thus recorded time can also be helpful to know driver performance, traffic situation, delay due to natural accidents etc. So in all this functionality will help administrator more in comparison to the passengers.

Fare Charging

1. Based on Source and Destination.
2. Based on the distance to be traveled.
3. Using prepaid card of fixed balance, easily available at bus depots.

The fare charging function can be subdivided into two separate functions, which it serves - Ticketing and Payments process.

Tickets are generated for the journey selected by passenger. From the device as in fig 4.8, the passenger may select the destination station. Source station will be decided by the current location of the bus on the journey. The fare will be computed by the system using these informations. The fare will be deducted from the prepaid card held by user. Fare calculation and charging both are interlinked processes. Once payment is successful then only ticket will be issued to the passenger for the journey selected.

In all transport systems payment is more of interest for administrator. The system uses concept of prepaid card. This card is assumed to be easily available at all bus depots. And these cards are assumed to have fix balance on it and it will carry its unique number which the system knows in advance. Using the card purchased from deports, journey can be done and ticketing and payment can also be carried out.

Passenger Information

1. Total capacity of the Bus.
2. Total vacant seats.

This information is generated for the system only. This is advance information for systems performance. This particular operation can be done in future. By use of sensor attached to every seat, or by some sensing mechanism the seats can be counted, i.e. which are occupied and which are vacant. This information can be communicated to central server then central server would display the details at the bus stops.

4.3 ASSUMPTIONS

While designing any embedded system, there are certain assumptions to be made about the environment under which this system will operate. Without assumptions, in all possible cases system has to be operable but it is not the case here. The assumptions are made for flexibility of the need of the system being modeled. Some of the assumptions are listed below.

1. There are always reliable passengers.
2. Prepaid cards are easily accessible at bus depots.
3. No failure in traveling is considered for accidents or traffic.
4. Every single detail that the system is able to represent is based on the data entered earlier.

5. There is no communication concept used for the payment, the device or the card is assumed to have details.
6. All operations are modeled for ideal conditions.

4.4 STATECHART REPRESENTATION

Statecharts is a visual design methodology and notation, for managing the complexities of designing simulators, such as medical, aviation, and consumer electronics devices. It is an extension of deterministic finite state automata or state machines from the field of computer science. More recently, statecharts have been adopted into the Unified Modeling Language (UML) [3]. A statechart is a complete graphical characterization of a system's potential behavior to the level of detail required for the simulation. It consists of discrete "states" and "transitions." Each state represents a distinct context for behaviors of the device, such as an ON state and an OFF state.

The model can be represented graphically with use of the statecharts of the system. The state chart is an easy way to represent the functionalities and state of the system. It uses concept of hierarchy, clustering etc. which enables model to be compact and understandable. All systems can be defined and explained in better with use of statecharts. These diagrams covers details of each phase of system, outcomes, initial states, preconditions and etc. [3]. Given below are the statecharts of the BTS.

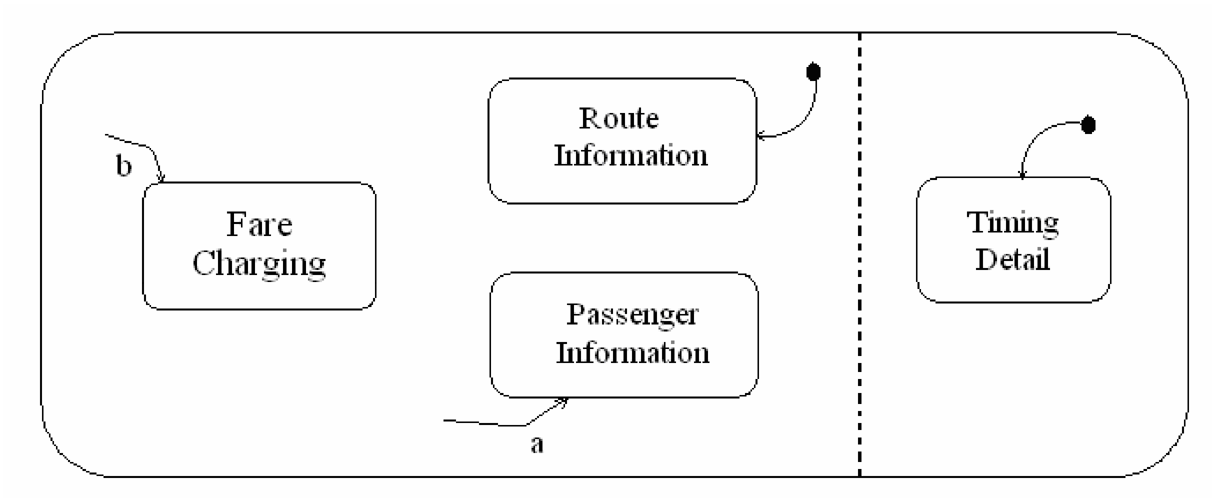


Fig 4.1 Bus Transport System

The fig. 4.1 shows the main system. Here these are the functionalities that BTS will provide. The timing detail can be provided in parallel with any of the other four. Based on the button pressed the other functions can be activated, and executed. Below are the individual state charts of each of the functional block of the system. The representation shows that the default state of the system will be route information and timing details. So that throughout the system's execution these states will be executing all the time. Ticketing and fare charging are linked to each other so; they can not be separated even for the execution purpose.

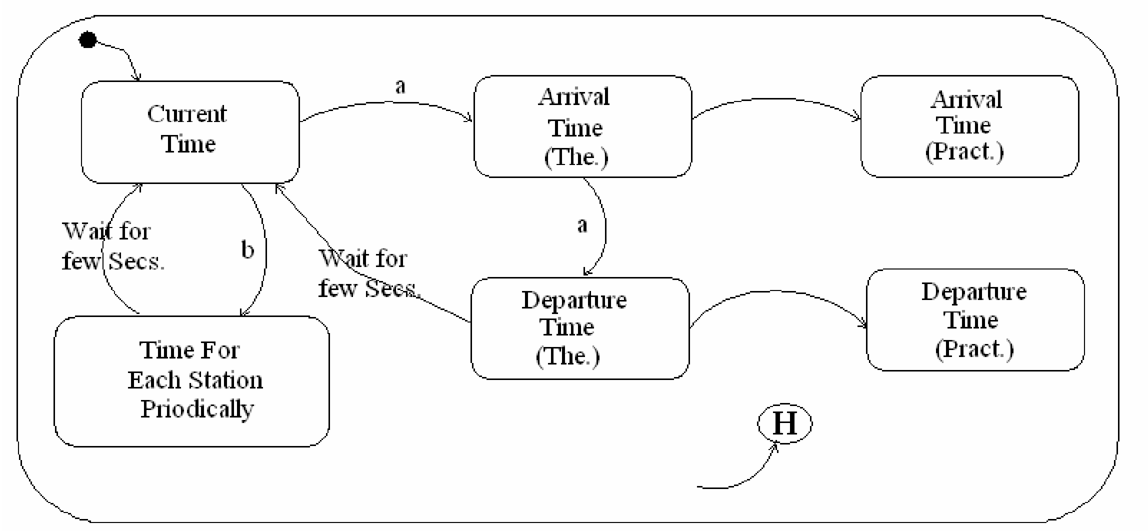


Fig 4.2 Timing Details

Above shown fig. 4.2 explains timing detail function of BTS. There is always concept of history present when we are concerned with time as the time needs to be updated and registered. Here the theoretical and practical time for both arrival and departure is used. Practical times will not be updated in current execution, but this detail can be verified with theoretical values and if on average it is found that the value has to be changed then, theoretical value can be updated. This detail can be very useful in system modification later on. When ever the system refers to this block, it will show current time on default which is highlighted with the black dot. The waiting for few seconds can be configured based on the requirement.

Fig. 4.3 below shows passenger information operation of the system. It will inform about the capacity occupied on the bus. The count of passenger needs to be monitored so history is needed. When some passengers get off the bus or board the bus then at that time the count of passenger is required to be updated. This information as stated above is for administration use.

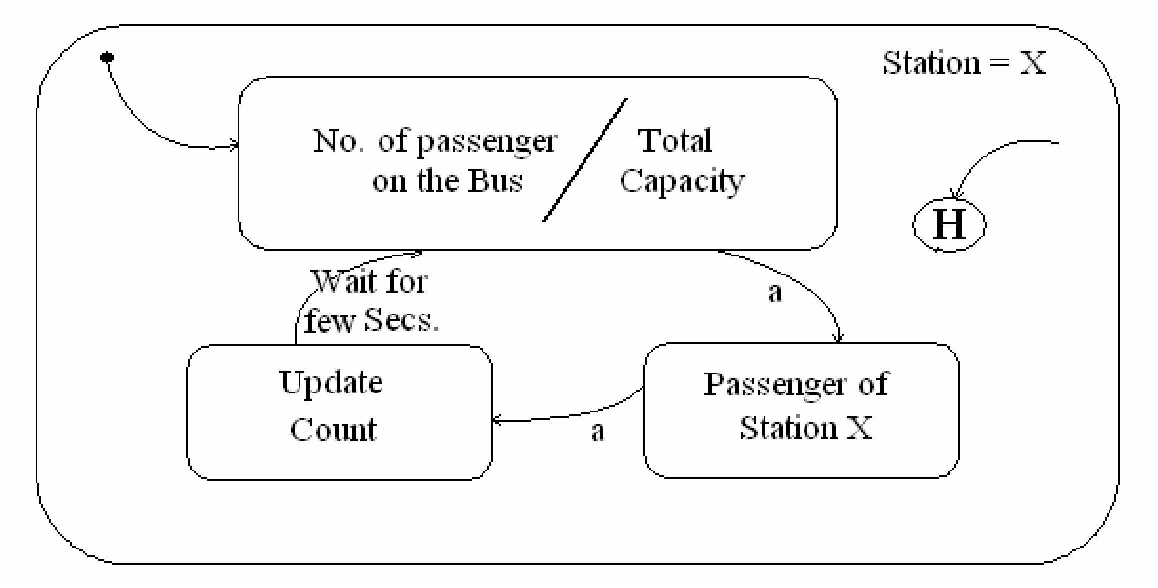


Fig 4.3 Passenger Information

Same way as in passenger information, here also the passenger for a particular station or route is informed. Also additional information which may be helpful to the passenger is given. The fig. 4.4 gives route information, which has functionalities listed earlier in section 4.2.

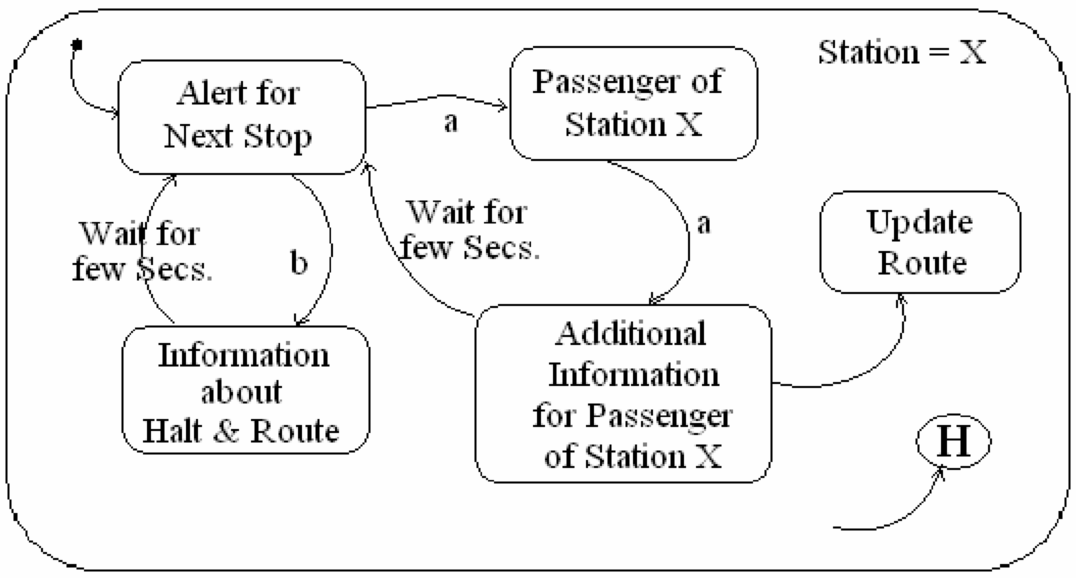


Fig 4.4 Route Information

For long route journey, periodically details of upcoming route and halt are given to the passengers. The route needs to be updated and monitored as the passed stations and the next station gets changed and should not appear more than once. The route information is stored in the system. The system will continuously generate route announcement, which will be shown on LCD display panel.

Fig. 4.5 shows ticketing function of the system which is sub-part of fare charging function. The same system will decide charges to be done, once charges are decided then payment module can be called. Payment module which is another sub-part of fare charging module is used for further transactions. All the stations are numbered by two digits, so here the system waits for the second digit from the user and then recognize the destination. Otherwise it is not possible to proceed. The station's code is listed on the device itself as shown in fig. 4.8.

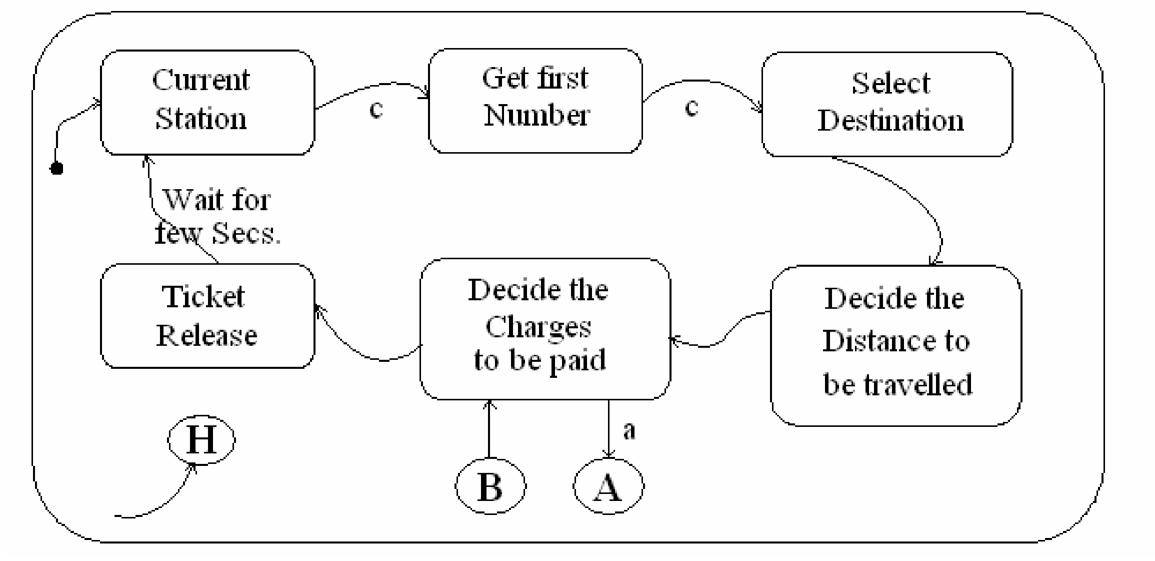


Fig 4.5 Ticketing Process

Ticketing and payment are interactive process, that is payment can also be included as part of ticketing only. Here the assumption is that the ticketing process is directly followed payment. The complete process is uninterruptible. The details of the charges are feed into the device.

Fare charging is carried out with use of the prepaid card. System is assumed to be capable of getting card details. User has entered details on the destination. And now the system decides upon the fare which is to be deducted from the prepaid card. Then the available balance on prepaid card is checked for sufficiency, if there is sufficient balance available then the charges can be deducted. Otherwise the process can not continue and there will be error message. Payment process is done as shown in fig. 4.6.

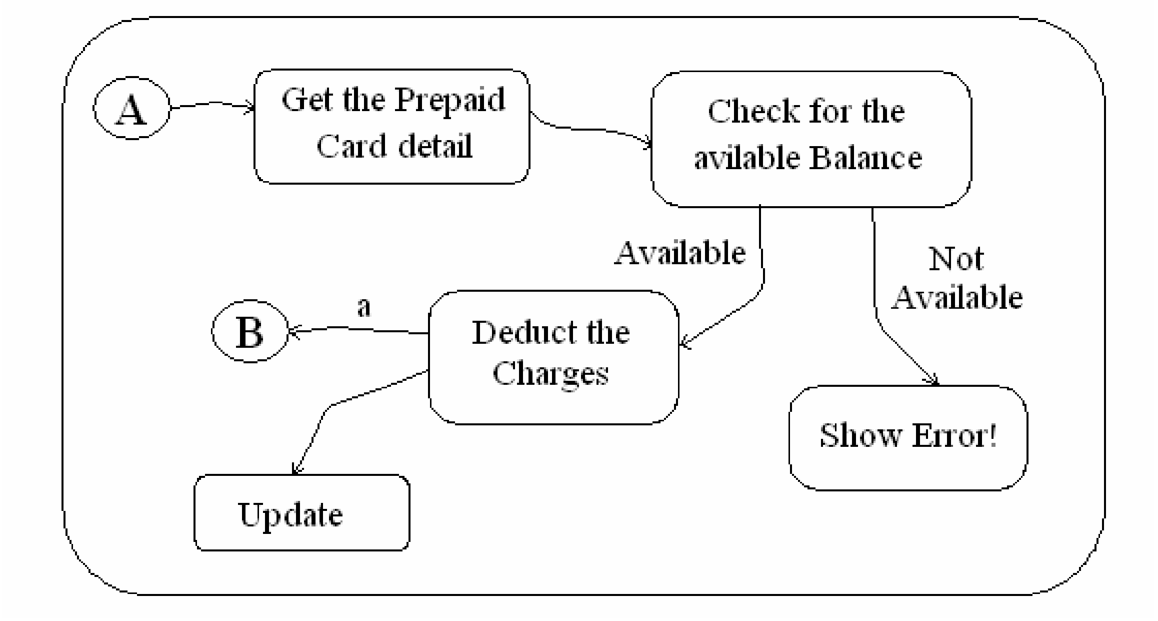


Fig 4.6 Payment Process

As with any prepaid card the remaining balance information is then updated. Whatever approach used earlier, same way update is done. Once the charges are deducted then the tickets are issued. No tickets are given to passengers but only just the charges are deducted from card. Then system jumps back to the default state of listing current station and waiting for user to request.

4.5 HARDWARE DEVICE

Figure 4.7 shows the various hardware components needed to build the working system. These devices are assumed for theoretical modeling only, such combinations of hardware can be used for the device manufacturing. Various hardware options are available, say for clock, processor, and memory; there can be any combination used. The proposed devices are just for design illustration only, no particular is considered for modeling also. Memory is used for monitoring and containing history of various values. The previous values of time, station, passengers, route etc. are needed to be stored.

System clock and timer is used for timing details. The timers are used in waiting that is used in changing the system state. The button panel and control buttons are used for input from user. There is a light source which can be used at night and loud speaker is used for announcement i.e. for alerting the users.

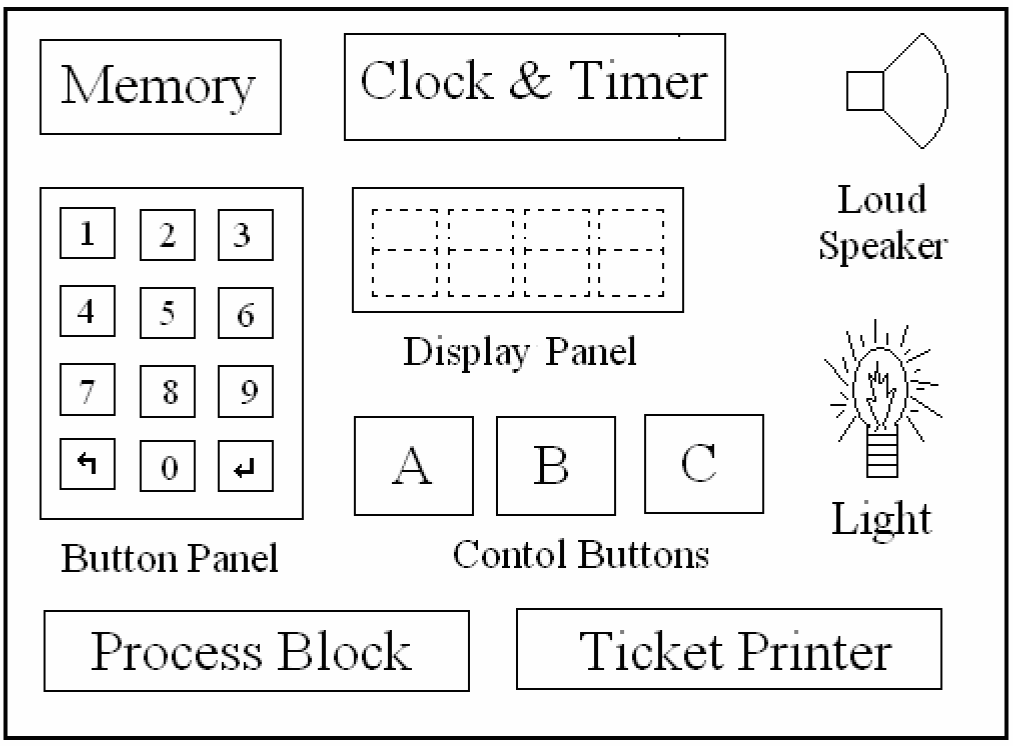


Fig 4.7 Hardware Components

The display panel is alpha numeric LCD display. The size can be standard available at market like 2 X 24 or 4 X 40 etc. The process block is general block, where the processor part will fit. As this is modeling and not actual system building no specific processor is shown. For whatever processor family selected the generated code can be cross compiled and then ported.

There will be two devices as written in assumption. One device is there for the user for ticketing and payment purpose. The device will have instructions written about how to operate it. Also it will have the button panel and the control buttons for the input operations. The list of stations will be written for users to refer and chose. On selecting past station, the system normally shows error information. So there will be small display which will show the error information, the charge to be paid, the destination selection, etc. The user device is only used for selecting destination. If concept of RFID is used then in that case, there is no need to have any device for passengers.

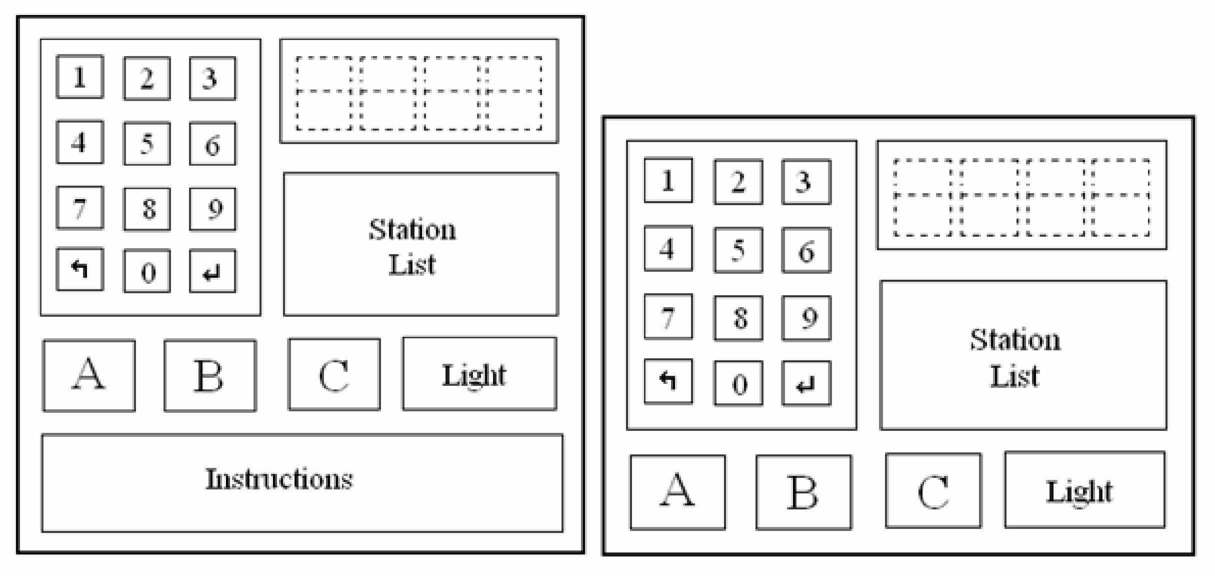


Fig 4.8 System Devices

The other device is for driver to handle for controlling and alerting functions. That is the driver will use the device to inform the passengers. Other than the instructions for usage, the device fro driver will be the same as for the users. The controlling on the part of driver consists of, opening and closing door, alerting the users about the next stop, alerting the users about the timings, marking a visited station on the device so as not to have it more than once, etc.

5. IMPLEMENTATION DETAILS

Every single implementation details are covered in this section. It starts with modules that are devised for the system. Then methodology of implementation with Ptolemy is discussed. In later part all modules' designs are explained that are developed. Finally it covers the design of actors that are generated. The actors that are not available with Ptolemy are required to be developed.

5.1 MODULES

As discussed below the system is divided into three subsystems / modules. It has to be devised in some number of modules. These modules are developed and checked individually and then they are integrated into a single system. The modules are,

1. Display Notification
2. Audio Announcements
3. Fare Charging

They are devised based on the functionality they serve. These three are main operation which in whole the system will be doing. So for each of the functionality, separate modules have been designed and modeled here with Ptolemy II. The functionality that BTS provides is served with one or more modules.

Table 5.1 Modules and Functions Mapping

	Display Notification	**Audio Announcements**	**Fare Charging**
Route Information	√	√	
Timing Detail	√	√	
Fare Charging	√		√
Passenger Information	√		

Table 5.1 shows which of the modules will be active for each function the system provides. It can be observed that the display i.e. visual form of information is required in all modules. Fare charging module requires the display to inform user about the charge done by the system.

Audio announcement module has memory where the audio messages are loaded. And these messages then can be announced according to the settings. Once a station is gone, new set of messages are loaded to memory and then that will be played. This way all messages have to be available there in the system. If system generated audio is there, then in that case there can be no dynamic announcements. That alert has to be done manually if required.

The display notification works in similar way as audio alert. Only change is in terms of the output generated and input given. In audio, audio files stored in memory are loaded and messages are announced as output. Here in display notification the messages are stored as string array, as time of generating the alert, either string wise or character wise the messages can be displayed.

Ticketing works independently of any other module. Here user has to select the destination. And source is decided by control action of driver. Then the system will decide fare based on the mappings given in table 5.2. Here there is 5 station assumed and there charges are also pre-decided.

Table 5.2 Ticket Charges

Station	**A**	**B**	**C**	**D**	**E**
A	*	5	10	15	20
B	5	*	5	10	15
C	10	5	*	5	10
D	15	10	5	*	5
E	20	15	10	5	*

Then payment is done with the prepaid card. The card will give detail on card number and available balance. If there is sufficient balance for ticket then charges will be deducted and ticket will be dispatched to passenger.

5.2 AUDIO ANNOUNCEMENT

Here this module will generate audio announcements, for giving information to the bus passengers. The passengers will avail information about, route and time, as in table 5.1. Design of the module is shown in fig. 5.1.

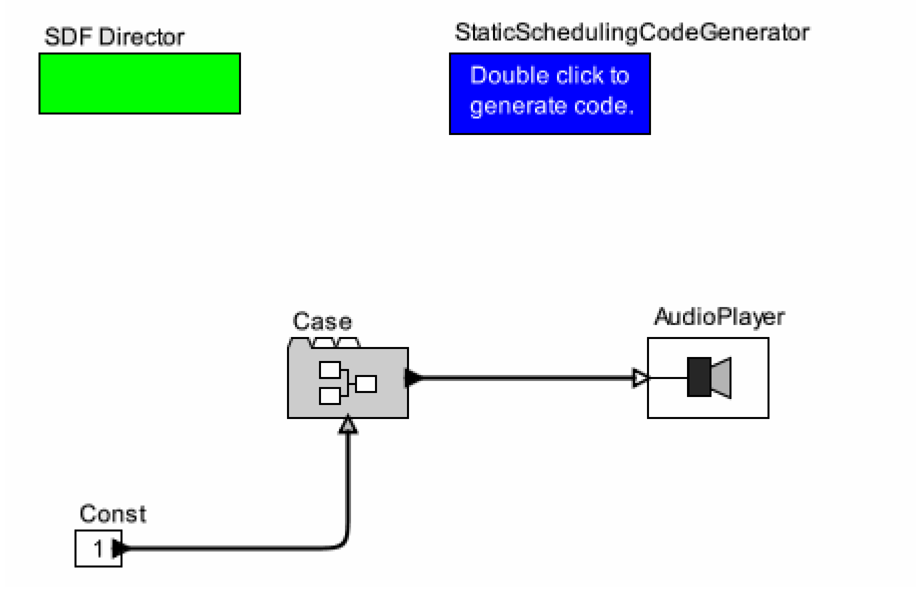


Fig 5.1 Audio Module on Abstract level

As in the figure shown, the audio player is used in here for playing the announcement messages. Here the constant value is the input to the system; constant value indicates one of the stations selected by driver's control. Each time driver presses button to open door, a signal is sent to the system. For that signal value an integer is specified and that is given as input to the subsystem. According to the integer value given, message will be loaded from the memory and then it will be played using the audio player.

The audio player will receive tokens, for whatever file that has been selected for playing. The audio system, i.e. the file reader and file player both follows a common set of configuration. The configuration parameters used here are:

1. Sample Rate – 8000
2. bitsPerSample – 8
3. Channels – 1
4. Transfer Size – 1

The director has configurations for number of iterations, period of execution and time resolutions of the execution. The director controls all the execution of the models.

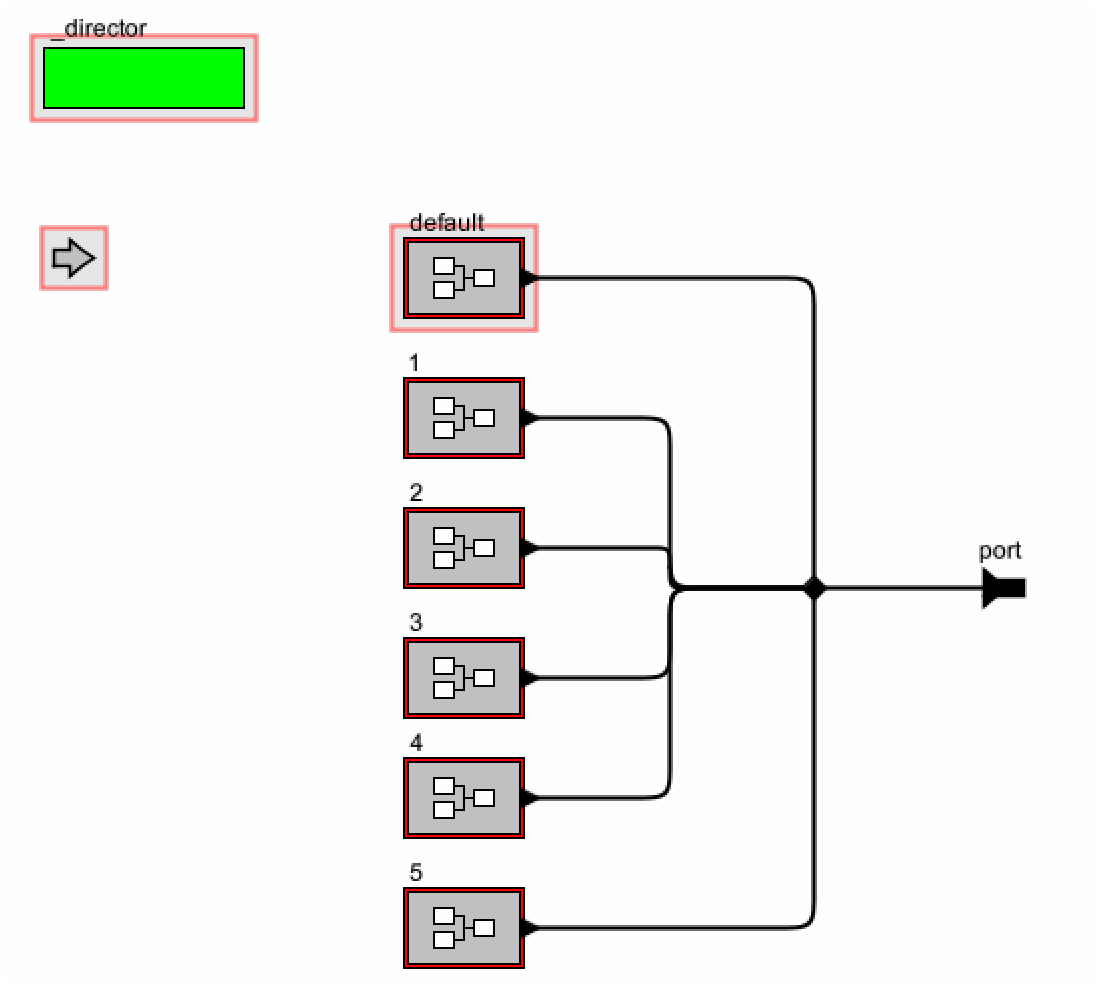


Fig 5.2 Audio Module First level

From fig. 5.1 the case structure is explained in detail in fig. 5.2. Here for complete modeling only 5 stations are used as reference. The _director refers to the director of upper level, i.e. from which the case structure has been referenced. So the case also has parameters values of SDF director. Like all case structure here also the default state is required. As seen in diagram above default, input, and _director is highlighted. The input port is for selecting any of the case available. If none is selected then default case is called. The constant input from fig. 5.1 is used as the input port for the case structure. Each of the five cases has fixed functionalities inside it which will be executed when it has been called. So for each five stations there is a case which will be selected and according to that the audio will be selected and then it will then be announced. For each case the function is as shown in fig. 5.3.

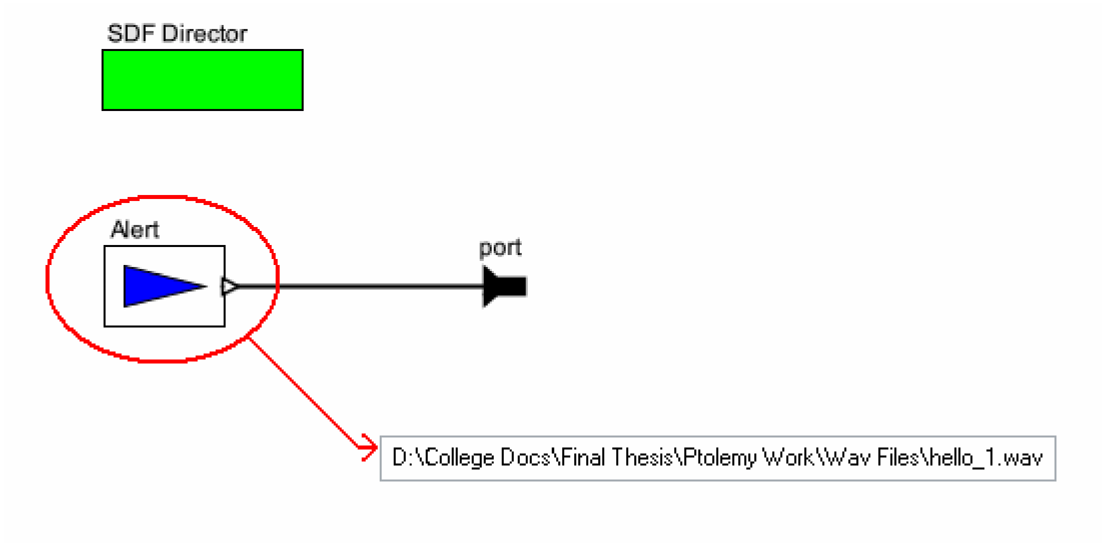

SDF Director
Alert
port
D:\College Docs\Final Thesis\Ptolemy Work\Wav Files\hello_1.wav

Fig 5.3 Inside each cases of Audio Alert

For each case there is particular file selected as shown is fig. 5.3. So in here for every time the file will be loaded from memory. There are set of files loaded into system memory, so for each case the files will be selected and loaded. So the loaded file can be played through the audio player.

Starting from base level in fig. 5.3 the file is read through the audio reader. Then the audio file will be converted to set of tokens and that tokens are sent to the audio player. Base level's port will send the tokens to the first level's output port. The output of fig. 5.2 will be used by audio player as its input. Audio player can read the tokens generated and then that can be played. The audio player has the parameters Sample Rate, bitsPerSample, Channels and Transfer Size. So the audio is played according to that.

For implementation with some target platform, for playing audio SDL library is a requirement. The code is to be transformed to "C" code, and that code is then cross compiled for target. Here for playing audio with C code, SDL – Simple DirectMedia Layer is used. These libraries are required, for availing any Media options with C code. These libraries are freely available. Simple DirectMedia Layer is a cross-platform multimedia library designed to provide low level access to audio, keyboard, mouse, joystick, 3D hardware via OpenGL, and 2D video framebuffer. It is used by MPEG playback software, emulators, and many popular games [9].

5.3 DISPLAY NOTIFICATION

This particular module is used for displaying information on LCD display panel. The display gets string values as input and that notifies the passengers. For all system functions, display module will be working. The route information, timing information, ticketing and payment all needs support from display. Design of this module is as shown in fig. 5.4.

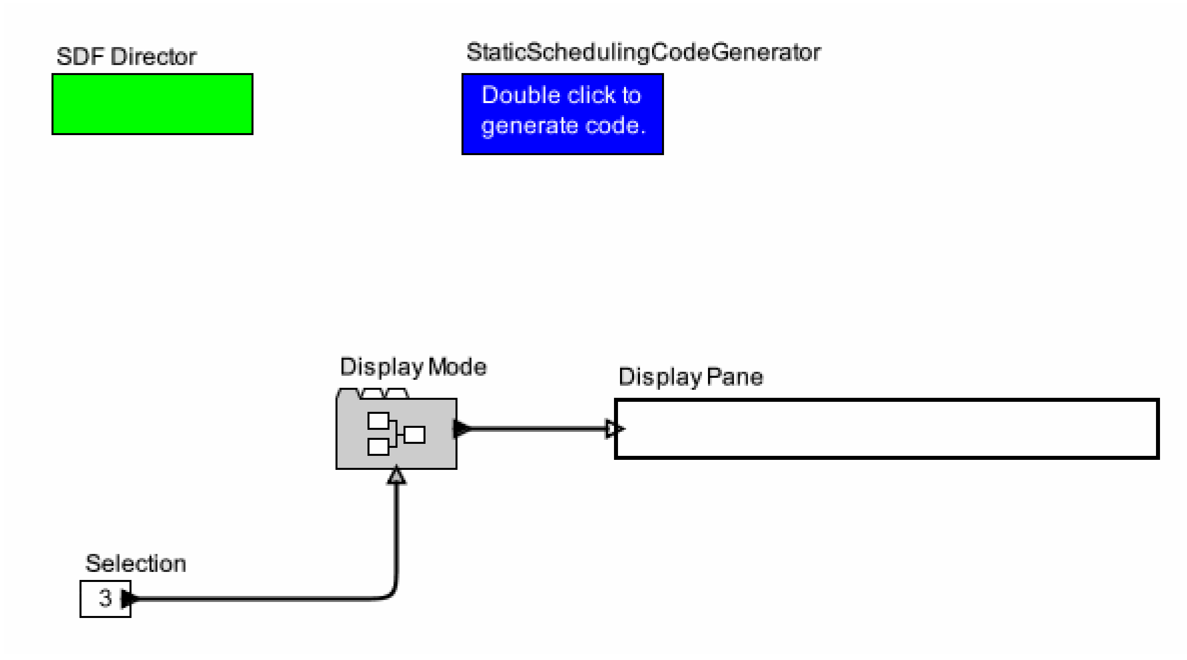


Fig 5.4 Display Notification Module

As in case of audio alert module, same ideology applies to here also. Same way there is selection input as it was constant in audio alert. The selection’s value is integer which is converted value of the signal generated by driver’s control action. The display mode is a case structure as earlier. And finally the display pane is used for continuous string output. According to the station selected by selection, a set of text messages will be loaded to memory. These messages are already entered for each station. Now the messages are there with system, once the station is selected, the set will be chosen. Each message from the set will be displayed on the panel, in random manner with a period of 2 seconds. The set of messages will be changed only when driver's control selects some new station as current station. Fig. 5.5 shows the case structure for display module in detail.

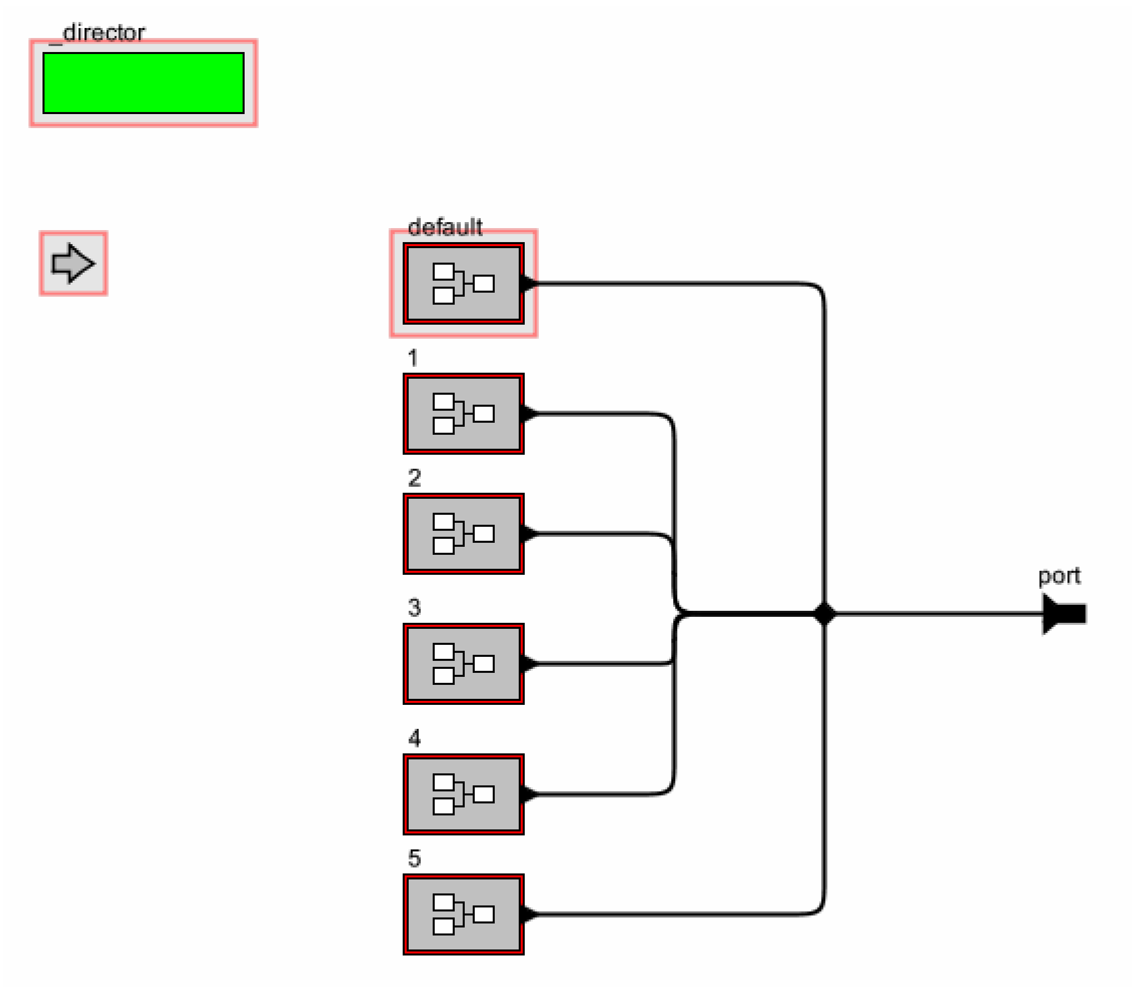


Fig 5.5 Display Notification Case Structure

This will work in same manner as fig. 5.2 does. Every thing is similar to above shown audio alert. Five stations are used for modeling. Also a default case is there which will alert “Error” to the system, as the messages can be chosen for any of the five stations. All the cases executes according to the selection input.

For each case set of messages are like. Here for example messages for station “A” is shown,

"Welcome to BTS.",

"Current Station is 'B'.",

"Next Station is 'C'.",

"Bus departed 'A' at 9 AM.",

"Bus reaches 'E' at 10:30 AM."

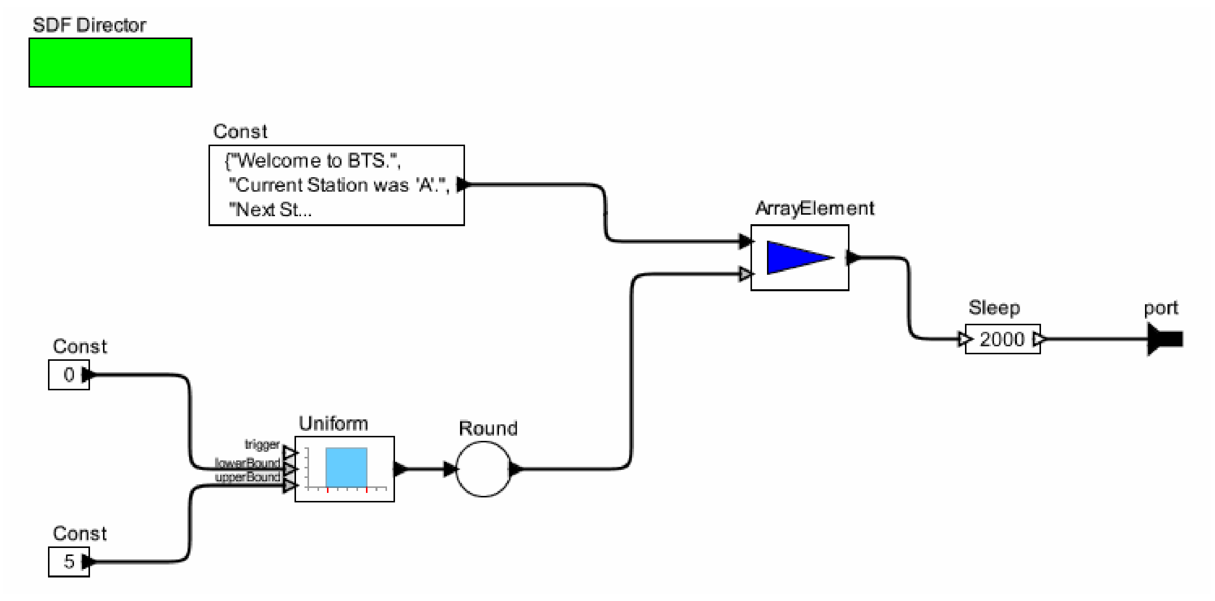


Fig 5.6 Inside each cases of Display Notification

Fig. 5.6 shows the inside functional implementation of each cases of display module. Here using uniform distribution a random number is generated at every unit of time. The random number will lie between 0 and 5, so the uniform distribution will generate any real number ranging from 0.0 to 4.99. This number generation is used for having the facility to select the message in random manner.

The array element actor is used to select the message according to the index loaded from the uniform distribution function. The index of an array is always an integer value and the number generated from the uniform distribution will be always real value. So the real value has to be rounded off to the nearest integer value. So now the integer value will select any message from the constant loaded as the set of messages. For having the messages shown at interval of 2 seconds, the sleep actor is used. The value 2000 indicates the number of milliseconds for which the message will be displayed.

According to the index generated with uniform distribution and rounded to an integer value, the message will be selected from the memory. This text message will now be converted to tokens; these tokens are sent to the output port. And same ways this tokens travels back to the upper level and from there to the main level, where these tokens are sent as input to the display pane. So now the display pane will show the selected messages.

5.4 FARE CHARGING

As discussed earlier in section 4.2 this functionality can be subdivided into two functions it serves – Ticketing and Payment. This particular module contains very complex design structure, as it has to cover the card details, the fare calculation part and then charging part. The fig. 5.7 shows top view of the model's design.

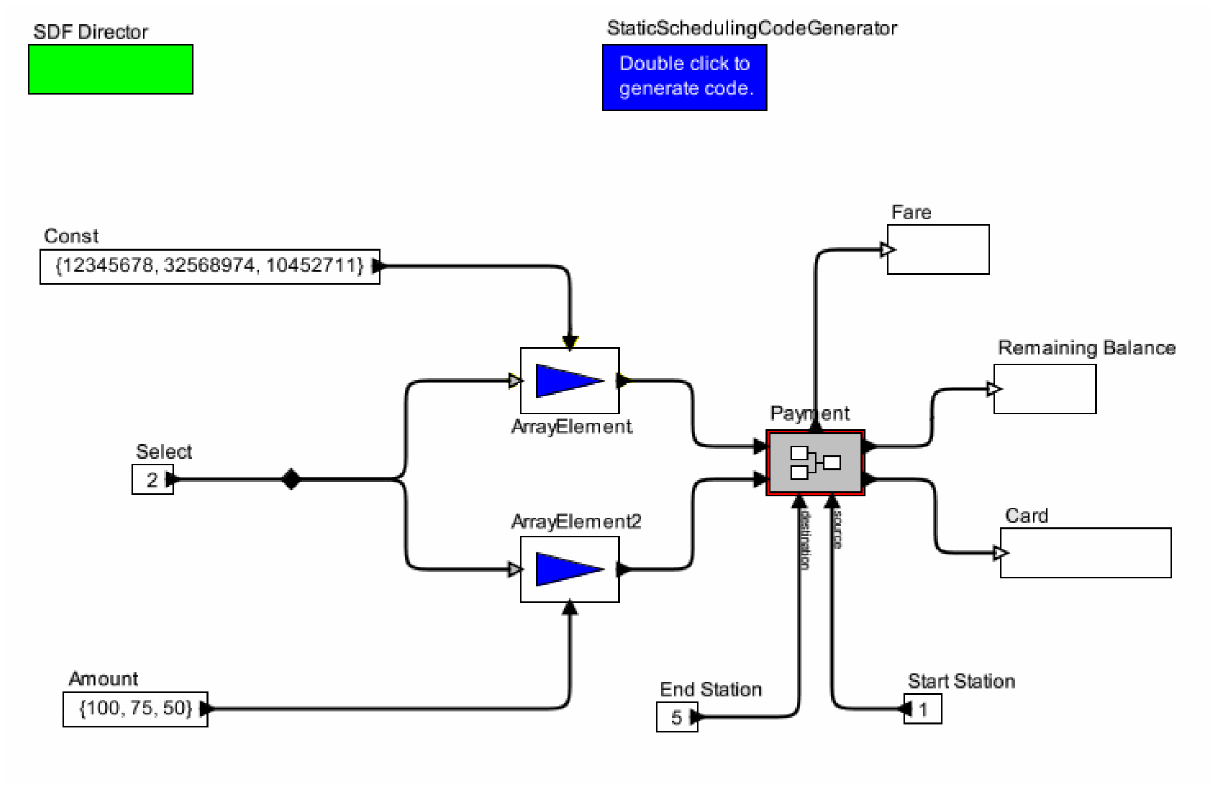


Fig 5.7 Fare Charging Module

Here the system can be divided in 3 parts, first for getting the card details, second for deciding the fare to be deducted and third part to deduct amount from the card. Here the card details are recorded into array. The const and amount are two constant arrays, which are having values for card number and amount respectively. For selecting any card the array index is chosen and then that particular value is loaded from both of the arrays. So card number and payment is found that way.

For calculating the fare to be deducted, system follows the table 5.2. To calculate fare, here the system now requires two stations, once the stations are found then from pairs of table 5.2, the fare can be decided.

To get two station numbers, the system gets one i.e. the source station from the current location of bus. And to get another station number, passenger will enter the destination from the hardware available. So now two station numbers will be available. So in this particular module, total 5 inputs are available. The fare will be calculated and remaining balance shall be restored to the particular card. Fig. 5.8 shows detail of payment block of fig. 5.7.

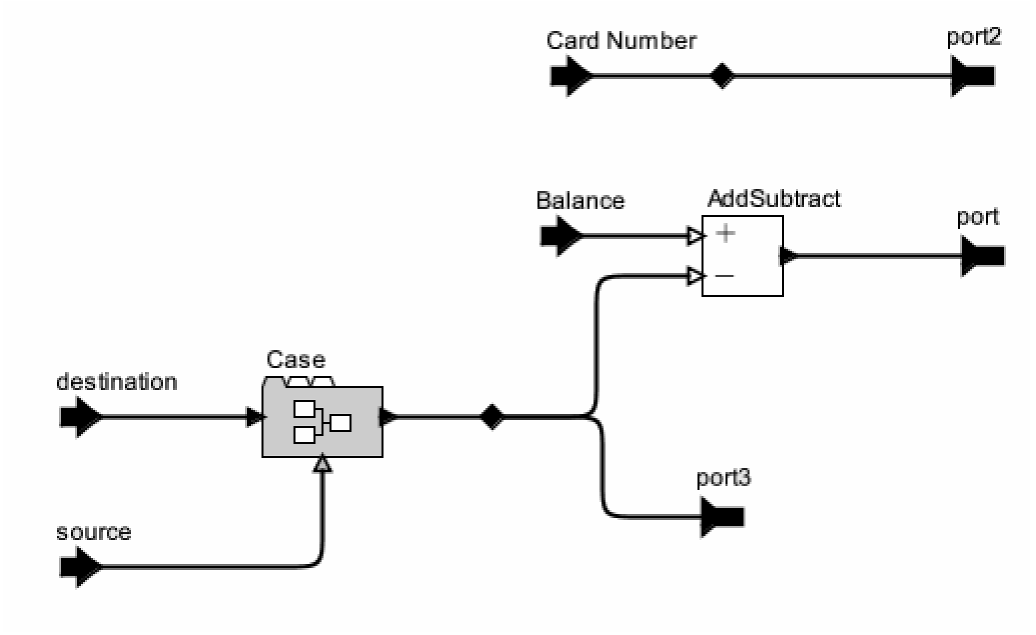


Fig 5.8 Detail of Payment block

Card number is returned as it is; the balance gets deducted by the amount that is decided by the case structure. Inside case structure there is the logic for calculating the fare to be deducted. Source and destination are the two input for the case structure. AddSubtract block has two input values, one of which is positive and other is considered as negative, in case of subtraction. Both the values will give result which will be deducted balance. Port3 displays the fare that will be charged to the passenger, port displays remaining balance and port2 shows card number.

For the case block, source will select the case which will be called for execution. And the destination will be used as input to all the cases, which will process the value of the destination for getting the result. Fig. 5.9 shows details for the case structure. For each five station we have a separate case here, which shall be called according to the value input at the source port of fog. 5.8.

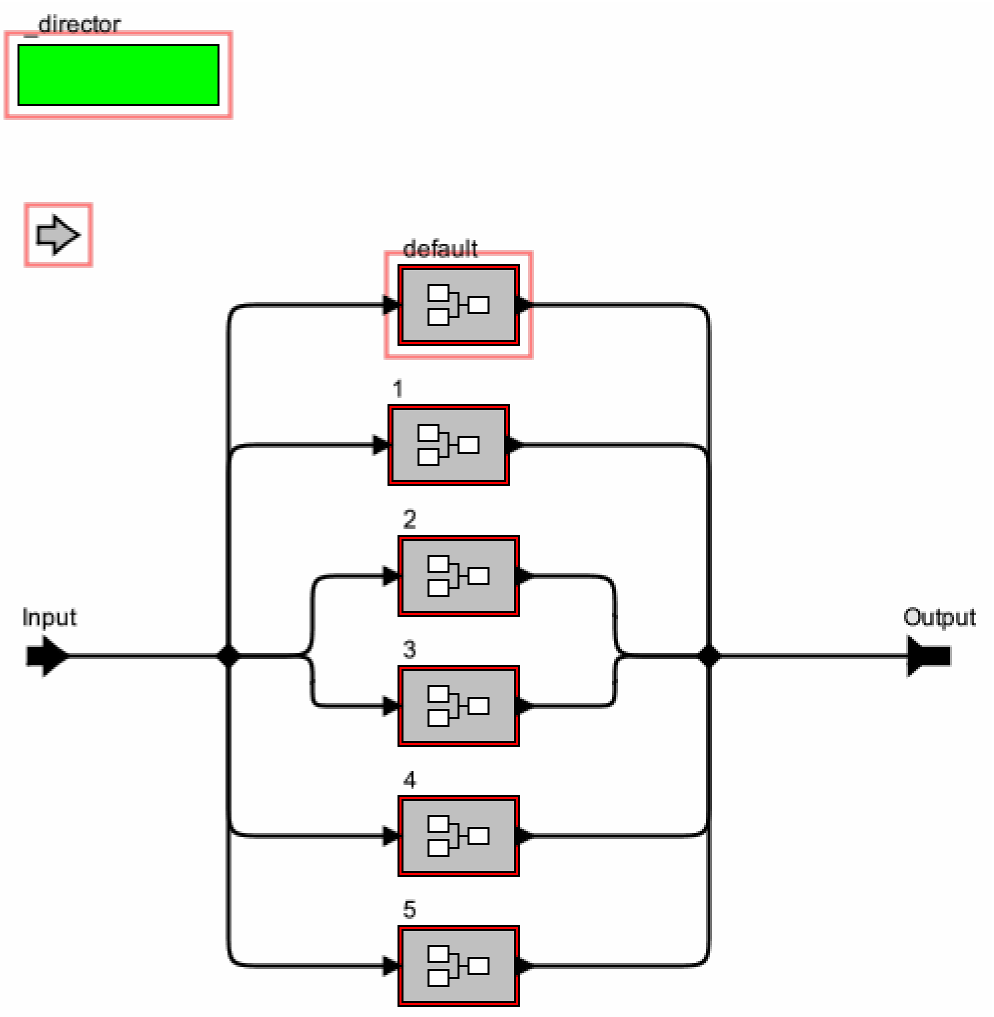


Fig 5.9 Inside case structure

This case structure differs from earlier two as in fig. 5.2 and fig. 5.5. Here the main difference is that along with the input of case selection, there is separate input also. This input is given to all the cases as input itself. The case will be chosen from the port shown above and input is used for calculation purpose.

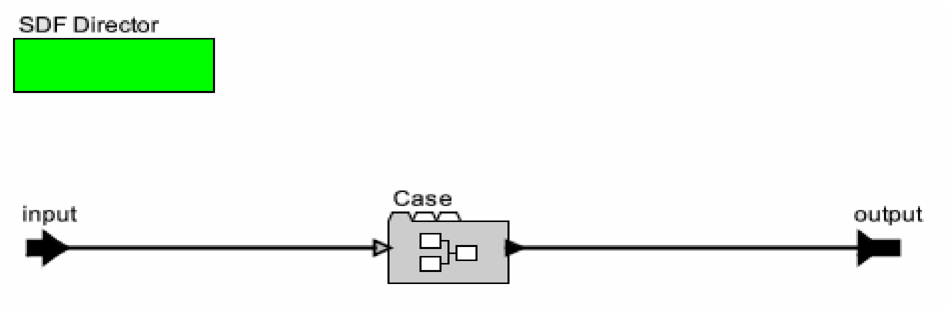


Fig 5.10 Payment Block Hierarchy

Fig. 5.10 shows the content of each of the case of fig. 5.9. With case of fig. 5.9 the source station is selected, now to choose the destination again another case structure is required. That case structure is shown in fig. 5.10. Here the input value seen in fig. 5.9 is used as the case selector. That is, according to the value of the input here in fig. 5.10, any case will be called and executed. Fig. 5.11 shows inside of the case structure of fog. 5.10.

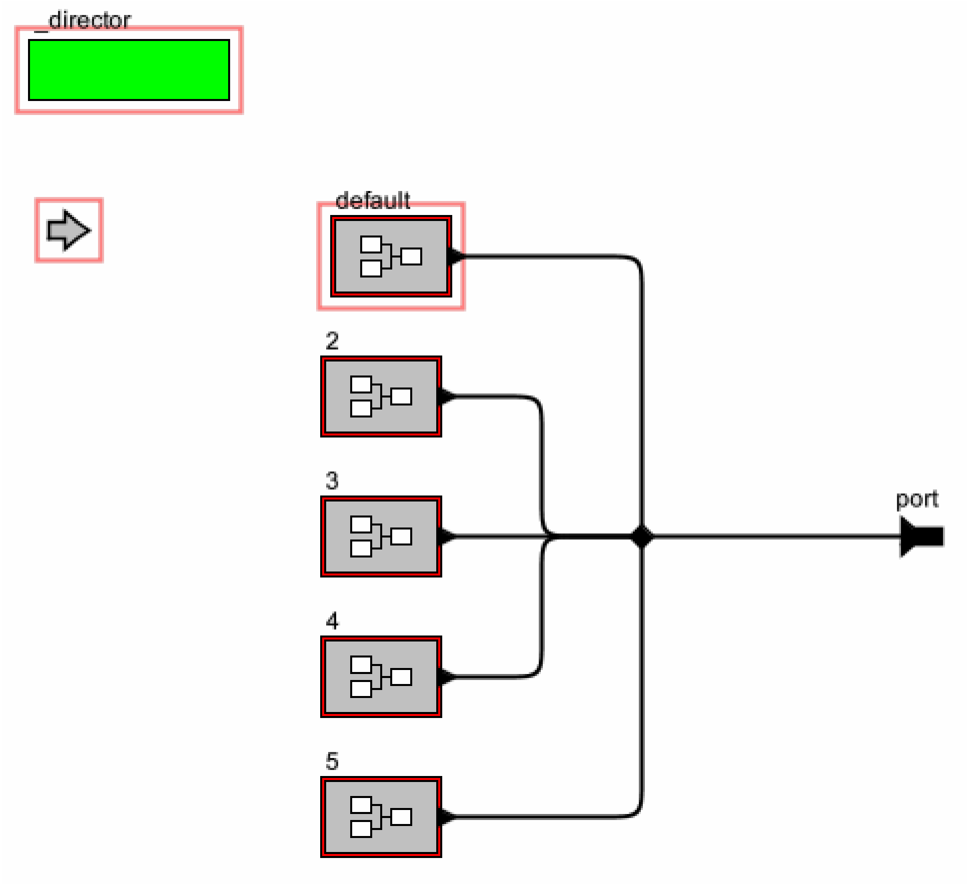


Fig 5.11 Individual Cases for Destination

This case structure differs from earlier, in terms of the cases defined. If the source station is chosen to be station 1, then for destination only four has to be considered. As seen in fig. 5.11, if source is 1, then the destination can be one of 2, 3, 4 or 5. Here each station is coded to an integer value. So considering that the cases can then be executed. For case structure, each possibility has individual refinement in the definition. For each source, there will be four possible destinations, which will be selected. And from the source – destination pairing the fare will be calculated.

Fig. 5.12 shows the final level, for payment. This defines the function inside the cases of fig. 5.11.

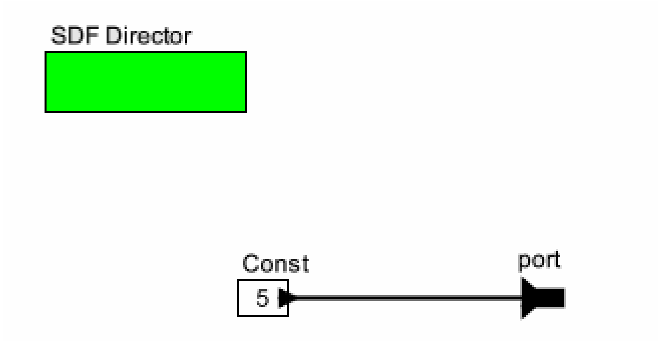


Fig 5.12 Final level of Payment

Here this particular fig. 5.12 shows the fare that is charged. Here the case is for station 1 as the source and station 2 as the destination. So from table 5.2, it is observable that the fare is 5 unit of currency. This is just assignment of a constant value for each combination. So now the result will pass back to the root level of payment, where this will be deducted from the card's current balance. Each port in each abstraction will pass the result to upper level, and finally this value is used as input to the AddSubtract actor for charging. The result of which is remaining balance on the card.

In fig. 5.1, 5.4 and 5.7, a dark gray colored actor can be seen, where the title of the actor is StaticSchedulingCodeGenerator. This particular actor is used for "C" code generation. Details of code generation are covered in appendix B.

5.5 PTOLEMY ACTORS

Many inbuilt actors of Ptolemy are used in the designs shown above. Some of them are also changed according to the need. These actors are available in both normal and code generation environment, so that the code can be generated for that. The actors used have individual functions for, input, output, mathematical, array implementation, case structure and etc. All these actors are compatible with SDF model of computation. The SDF model of computation has no timing constraints included for executions. Also this model or director is best for code generation. List of actors and their particular functions are described in brief in table 5.3 below.

Table 5.3 Ptolemy Actors

Actors	Functions
Audio Reader	Used for reading .wav file and generating tokens for that.
Audio Player	Plays audio for .wav files' tokens.
Const	Used for storing constant values or constant arrays.
Case	Case structure.
Refinement	Individual cases of the Case structure.
Input Port	Used for input of a model or hierarchy.
Output Port	Used to produce output of a hierarchy or model.
Monitor Value	Used for displaying the some value continuously.
Uniform Distribution	Generates real number using the uniform distribution.
Round	Rounding of the real number to integer value.
Array Element	Used for selecting an element with given index.
Sleep	Used for generating delay between the outcomes.
AddSubtract	Mathematical add and subtract function.

5.6 GENERATED ACTORS

Though Ptolemy has variety of actors available to design any model, still some actors are required for modeling BTS. The actors that are required are for, generating the timing information and for getting the display for LCD device. Ptolemy has actor that gives normal display but to have the display on LCD, an actor is required.

5.6.1 Timing Information

The Ptolemy environment supports timing information in terms of the model timing, but not in wall clock time. That is, Ptolemy actor can give the time that system took for execution, but not about the wall clock time. The system here needs to have such information, in order to notify and also for the system history. The timing helps system to notice and update the system's functions if necessary. The timing history helps for traffic study, driver performance, and etc. So for that an actor as shown in fig. 5.13 is designed. This particular actor gives, two information every time it is called.

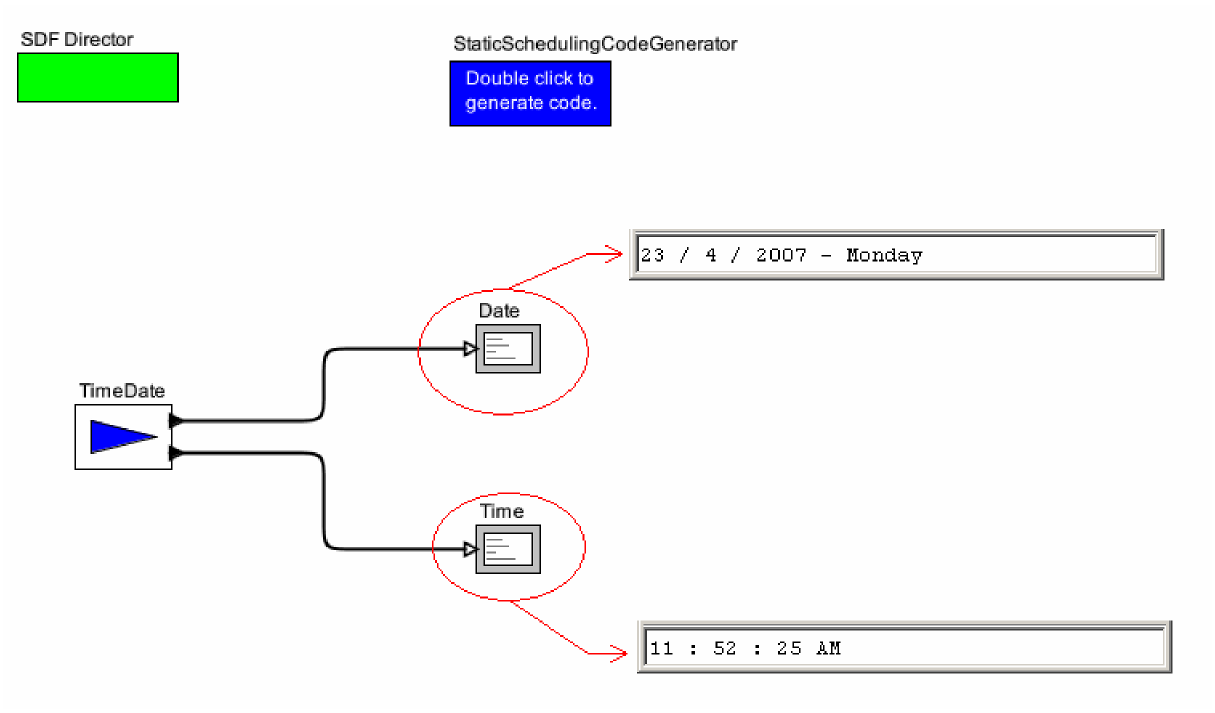


Fig 5.13 Timing Information Actor

It gives, current time in HH : MM : SS AM/PM format. And it generates date in DD / MM / YYYY – DAY format. As in fig. 5.13 it shows the sample output how it will be generated. Here these details are displayed with a display actor of the system. But the same detail can be used with any other actors which can receive string input and operate with that.

5.6.2 LCD Display

In many embedded system display device used is LCD display. The Ptolemy actor can generate display in normal form and in similar manner it may output to any hardware device. But LCD device required input in 8 bit form and with combination of those 8 bits the output is generated. Here four actors are generated to implement and check the LCD functionality. One set of actors have two actors in it, one will convert string into bits form and other will convert the bits back into the display.

So here the first actor that converts string input to bits is the key actor. Output of this actor will be sent to the LCD device as input and then its upto the device to show the result in desired form. For getting the result into the desired form, the string is converted to bits according the table 5.4. Table shows the 8 bit code for each symbol on the right. The 8 bit code is converted to hex number form in the table.

Table 5.4 LCD Code

30	0	**40**		**50**	P	**60**	`	**70**	P
31	1	**41**	A	**51**	Q	**61**	A	**71**	Q
32	2	**42**	B	**52**	R	**62**	B	**72**	R
33	3	**43**	C	**53**	S	**63**	C	**73**	S
34	4	**44**	D	**54**	T	**64**	D	**74**	T
35	5	**45**	E	**55**	U	**65**	E	**75**	U
36	6	**46**	F	**56**	V	**66**	F	**76**	V
37	7	**47**	G	**57**	W	**67**	G	**77**	W
38	8	**48**	H	**58**	X	**68**	H	**78**	X
39	9	**49**	I	**59**	Y	**69**	I	**79**	Y
3a	:	**4a**	J	**5a**	Z	**6a**	J	**7a**	Z
3b	;	**4b**	K	**5b**	[	**6b**	K	**7b**	{
3c	<	**4c**	L	**5c**	.	**6c**	L	**7c**	|
3d	=	**4d**	M	**5d**	]	**6d**	M	**7d**	}
3e	>	**4e**	N	**5e**	^	**6e**	N	**7e**	,
3f	?	**4f**	O	**5f**	_	**6f**	O	**7f**	!

For coding the controlling of the LCD device is not considered. But here only the data portion is done. That is the string is converted to 8 bit form and is sent as either single sting of bits or it is sent as 8 separate bits on 8 different outputs.

In each set of actors, the main part is string to LCD code. The second actor is LCD code to string output, which is used to cross verify the result produced. So first can be considered part of system's development and second actor can be considered as the LCD device. The logic part of this actor for converting string input to LCD string output is,

1. Read in the string inputs and match the ascii value.
2. For ascii value match the value in table 5.4.
3. For each value convert the hex to 8 bit value.

The reverse of above logic applies for converting LCD string to normal display. That is the actor generated which simulated the LCD device. Actors and model for that is shown, this can be used with normal display module as in fig. 5.4.

Fig. 5.14 shows the set of LD actors, generated which simulates the value for the normal string output. That is complete string is given as a single input to the LCD device. In fig. 5.14 LCD Sting is the outcome of StringToLCD actor, and which is the key part of the actor. This can be given as input to LCD device for display.

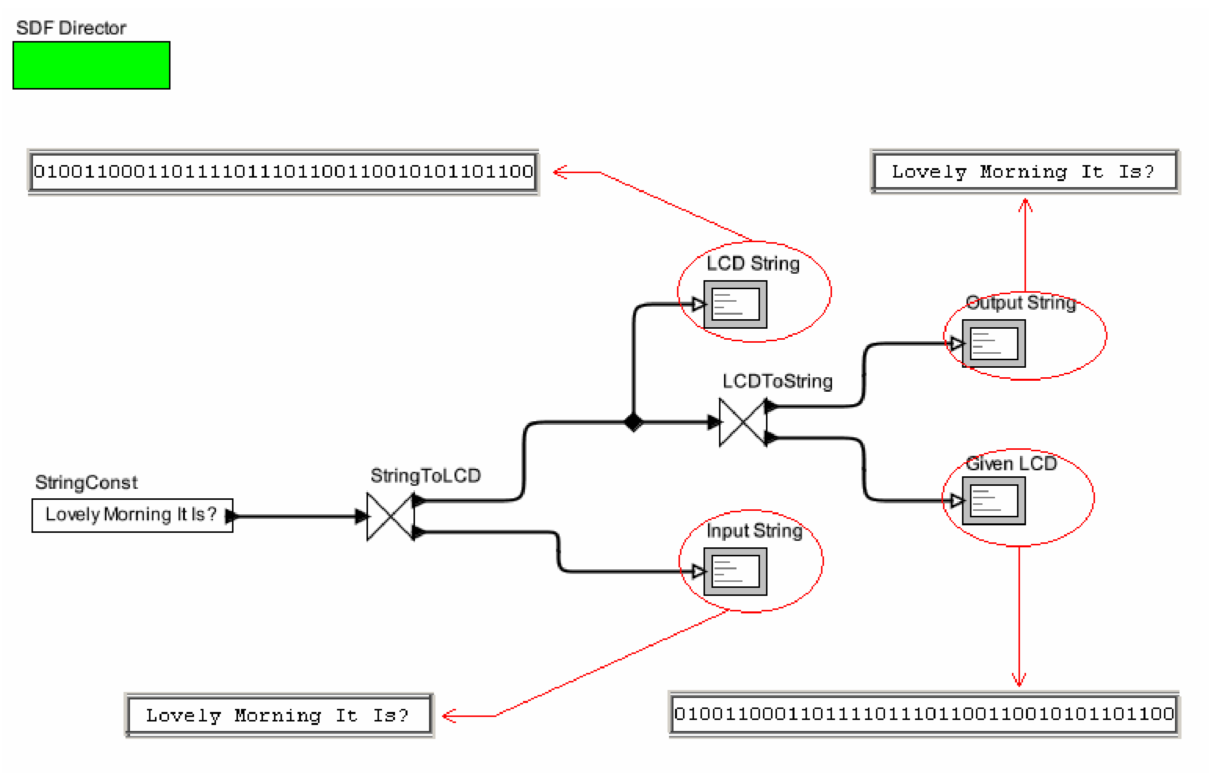


Fig 5.14 LCD Display in Normal Form

The module shows two separate set of actors. One for normal string output and other for the 8 bit outputs. The complete set is shown with two individual actors in each model. First actor converts string to LCD bits and the second LCD to string. The second actor in each of the set can be assumed as the output LCD device where the 8 individual bits for each input string is input. Then the device generates the string output. Here so far no controlling of LCD device is included. In fig. 5.15 another set of actors is shown, which is operating for 8 bits. That is here string to LCD generated 8 individual bits for each character and this individual bits are given as input to separate 8 ports on LCD device. So now the device may interpret the characters with use of the table 5.4. In fig. 5.15, output of the first actor i.e. String to 8 bit LCD is shown. Bit 0 to 7 are shown individually, whose combination is any character. These 8 bits are given as input to LCD8ToString as shown in figure. The resultant message gets generated as shown. Here each character is shown on individual line, as no controlling is included. After adding controlling of LCD this device will work as a cursor based display device.

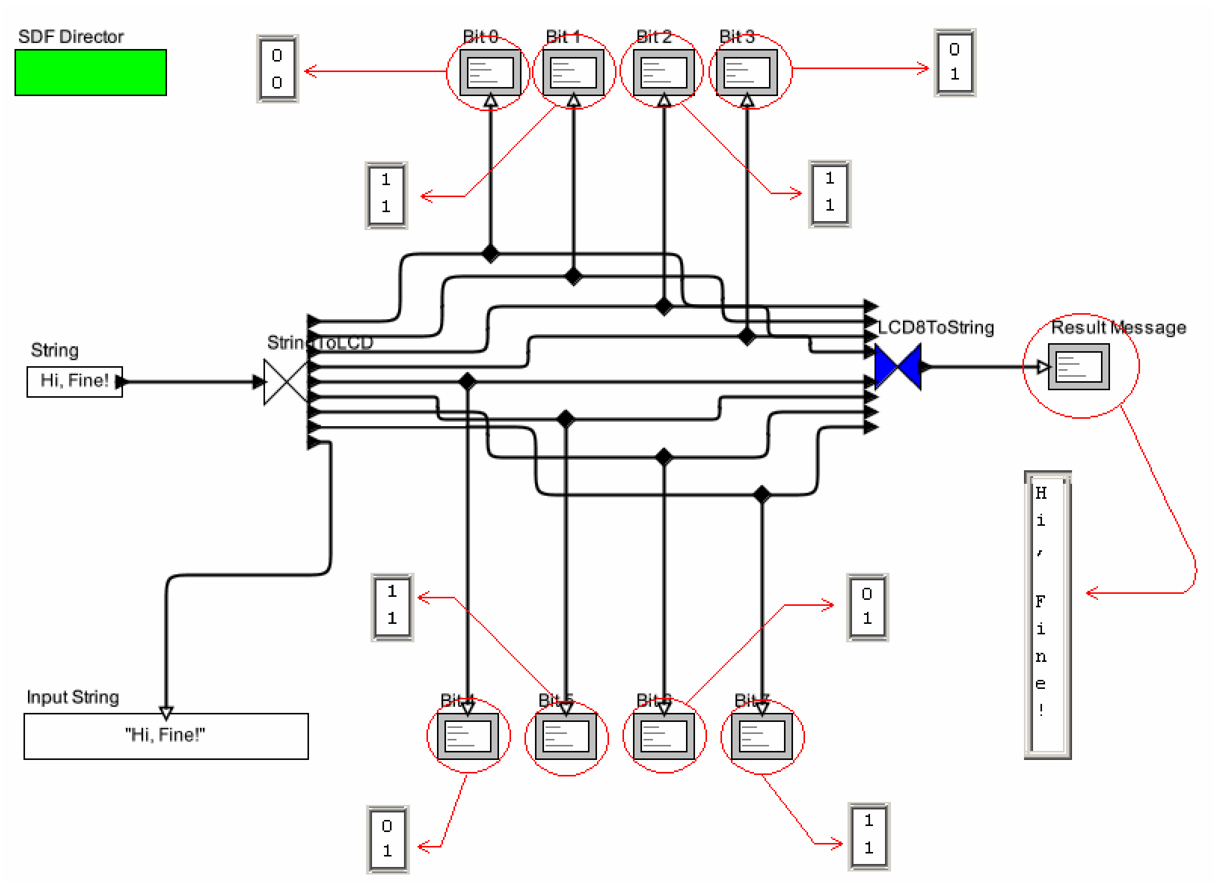


Fig 5.15 LCD Display in 8 Bit Form

For the 8 bit or the normal string the only difference lies in the way output is generated. In the normal the string of LCD bits is sent to single port. While in the 8 bit format the separate 8 bits are sent to 8 input ports of device for each character. The LCD and timing actors are generated and added to the local Ptolemy environment.

Any of the actors shown in fig. 5.14 or fig. 5.15 can be used for converting the given sting to LCD bits and then these bits can be used as input to LCD device. So now the model can work in exact manner to original hardware, as the LCD is also generated and used in modeling.

6. INTEGRATION AND TESTING

Here the chapter includes details on full system's integration and testing. Prior to integrating the system, each modules needs to be tested. So here the testing details are also included. Modules are tested with C code and also from the Ptolemy's execution are done. Then integration of the full system is covered in later part of the chapter.

6.1 MODULES' TESTING

In this section, configuration and executions of each of the module, shown in earlier chapter is shown. Complete modules are shown and then how the executions will take place is explained and finally the testing. Testing here shows outcome of each module.

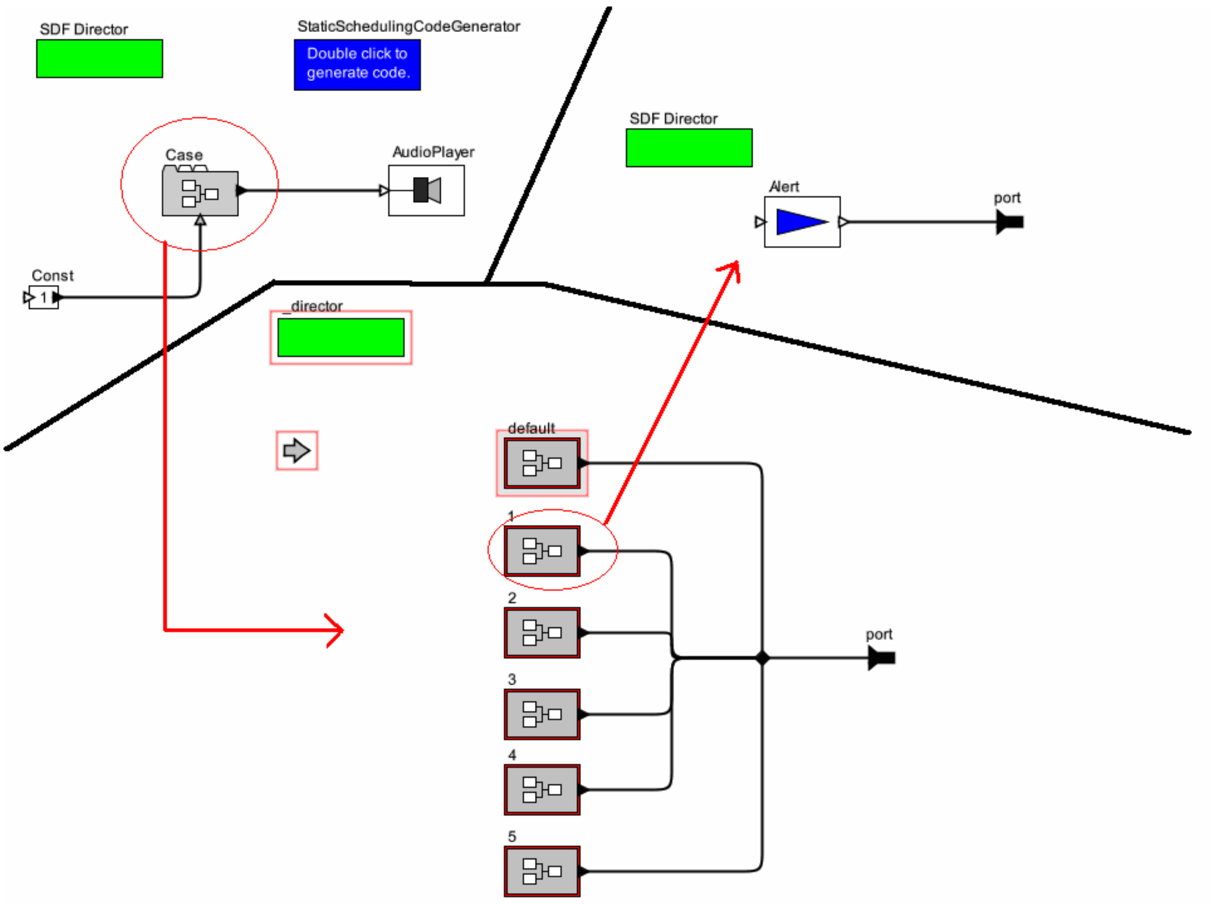


Fig 6.1 Complete Audio Module Design

Fig. 6.1 shows the audio module which is explained in section 5.2. Below in fig. 6.2 the execution is shown. Here an audio file is selected by the constant input and then it is played with the audio player. To show the working plotter is used which plots the frequency values against the time. So the highlighted portion is for output and the plot shown is for the audio file played.

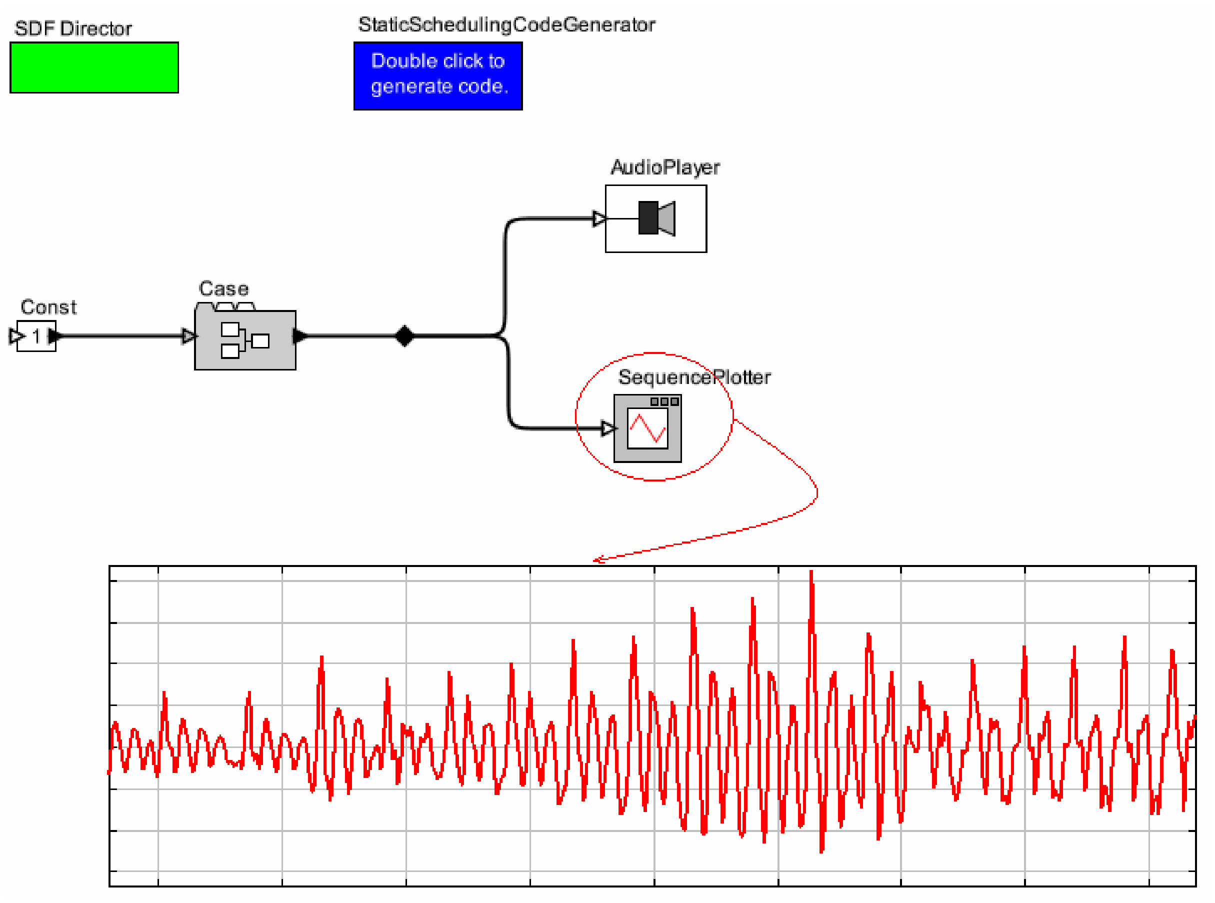


Fig 6.2 Testing of Audio Module Design

Here the "C" code is generated for the timing module but to show modules' working state this approach is used. Execution of C code requires SDL library [9] support and currently it is having mismatch with standard C library. So that execution is not referred here.

Next is the display module, which is shown in fig. 6.3. Complete design is explained in section 5.3, the same is shown here. As the module functions are defined, it will display set of messages for the station chosen from the driver's control. According to that set of messages are displayed.

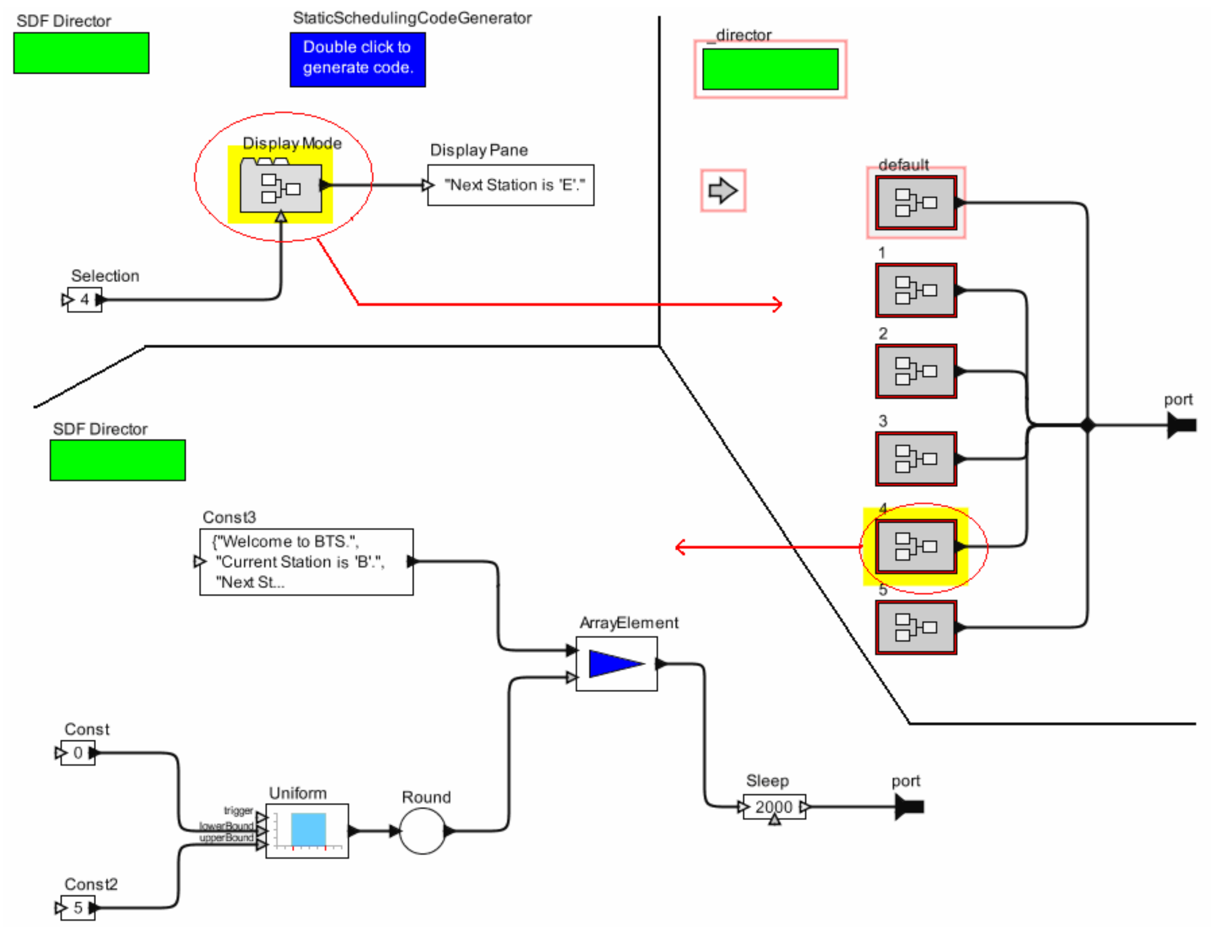
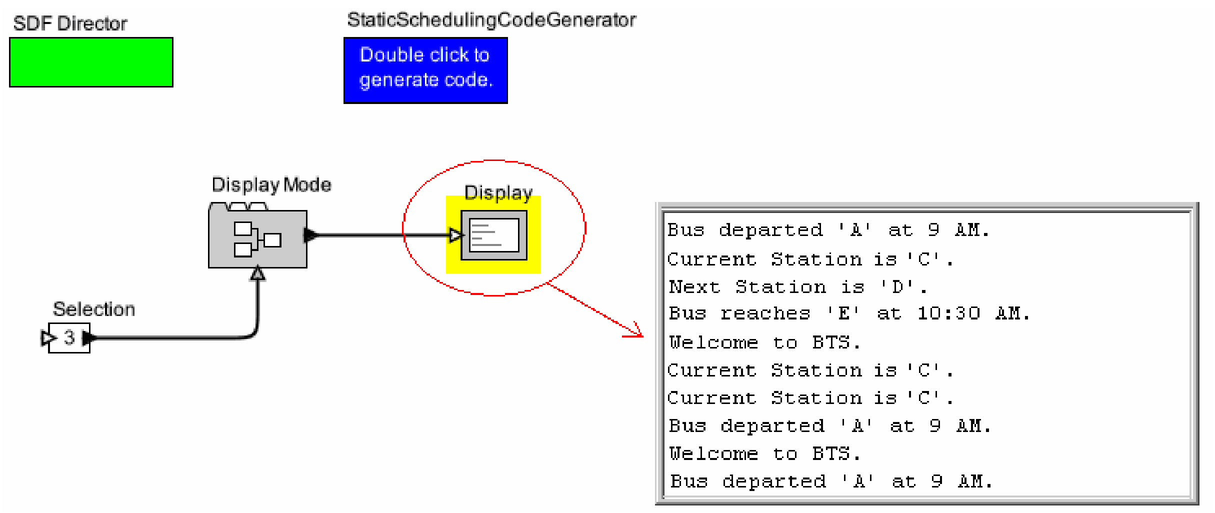


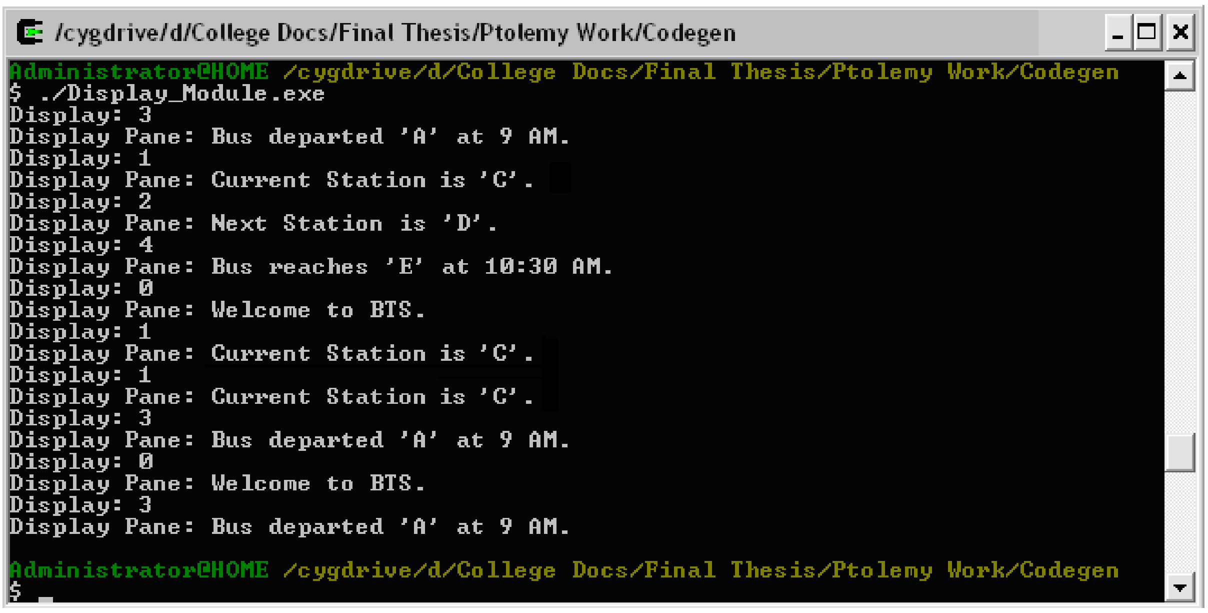
Fig 6.3 Complete Display Module Design



```
Bus departed 'A' at 9 AM.
Current Station is 'C'.
Next Station is 'D'.
Bus reaches 'E' at 10:30 AM.
Welcome to BTS.
Current Station is 'C'.
Current Station is 'C'.
Bus departed 'A' at 9 AM.
Welcome to BTS.
Bus departed 'A' at 9 AM.
```

Fig 6.4 Testing of Display Module

Here fig. 6.4 shows the execution of the module under Ptolemy II environment. The set of messages are shown with display actor of Ptolemy. Fig. 6.5 below shows the execution of generated "C" code of the display module. The same set of message as in fig. 6.4 is generated when model is executed.



```
/cygdrive/d/College Docs/Final Thesis/Ptolemy Work/Codegen
Administrator@HOME /cygdrive/d/College Docs/Final Thesis/Ptolemy Work/Codegen
$ ./Display_Module.exe
Display: 3
Display Pane: Bus departed 'A' at 9 AM.
Display: 1
Display Pane: Current Station is 'C'.
Display: 2
Display Pane: Next Station is 'D'.
Display: 4
Display Pane: Bus reaches 'E' at 10:30 AM.
Display: 0
Display Pane: Welcome to BTS.
Display: 1
Display Pane: Current Station is 'C'.
Display: 1
Display Pane: Current Station is 'C'.
Display: 3
Display Pane: Bus departed 'A' at 9 AM.
Display: 0
Display Pane: Welcome to BTS.
Display: 3
Display Pane: Bus departed 'A' at 9 AM.

Administrator@HOME /cygdrive/d/College Docs/Final Thesis/Ptolemy Work/Codegen
$ _
```

Fig 6.5 Testing of "C" code of Display Module

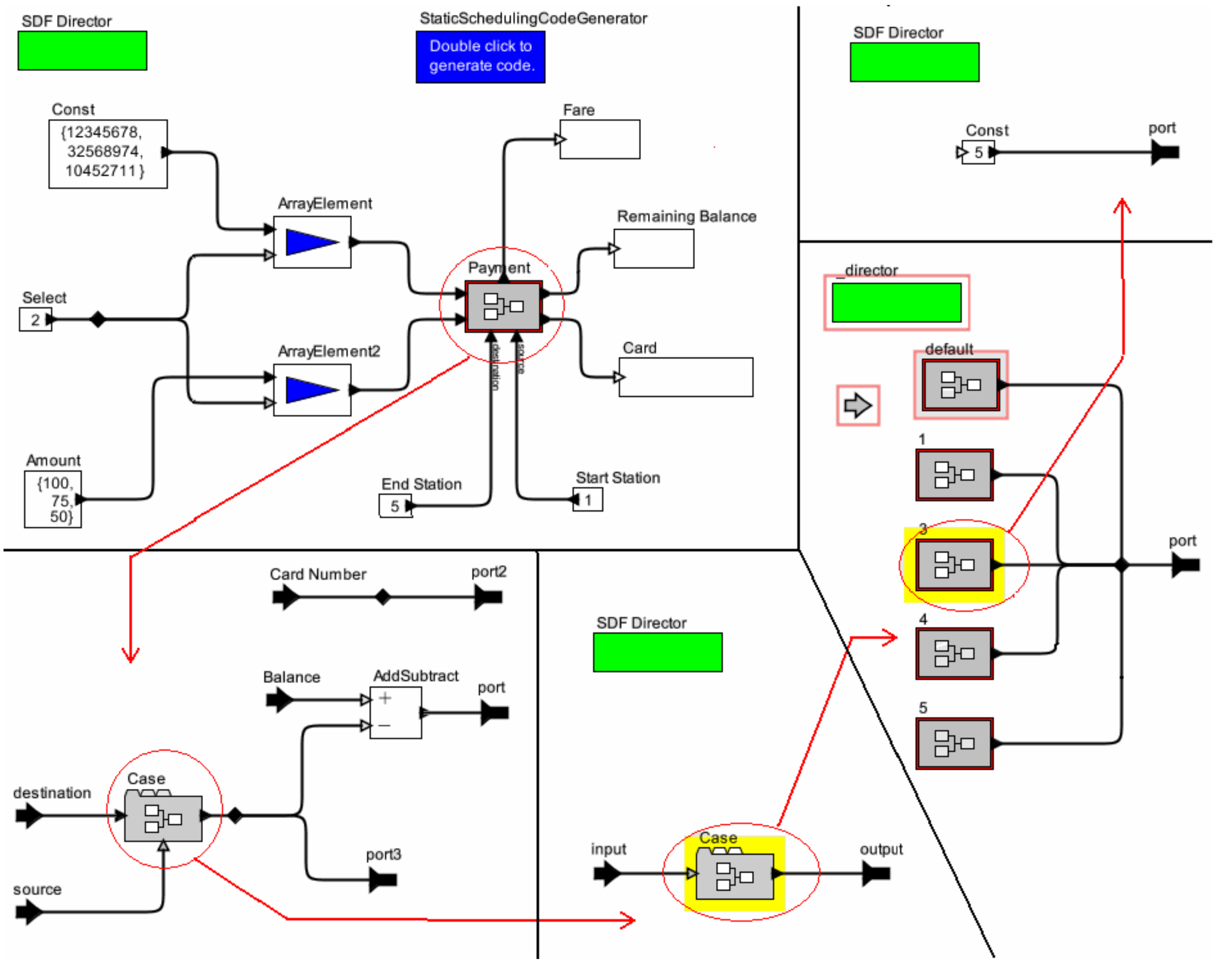


Fig 6.6 Complete Fare Charging Module Design

Fig. 6.6 shows the complete design of ticketing and payment module. The same is explained in detail in section 5.4. Below given Fig. 6.7 shows testing of the compete module. The station selected here are, source station is 5 and destination station is 1. Also the card selected is 2nd from the array. The card details are number – 32568974 and balance – 75. From table 5.2 the charge can be seen as 20, so that all results are shown as Fare – 20, Remaining Balance – 55 and the Card Number – 32568974.

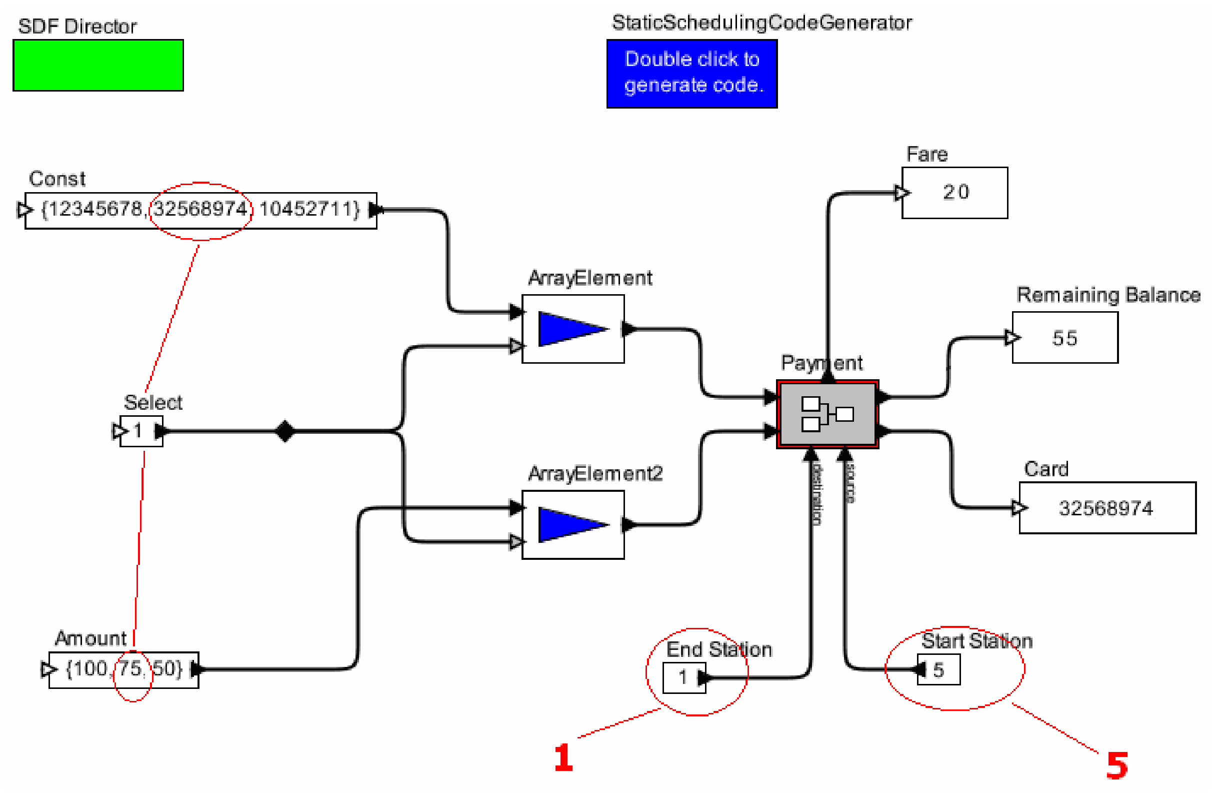
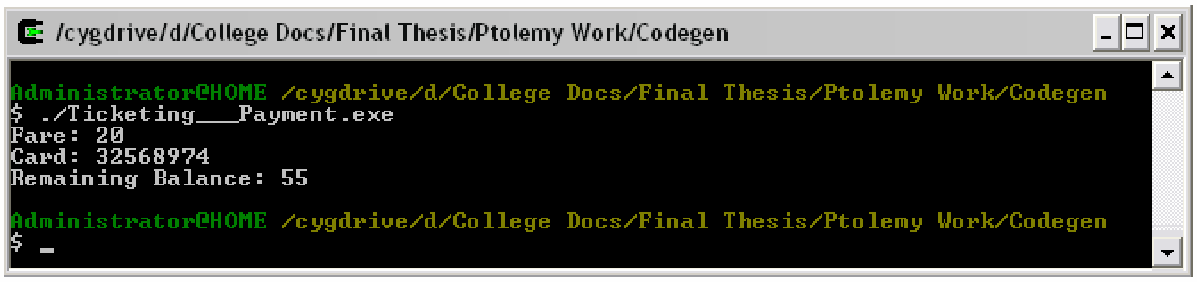


Fig 6.7 Testing of Fare Charging Module

The same result is achieved with "C" code execution under Cygwin [8]. It shows all the outputs those results when source and destination is 5 and 1 respectively.

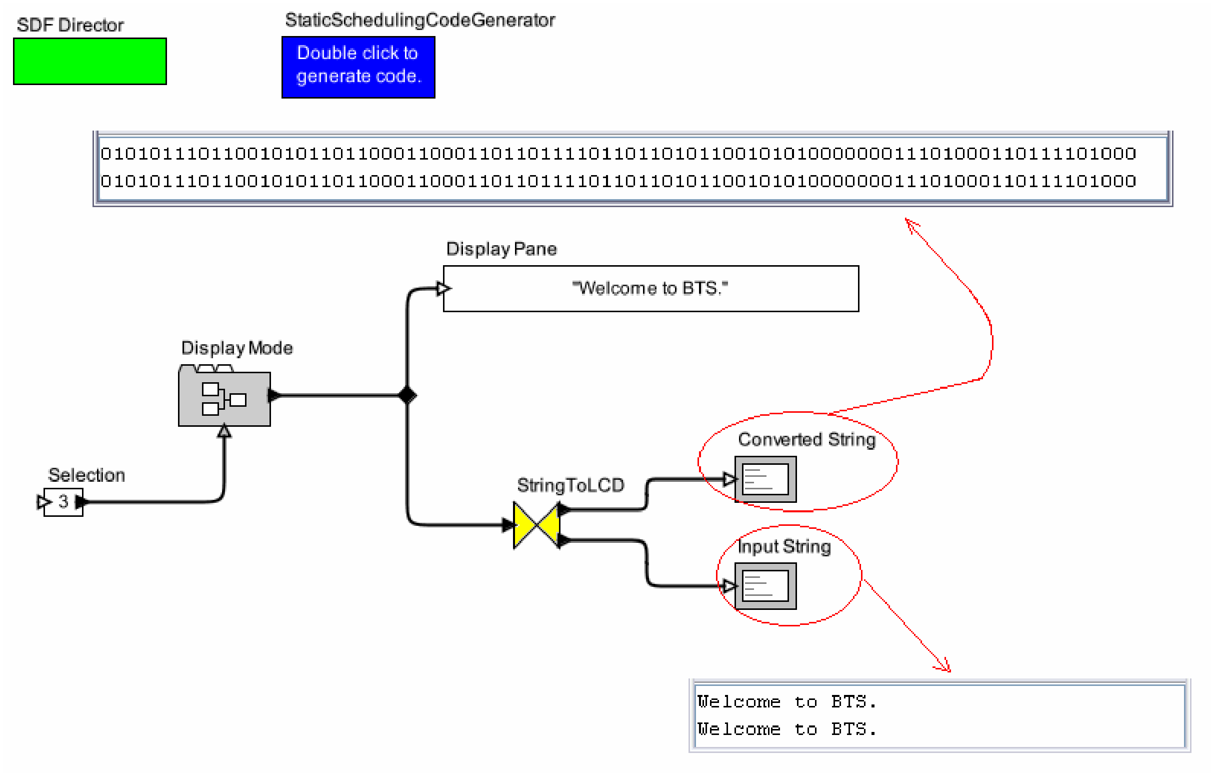


```
/cygdrive/d/College Docs/Final Thesis/Ptolemy Work/Codegen

Administrator@HOME /cygdrive/d/College Docs/Final Thesis/Ptolemy Work/Codegen
$ ./Ticketing___Payment.exe
Fare: 20
Card: 32568974
Remaining Balance: 55

Administrator@HOME /cygdrive/d/College Docs/Final Thesis/Ptolemy Work/Codegen
$ _
```

Fig 6.8 Testing "C" code of Fare Charging

6.2 INTEGRATION

This section covers details on system's integration. Two examples are taken which shows the results with two of modules integrated. And then how the full system is integrated, is shown.



```
01010111011001010110110001100011011011110110110101100101010000000111010001101111101000
01010111011001010110110001100011011011110110110101100101010000000111010001101111101000
```

```
Welcome to BTS.
Welcome to BTS.
```

Fig 6.9 Display with LCD display

Fig. 6.9 shows Display module integrated with the LCD display actor generated. Here output of Display module is given as input to the LCD actor. LCD actor gives result in two strings; one shows the input string and the other shows the converted string. Each character here is converted to 8 bit value and that way a full string is generated. Example shows how the modules can be integrated to result into full system. So now from the example it is obvious how the LCD actor can be used in conjunction with any display. Actually the converted string result is more of interest, but the original is also used which verifies results.

Next fig. 6.10 shows, timing information actor used with 8 bit LCD display actor. Here the output of each date and time is give as input to the actor that converts a string to 8 individual bits per character. The intermediate bit values are not shown here. 8 bits of output string is given as input to LCD8ToString actor which generates character out of each combination of bits given as inputs. Again here also the input string is shown as one of the output of StringToLCD8 actor, so as to verify the input and the result.

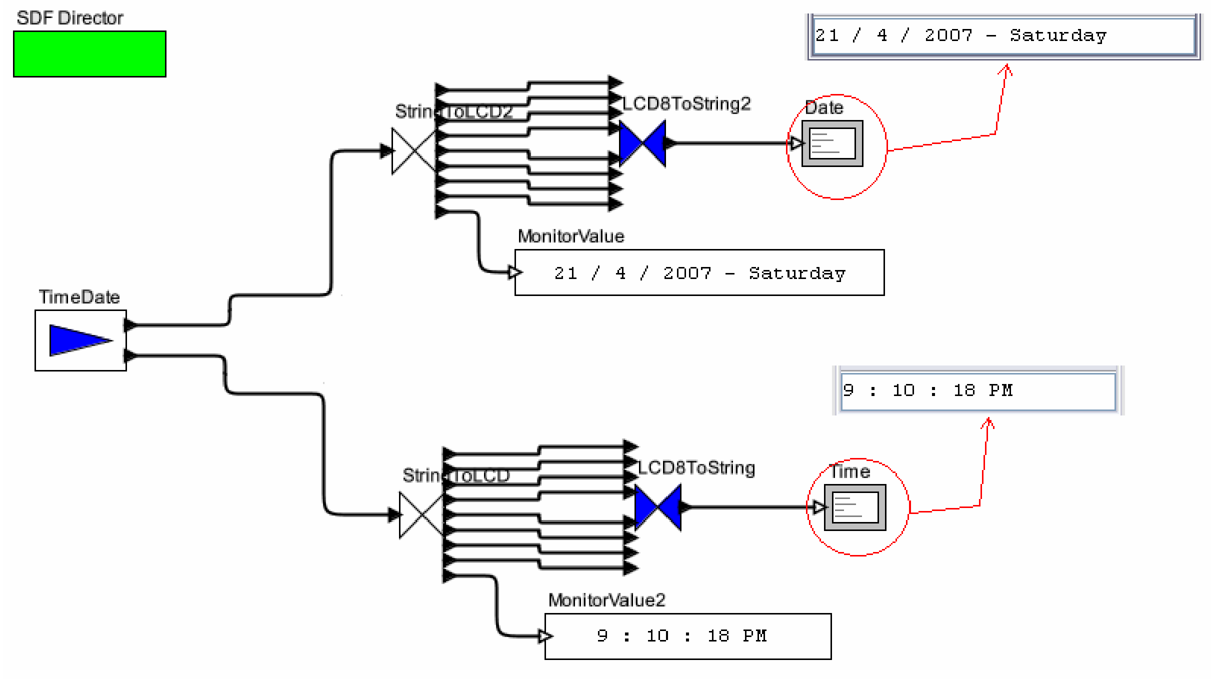


Fig 6.10 Timing and LCD 8 Bit Integrated

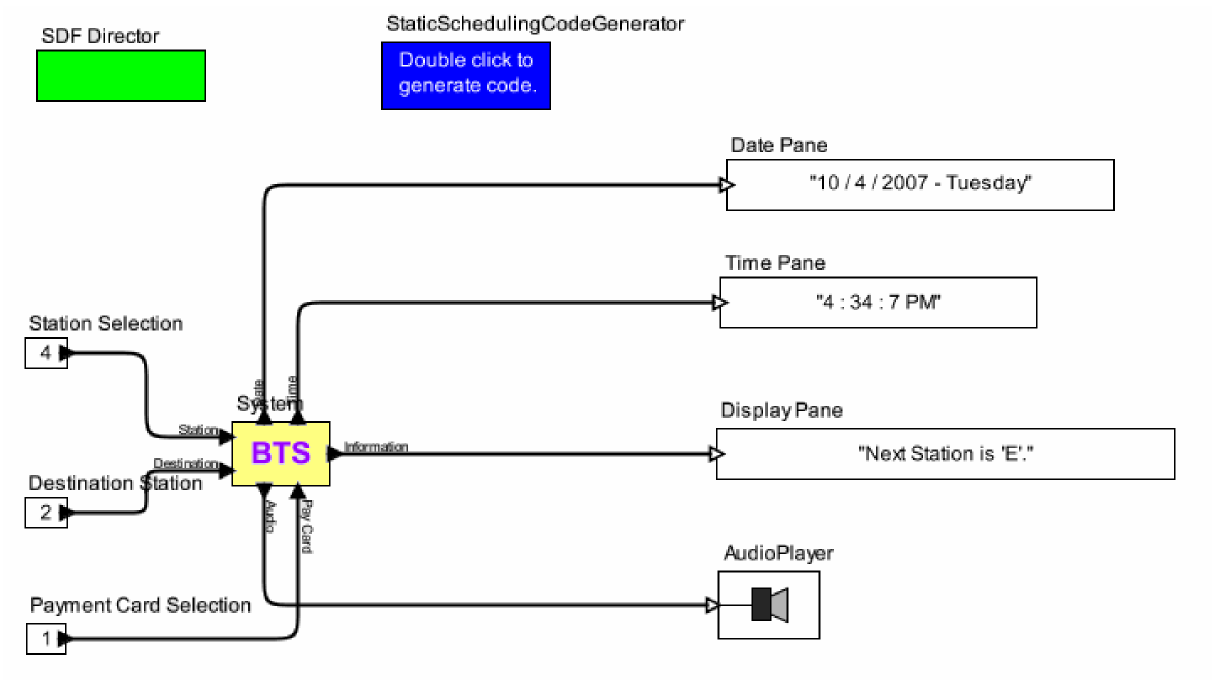


Fig 6.11 Full BTS Design

Fig. 6.10 shows complete model of BTS. Here the figure shows continuous outputs. That is those output observed in fig. 6.10 will be active during complete execution. The audio player will play audio announcements according to the station selected by driver's control. The central part of the fig. 6.10 is the system. Inside that all the modules developed are integrated. Fig. 6.11 shows inside of BTS, which has hierarchy.

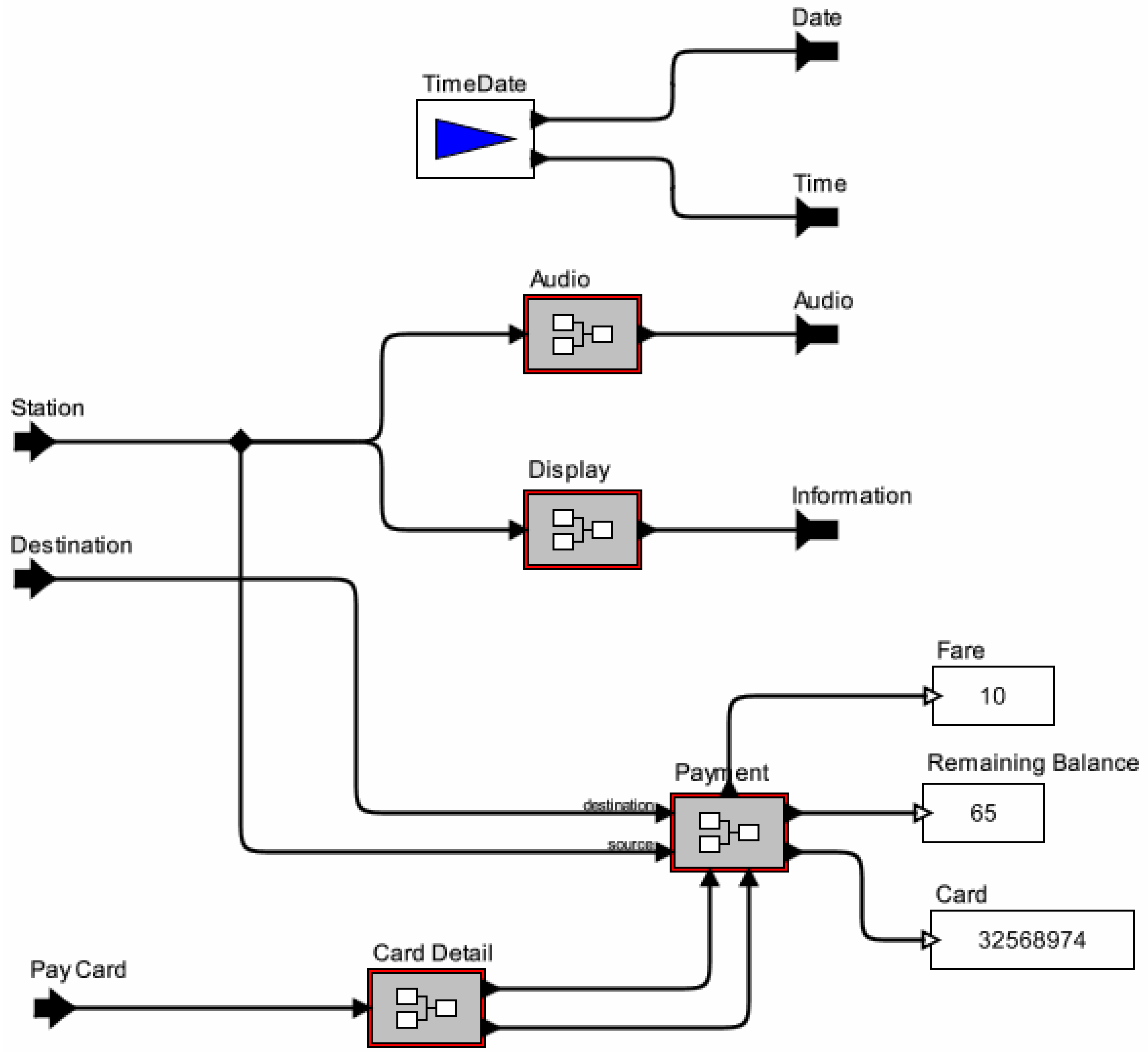


Fig 6.12 Inside of BTS

As seen in fig. 6.10 the system has 3 inputs. Out of which one will be active for complete execution. The station selection input is active all the time. Other two inputs are for destination station selection and card selection. These two actors will be active only if the payment function is required. So for that the destination is selected and also card is chosen by passenger, from the device.

As in fig. 6.10 there are four outputs that will be active all the time of execution. Audio alert will be always on, as one of the stations will be always selected. So that will play any audio message, according to the value selected. Same way display notification will work always and inform passengers about various details. Time and date are static information which will be active all the time. For this information to be displayed no input is required. From fig. 6.11 it can be observed that Timing information actor is disconnected from any other actor present in the hierarchy. That's why this information is static. It keeps on changing at each unit of time, during system's execution.

6.3 SYSTEM OPERATION

So for the system's execution only station has to be selected. This particular selection in actual system will be done with a signal that is generated when driver opens or closes the door of the Bus. That signal will cancel a station on the list every time it is generated. So now depending on the count of the stations remaining, the current station can be decided. So once the current station gets decided, audio module and display module will receive input. So now the continuous outputs are generated.

In the case of ticketing also the selected station is needed. The selected station will be taken as the source station in case of the calculation from the table 5.2. For selecting the destination, passenger has to select it from the device present on system [Fig. 4.8]. So now the system has two stations, from which the fare can be calculated. For payment, the card has to be selected. Just for modeling only limited card detail is feed into the system. So one of the cards has to be selected for payment purpose. As now the system has two stations and card detail available, so charging can be done.

In here, when it is the case of ticketing at that time also the continuous outputs will be generated and can be observed. Along with the displaying route and timing information, now the system also displays fare that shall be charged. In this manner the system can be executed and results can be verified.

7. SUMMARY AND CONCLUSION

7.1 SUMMARY

Modeling an embedded system is key requirement prior to manufacturing it. There are plenty of modeling options and tools are available. First of all Ptolemy tool's study is necessary as the goal is to make a model of an embedded system. Here an example system BTS is used and modeled with Ptolemy. BTS stands for Bus Transport System and as name suggests it is made for better amenity of passengers and easier administration. The BTS is designed with functionalities like route information, passenger details, timing details, ticketing and payment. There are, as always several assumptions made for system's conceptualization, which are about passengers and system's operation under all situations. BTS is modeled under Ptolemy's Codegen Environment. For modeling, complete system is divided in 3 modules, Display Information Module, Audio Alert Module and Fare Charging Module. These modules are used in achieving system's various functionalities. During development, it was found that Ptolemy doesn't have any actor for having system's current time and date. Also there is no provision to get the display on the LCD device. So for these two functionalities, actors have been developed and patched to the environment and then they were used. With all these, Ptolemy's available actors and generated actors complete model has been generated. Finally all three modules were integrated and then tested for desired outcome.

For simulation, the system's code is required. To get the system's code, Ptolemy has provision for code generation. Using code generation, “C” code can be generated. To get “C” code, the libraries are also required and also there should be no mismatch. Once all this done, “C” code can be generated. With generated “C” code simulation can be done. For that the “C” code is cross compiled for the target processor family, that code can be ported to the target processor and can be verified. Because of mismatch in standard C library and SDL library, the C code has not been generated for full model. SDL library is needed for multimedia functionalities with C environment.

7.2 CONCLUSION

The project carried out here is to learn Embedded System Modeling and Simulation. There are various available modeling options, but model based approach is used here. For modeling, here Ptolemy tool is used which supports GUI based modeling. System modeled here is BTS – Bus Transport System. The system does have limited functionalities as stated earlier.

Ptolemy has set of actors available which are used for GUI based modeling. For some functionality of BTS, some new actors were required and they have been developed and then used. Some of the actors were modified according to need and have been found better appropriate later. Actor adding and using is indeed a tricky part of modeling yet it has been carried out to achieve final model of the system.

System is modeled and tested for desired outcome. All defined functionalities were perfectly modeled; still they can be updated to function in better way. Still simulation is to be carried out. Simulation can only be carried out on target processor. Simulation process requires code of the system and it is yet to be generated.

After carrying out complete modeling with Ptolemy, it is experienced that with Ptolemy any embedded system can be modeled and tested for performance.

7.3 FUTURE WORK

The system is developed to have the functionalities listed earlier and everything is checked. Still it needs to be simulated on target platform. For getting the code for target, "C" code is required. Separate "C" code for all modules has been generated but they are still to be integrated. SDL library creates problem by mismatching with C libraries for unsigned integers. Once that problem gets resolved, rest things can be done. Once "C" code is there, the integration and target code generation can be done. Also controlling logic has to be added to the LCD module that has been generated for the displaying.

Further system can be extended to have advance passenger’s information. System can be extended for payment process. One of the option is to get payment done with all credit cards, for this the system needed to communicate with central server. So adding of some sort of communication will be required. The communication can be using wireless network or GPRS. One other option for making payment process more efficient and errorless, concept of RFID can be used. With RFID the charges will be deducted when a passenger gets of the bus, i.e. at the end of journey. Also the system can communicate with administrator to update about the traffic situation and that way it may help in system administration.

REFERENCES

[1] Christopher Brooks, Edward A. Lee, Xiaojun Liu, Steve Neuendorffer, Yang Zhao, Haiyang Zheng, *Ptolemy II – Heterogeneous Concurrent Modeling And Design In Java, Volume I – Introduction to Ptolemy II,* 2005

[2] Edward A. Lee, Stephen Neuendorffer, "Actor-Oriented Models For Codesign – Balancing Re-Use and Performance". Formal Methods and Models for System Design, Kluwer, 2004

[3] David Harel, "Statecharts : A visual formalism for complex systems". *Science of Computer Programming*, 8:231–274, 1987.

[4] Christopher Brooks, Edward A. Lee, Xiaojun Liu, Steve Neuendorffer, Yang Zhao, Haiyang Zheng, *Ptolemy II – Heterogeneous Concurrent Modeling And Design In Java, Volume 2 - Ptolemy II Software Architecture,* 2005

[5] Edward Lee, Stephen Neuendorffer, Michael J. Wirthlin, "Actor Oriented Design of Embedded Hardware and Software Systems". Journal of Circuits, Systems, and Computers, Vol. 12, No. 3 (2003) 231-260.

[6] Gang Zhou, Man-Kit Leung and Edward A. Lee, "A Code Generation Framework for Actor-Oriented Models with Partial Evaluation", EECS, Technical Report No. UCB/EECS-2007-29

[7] "Ptolemy Official Website", http://ptolemy.eecs.berkeley.edu/

[8] "Cygwin Source and Information", http://www.cygwin.com/

[9] "Information about SDL", http://www.libsdl.org/

[10] "Vergil – Tool from Ptolemy", http://ptolemy.eecs.berkeley.edu/ptII6.0/ptII6.0.2/doc/design/usingVergil/index.htm

APPENDIX A

In this section, code of the actors that are generated for the modeling is covered. These are the actors that are developed and patched with Ptolemy environment and then used for modeling. All the actors are explained in section 5.6. The coding is done in Java. Java standard libraries and libraries for Ptolemy are used here. Coding follows the coding style of the Ptolemy [1].

A.1 TIMING ACTOR

Functioning and operation of timing actor is covered in section 5.6.1. The code is as given below,

```
package ptolemy.actor.lib;

import java.*;
import java.util.*;
import ptolemy.actor.TypedIOPort;
import ptolemy.actor.TypedAtomicActor;                Standard
import ptolemy.data.DoubleToken;                       Library
import ptolemy.data.StringToken;                        Files
import ptolemy.data.Token;
import ptolemy.data.type.BaseType;
import ptolemy.kernel.CompositeEntity;
import ptolemy.kernel.util.IllegalActionException;
import ptolemy.kernel.util.NameDuplicationException;
import ptolemy.kernel.util.Workspace;

public class MyTime extends TypedAtomicActor {

   public MyTime(CompositeEntity container, String name)       // Constructor
         throws IllegalActionException, NameDuplicationException {
      super(container, name);

      dt = new TypedIOPort(this, "Date", false, true);
      dt.setTypeEquals(BaseType.STRING);                    IO Port of
                                                            the Actor
      tm = new TypedIOPort(this, "Time", false, true);
      tm.setTypeEquals(BaseType.STRING);
   }

   public TypedIOPort dt, tm;

   public void fire() throws IllegalActionException {        // Execution Starts Here

      super.fire();

      String res_tm = new String();
```

```
    String res_dt = new String();

    res_tm = _getCurrentTime();
    tm.send(0, new StringToken(res_tm));

    res_dt = _getCurrentDt();
    dt.send(0, new StringToken(res_dt));

  }

public void initialize() throws IllegalActionException {     // Private variable
                                                              Initialization
    super.initialize();
    _startTime = System.currentTimeMillis();
  }

 protected String _getCurrentTime() {    // Method to get the Current Time
    Date dt = new Date(_startTime);
    String temp = new String();
    int hrs, min, sec;

    hrs = dt.getHours();
    min = dt.getMinutes();
    sec = dt.getSeconds();

    if (hrs > 12) {
      hrs -= 12;
      temp = temp + "" + hrs + " : " + min + " : " + sec + " PM";
    }
    else
    temp = temp + "" + hrs + " : " + min + " : " + sec + " AM";

    return temp;
  }

 protected String _getCurrentDt() {      // Method to get the Current Date
    Date dt = new Date(_startTime);
    String temp = new String();
    String d = new String();
    int day, mon, yr, dat;

    dat = dt.getDate();
    day = dt.getDay();
    mon = dt.getMonth() + 1;
    yr = dt.getYear() + 1900;

    switch(day) {
    case 0 : d = "Sunday";           break;
    case 1 : d = "Monday";           break;
    case 2 : d = "Tuesday";          break;
    case 3 : d = "Wednesday";        break;
    case 4 : d = "Thursday";         break;
    case 5 : d = "Friday";           break;
    case 6 : d = "Saturday";         break;
    default : break;
    }
    temp = temp + "" + dat + " / " + mon + " / " + yr + " - " + d ;
```

```
        return temp;
    }

    private long _startTime;
}
```

A.2 LCD ACTORS

This section covers the core functions of LCD actors. As the coding style for each actor will remain same [1]. So from next onwards only code functions of LCD is shown. The codes are shown for LCDToString, StringToLCD, LCD8ToString and StringToLCD8. Functioning of all these actors is explained in section 5.6.1.

A.2.1 StringToLCD

// Libraries to Include

```
public class StringToLCD extends TypedAtomicActor {      // Constructor

    public StringToLCD(CompositeEntity container, String name)
            throws NameDuplicationException, IllegalActionException {
        super(container, name);

        input = new TypedIOPort(this, "input", true, false);          ⎫
        input.setTypeEquals(BaseType.STRING);                          ⎪   IO Port of
                                                                       ⎪   the Actor
        output = new TypedIOPort(this, "output", false, true);        ⎬
        output.setTypeEquals(BaseType.STRING);                         ⎪
                                                                       ⎪
        extra = new TypedIOPort(this, "extra", false, true);          ⎪
        extra.setTypeEquals(BaseType.STRING);                          ⎭

        _attachText("_iconDescription", "<svg>\n"          // Actor's Icon
                + "<polygon points=\"-15,-15 15,15 15,-15 -15,15\" "
                + "style=\"fill:white\"/>\n" + "</svg>\n");
    }

    public TypedIOPort input;

    public TypedIOPort output;

    public TypedIOPort extra;

    public void fire() throws IllegalActionException {
        super.fire();

        if (input.hasToken(0)) {
            StringToken inputToken = (StringToken) input.get(0);
            String ip = inputToken.stringValue();

            String op = _doConvert(ip); //(Creates a new copy)
            extra.send(0, new StringToken(ip));
        }
    }
    public boolean prefire() throws IllegalActionException {
        if (!input.hasToken(0)) {
            return false;
        }

        return super.prefire();
    }


    private String _doConvert(String in) {    // Function to convert String to LCD Code

    char a[] = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', ':', ';', '<', '=', '>', '?', ' ', 'A', 'B', 'C',
                 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V',
                'W', 'X', 'Y', 'Z', '[', '.', ']', '^', '_', '`', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k',
                'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '{', '\n', '}', ',', '!'};

     String str = new String();
     String res = new String();

         for(int i=0;i<in.length();i++)
         {
                int jc;
                int c=in.charAt(i);

                for(jc=0;(int)a[jc]!=c;jc++);
                jc+=48;

                 str="";

                for(int j=0;j<8;j++)
                {
                    if(jc/(int)Math.pow(2,7-j)==0)
                         str +="0";
                    else
                         str +="1";
                     jc=jc%(int)Math.pow(2,7-j);
                }

                res += str;
          }

        return res;
   }
}
```

A.2.2 LCDToString

// Libraries to Include

```
public class LCDToString extends TypedAtomicActor {
    public LCDToString(CompositeEntity container, String name)
            throws NameDuplicationException, IllegalActionException {
        super(container, name);
        input = new TypedIOPort(this, "input", true, false);
        input.setTypeEquals(BaseType.STRING);                            IO Port of
                                                                         the Actor
        output = new TypedIOPort(this, "output", false, true);
        output.setTypeEquals(BaseType.STRING);

        extra = new TypedIOPort(this, "extra", false, true);
        extra.setTypeEquals(BaseType.STRING);

        _attachText("_iconDescription", "<svg>\n"          // Actor's Icon
              + "<polygon points=\"-15,-15 15,15 15,-15 -15,15\" "
              + "style=\"fill:white\"/>\n" + "</svg>\n");
    }

    public TypedIOPort input;

    public TypedIOPort output;

    public TypedIOPort extra;

    public void fire() throws IllegalActionException {
        super.fire();

        if (input.hasToken(0)) {
           StringToken inputToken = (StringToken) input.get(0);
           String ip = inputToken.stringValue();

           String op = _doConvert(ip);

           output.send(0, new StringToken(op));

           extra.send(0, new StringToken(ip));
        }
    }

    public boolean prefire() throws IllegalActionException {
        if (!input.hasToken(0)) {
            return false;
        }

        return super.prefire();
    }

    private String _doConvert(String in) {    // Converts LCD Code to String

    char a[] = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', ':', ';', '<', '=', '>', '?', ' ', 'A', 'B', 'C',
                 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V',
                'W', 'X', 'Y', 'Z', '[', '.', ']', '^', '_', '`', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k',
                'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '{', '\n', '}', ',', '!'};

    String str = new String();

    for(int i=0; i<(in.length()/8); i++)
    {
            int c=0;
            for(int j=0;j<8;j++)
            {
                c += (in.charAt((8*i)+j)-48)*Math.pow(2,7-j);
            }
            str += a[c-48];
        }
        return str;
      }
    }
```

A.2.3 StringToLCD8

```
// Libraries to Include

public class StringToLCD8 extends TypedAtomicActor {   // Constructor

  public StringToLCD8(CompositeEntity container, String name)
        throws NameDuplicationException, IllegalActionException {
     super(container, name);

     input = new TypedIOPort(this, "input", true, false);
     input.setTypeEquals(BaseType.STRING);

     output0 = new TypedIOPort(this, "output0", false, true);
     output0.setTypeEquals(BaseType.STRING);

     output1 = new TypedIOPort(this, "out1", false, true);
     output1.setTypeEquals(BaseType.STRING);

     output2 = new TypedIOPort(this, "out2", false, true);
     output2.setTypeEquals(BaseType.STRING);

     output3 = new TypedIOPort(this, "out3", false, true);
     output3.setTypeEquals(BaseType.STRING);

     output4 = new TypedIOPort(this, "out4", false, true);
     output4.setTypeEquals(BaseType.STRING);

     output5 = new TypedIOPort(this, "out5", false, true);
     output5.setTypeEquals(BaseType.STRING);

     output6 = new TypedIOPort(this, "out6", false, true);
     output6.setTypeEquals(BaseType.STRING);

     output7 = new TypedIOPort(this, "out7", false, true);
     output7.setTypeEquals(BaseType.STRING);

     extra = new TypedIOPort(this, "extra", false, true);
     extra.setTypeEquals(BaseType.STRING);

     _attachText("_iconDescription", "<svg>\n"  // Actor's Icon
          + "<polygon points=\"-15,-15 15,15 15,-15 -15,15\" "
          + "style=\"fill:white\"/>\n" + "</svg>\n");
  }

  public TypedIOPort input;

  public TypedIOPort out0, out1, out2, out3, out4, out5, out6, out7;
```

IO Ports of the Actor

8 Output and 1 Input

```
public TypedIOPort extra;

public void fire() throws IllegalActionException {
   super.fire();

   if (input.hasToken(0)) {
      StringToken inputToken = (StringToken) input.get(0);
      String ip = inputToken.stringValue();

      String op = _doConvert(ip);

      for (int j = 0; j < (op.length()/8); j++) {
          output0.send(0, new StringToken(""+op.charAt(0+(8*j))));
          output1.send(0, new StringToken(""+op.charAt(1+(8*j))));
          output2.send(0, new StringToken(""+op.charAt(2+(8*j))));     Sending
          output3.send(0, new StringToken(""+op.charAt(3+(8*j))));     Output
          output4.send(0, new StringToken(""+op.charAt(4+(8*j))));     On each
          output5.send(0, new StringToken(""+op.charAt(5+(8*j))));     Output
          output6.send(0, new StringToken(""+op.charAt(6+(8*j))));     port
          output7.send(0, new StringToken(""+op.charAt(7+(8*j))));
      }

      extra.send(0, new StringToken(ip));
   }
}

public boolean prefire() throws IllegalActionException {
   if (!input.hasToken(0)) {
      return false;
   }

   return super.prefire();
}

private String _doConvert(String in) {     // Conversion Function

char a[] = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', ':', ';', '<', '=', '>', '?', ' ', 'A', 'B', 'C',
             'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V',
            'W', 'X', 'Y', 'Z', '[', '.', ']', '^', '_', '`', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k',
            'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '{', '\n', '}', ',', '!'};

 String str = new String();
 String res = new String();

    for(int i=0;i<in.length();i++)
    {
         int jc;
         int c=in.charAt(i);

         for(jc=0;(int)a[jc]!=c;jc++);
         jc+=48;

          str="";

         for(int j=0;j<8;j++)
         {
            if(jc/(int)Math.pow(2,7-j)==0)
```

```
                str +="0";
            else
                str +="1";
             jc=jc%(int)Math.pow(2,7-j);
          }

          res += str;
       }

     return res;
  }
}
```

A.2.4 LCD8ToString

```
// Libraries to Include

public class LCD8ToString extends TypedAtomicActor {

   public LCD8ToString(CompositeEntity container, String name)
         throws NameDuplicationException, IllegalActionException {
      super(container, name);

      input0 = new TypedIOPort(this, "input0", true, false);
      input0.setTypeEquals(BaseType.STRING);

      input1 = new TypedIOPort(this, "input1", true, false);
      input1.setTypeEquals(BaseType.STRING);

      input2 = new TypedIOPort(this, "input2", true, false);
      input2.setTypeEquals(BaseType.STRING);

      input3 = new TypedIOPort(this, "input3", true, false);
      input3.setTypeEquals(BaseType.STRING);

      input4 = new TypedIOPort(this, "input4", true, false);
      input4.setTypeEquals(BaseType.STRING);

      input5 = new TypedIOPort(this, "input5", true, false);
      input5.setTypeEquals(BaseType.STRING);

      input6 = new TypedIOPort(this, "input6", true, false);
      input6.setTypeEquals(BaseType.STRING);

      input7 = new TypedIOPort(this, "input7", true, false);
      input7.setTypeEquals(BaseType.STRING);

      output = new TypedIOPort(this, "output", false, true);
      output.setTypeEquals(BaseType.STRING);

      _attachText("_iconDescription", "<svg>\n"  // Actor's Icon
            + "<polygon points=\"-15,-15 15,15 15,-15 -15,15\" "
            + "style=\"fill:blue\"/>\n" + "</svg>\n");
   }

   public TypedIOPort input0, input1, input2, input3, input4, input5, input6, input7;
```

IO Ports of the Actor

8 Input and 1 Output

```
    public TypedIOPort output;

    public void fire() throws IllegalActionException {
       super.fire();

       if ((input0.hasToken(0)) & (input1.hasToken(0)) & (input2.hasToken(0)) &
       (input3.hasToken(0)) & (input4.hasToken(0)) & (input5.hasToken(0)) &
       (input6.hasToken(0)) & (input7.hasToken(0))) {
         StringToken inputToken0 = (StringToken) input0.get(0);
         StringToken inputToken1 = (StringToken) input1.get(0);
         StringToken inputToken2 = (StringToken) input2.get(0);        Reading All
         StringToken inputToken3 = (StringToken) input3.get(0);        8 inputs
         StringToken inputToken4 = (StringToken) input4.get(0);        From 8
         StringToken inputToken5 = (StringToken) input5.get(0);        input ports
         StringToken inputToken6 = (StringToken) input6.get(0);
         StringToken inputToken7 = (StringToken) input7.get(0);

         String ip = inputToken0.stringValue() + inputToken1.stringValue() +
                     inputToken2.stringValue() + inputToken3.stringValue() +
                     inputToken4.stringValue() + inputToken5.stringValue() +
                     inputToken6.stringValue() + inputToken7.stringValue();

         String op = _doConvert(ip);

         output.send(0, new StringToken(op));
       }
    }

    private String _doConvert(String in) {   // Conversion Function

     char a[] = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', ':', ';', '<', '=', '>', '?', ' ', 'A', 'B', 'C',
                 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V',
                'W', 'X', 'Y', 'Z', '[', '.', ']', '^', '_', '`', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k',
                'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '{', '\n', '}', ',', '!'};

    String str = new String();

    for(int i=0; i<(in.length()/8); i++)
        {
          int c=0;
          for(int j=0;j<8;j++)
          {
                c += (in.charAt((8*i)+j)-48)*Math.pow(2,7-j);
          }
          str += a[c-48];
        }
        return str;
    }
}
```

APPENDIX B

This section covers various details of the project, which are very much needed as a backend or as a needed one. It covers details on Ptolemy Code Generation Process, about how to add and use actors with Ptolemy and finally Cygwin details. This information is covered in very concise manner.

B.1 CODE GENERATION PROCESS

In Ptolemy II, there are no separate code generations domains. Once a model has been designed, simulated and verified to satisfy the given specification in the simulation domain, code can be directly generated from the model. Each helper doesn't have its own interface. Instead, it interrogates the associated actor to find its interface with ports parameters during the code generation. Thus the interface consistency is maintained naturally. The generated code, when executed, should present the same behavior as the original model.

In the Ptolemy's code of any actor, the preinitialize() method is assumed to be invoked exactly once during the lifetime of an execution of a model and before the type resolution. The initialize() method is assumed to be invoked once after the type resolution. The prefire(), fire(), and postfire() methods will usually be invoked many times, with each sequence of method invocations defined as one iteration. The wrapup() method will be invoked exactly once per execution at the end of the execution.

The classes to support code generation are located in the subpackages under ptolemy.codegen. The counterpart of the Executable interface is the ActorCodeGenerator interface. This interface defines the methods for generating target code in different stages corresponding to what happens in the simulation. These methods include generatePreinitializeCode(), generateInitializeCode(), generateFireCode(), generateWrapupCode(), etc.

CodeGeneratorHelper, the counterpart of AtomicActor, is the base class implementing the ActorCodeGenerator interface and provides common functions

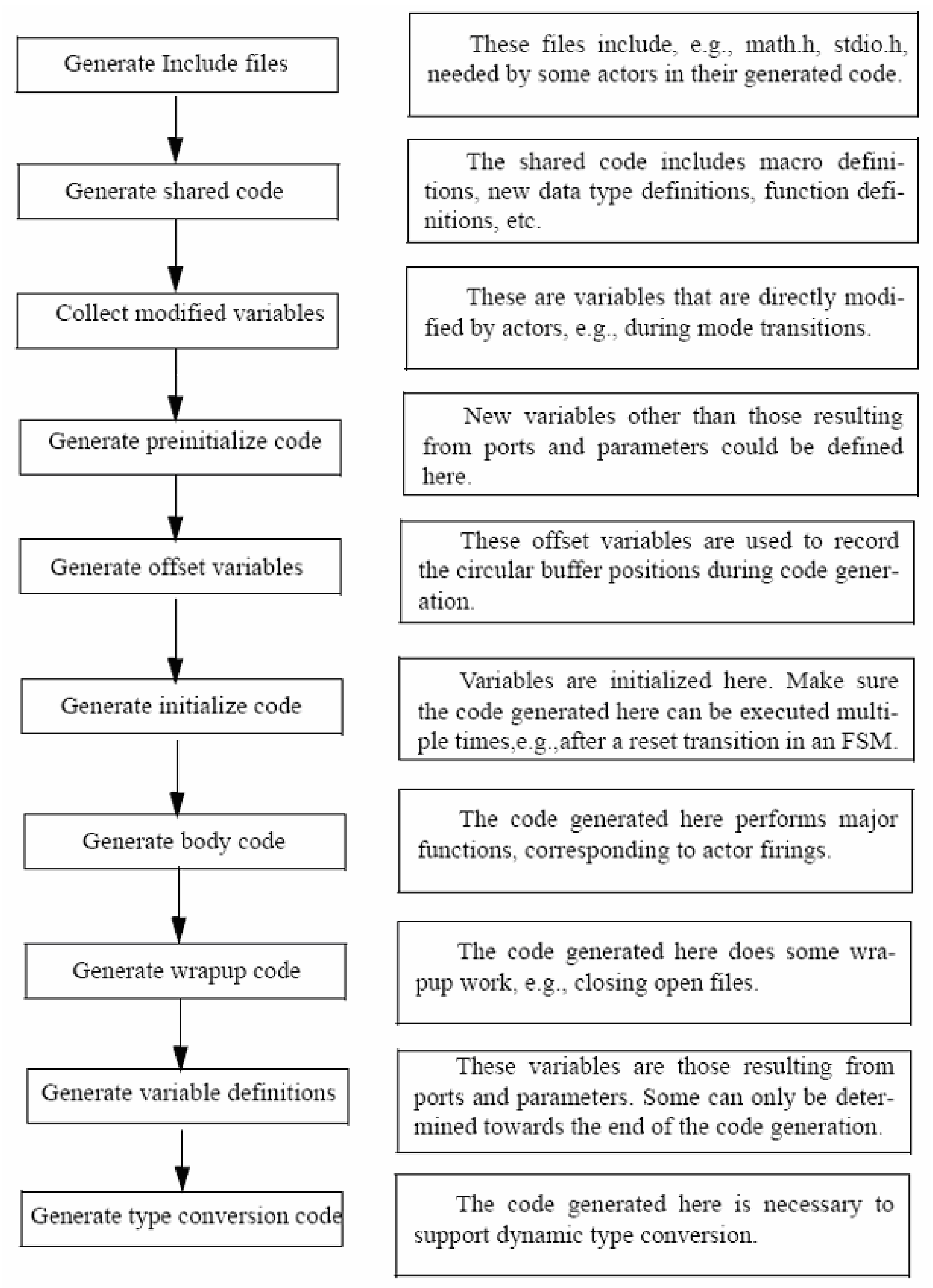

Generate Include files
These files include, e.g., math.h, stdio.h, needed by some actors in their generated code.
Generate shared code
The shared code includes macro definitions, new data type definitions, function definitions, etc.
Collect modified variables
These are variables that are directly modified by actors, e.g., during mode transitions.
Generate preinitialize code
New variables other than those resulting from ports and parameters could be defined here.
Generate offset variables
These offset variables are used to record the circular buffer positions during code generation.
Generate initialize code
Variables are initialized here. Make sure the code generated here can be executed multiple times,e.g.,after a reset transition in an FSM.
Generate body code
The code generated here performs major functions, corresponding to actor firings.
Generate wrapup code
The code generated here does some wrapup work, e.g., closing open files.
Generate variable definitions
These variables are those resulting from ports and parameters. Some can only be determined towards the end of the code generation.
Generate type conversion code
The code generated here is necessary to support dynamic type conversion.

Fig B.1 Code Generation Process

for all actor helpers. Actors and their helpers have same names so that the Java reflection mechanism can be used to load the helper for the corresponding actor during the code generation process. Finally the StaticSchedulingCodeGenerator class is used to orchestrate the whole code generation process. An instance of this class is contained by the top level composite actor. The code generation starts at the top level composite actor and the code for the whole model is generated hierarchically, much similar to how a model is simulated in Ptolemy II environment.

The flow chart in figure B.1 [6] shows the whole code generation process step by step. The details of some steps are MoC-specific. Notice that the steps outlined in the figure do not necessarily follow the order the generated codes are assembled together. For example, only those parameters that change values during the execution need to be defined as variables. Therefore those definitions are generated last when all the code blocks have been processed, but placed at the beginning of the generated code.

B.2 ACTOR ADDING TO PTOLEMY

There is no direct way to add an actor to Ptolemy environment. There is a set of steps that has to be followed to add an actor. These steps are to be done with accuracy. So the process to add actor is some what difficult. The steps are as shown below,

1. Create the new .java file that implements the actor.
   E.g. MyTime.java
2. Edit the $PTII/ptolemy/actor/lib/makefile and add MyTime.java to the value of JSRCS.
3. Run make in the $PTII/ptolemy/actor/lib directory. make will descend into the subdirectories and compile any needed files and eventually run

```
rm -f `basename MyTime.java .java`.class
CLASSPATH="../../.." "/cygdrive/c/j2sdk1.4.2_06/bin/javac" -g -O
MyTime.java
```

   After this gets done without any error, then class file for the code will be ready. To use this file under Ptolemy, it needs to be added to the

environment. To add it into the environment, it's added to any present xml files or a new file is to be generated.
E.g. $PTII/ptolemy/actor/lib/sources.xml

4. Edit $PTII/ptolemy/actor/lib/sources.xml and add MyTime to existing.

   To notice the change, PtolemyPreferences.xml is requiring to be reloaded. To reload this file, vergil has to be called from the Cygwin environment.

5. Start up Vergil.
   ```
   bash-2.04$ vergil
   ```
6. Now that actor can be found under Sources category of Actors in the left pane of the Vergil environment.

This way an actor can be added to the Ptolemy environment. It is always required to recompile any actor if any changes are done that. To do the recompilation step 3 is necessary to be carried out. For any new actor all steps are necessary, though step 2 can be omitted which is just required to make sure that newly generated code is patched and stored to some .jar file for backup.

B.3 CYGWIN DETAILS

Here the details about Ptolemy's configuration with Cygwin and its use are covered. Firstly Cygwin source and installation details can be found at http://sources.redhat.com/cygwin. So referring that Cygwin can be downloaded and installed to the system. It comes with bash shell to do the operations. To use Ptolemy with Cygwin's support, it is required to build the Ptolemy II under Cygwin first. The steps to do that are,

1. Install Java. (Check Compatibility with Cygwin's current version)
2. Install Cygwin toolkit. (Developer's Source)
3. Set the PTII environment variable to the top level Ptolemy directory, that is the directory above this directory,
   ```
   export PTII=c:/Ptolemy/ptII6.0.2
   ```
4. Run configure

```
cd "$PTII"
rm -f config.*
./configure
```

5. The safest thing to do is to run make fast install on the entire tree:

```
make fast install
```

6. Start vergil

```
$PTII/bin/vergil
```

All 1-6 steps are required for patching / building Ptolemy with Cygwin. Once patching is done vergil can be executed from Cygwin with $PTII/bin/vergil. This will execute vergil.bat file. Under this batch file all the latest PtolemyPreferences are loaded everytime it gets executed.

Cygwin is required when an actor is to be added to Ptolemy. Other options to add actor to Ptolemy is with Eclipse. For actor adding, first building Ptolemy under Cygwin is required and then actor adding steps can be carried out.