Simulators are useful to investigate the behaviors of new phenomena and systems before designing and implementing prototypes of such systems. This thesis presents the design, algorithms, and implementation of a new simulation platform for a novel material handling systems called the *Cloud Conveyor System* (CCS).

CCS is a new approach to conveying entities, i.e., materials and people, that comprises a collection of mobile units that periodically move along fixed tracks. Inputs and outputs are located at the ends of these fixed tracks; entities arrive via the inputs and transfer through the system to some output by riding on the mobile units. When two units meet, an entity may transfer from one unit to another as long as the receiving unit has adequate capacity. This simple model of conveying entities offers a rich spectrum of spatio-temporal behaviors that have interesting connections to core issues in scheduling, resource allocation, communication, embedded systems, automation, and programming. The complexity of CCS arises from the interactions between the mobile units, hence, it is difficult to construct a system-level model that can be analyzed, even though the behavior of individual units is simple.
The simulator presented in this thesis enables a systematic investigation of cyber-physical issues in CCS. Since all the details of CCS are not yet fully understood, an extensible simulator was designed using the Model-View-Controller (MVC) architecture. In this approach, there is a clear separation between the system-model, its presentation to users, and user interaction with the model.

To cope with the complexity of the CCS, an object-oriented design approach was used. Consequently, the system-model comprises a collection of classes that directly map to a natural description of the CCS. The relationships between different object instances were carefully analyzed and separated into multiple classes. Object interaction diagrams were constructed to understand different usage scenarios. In addition to serving as a design rubric, the MVC framework was found to be valuable for testing the CCS Simulator and for profiling the performance of this simulator. This comprehensive platform has been designed to enable future research into several aspects of Cloud Conveyor Systems.
ACKNOWLEDGEMENTS

This thesis would not be possible without the support of my friends and family. I thank my mom, sister, niece and nephew for understanding when I couldn’t play because I had to work. My grandmother was pivotal as she takes such great care of all of us. My lab partners Kranthi, Mukesh, Arijit, PJ, Hemanth and Abdo were great support in working with me to understand concepts or to help me prepare research material.

I would also like to mention Matt Petras, my work supervisor at The University of Akron, for being understanding and supportive when I needed to adjust my schedule or take time off to complete class and research assignments. I would like to thank my committee members: Dr. Branicky, Dr. Duan, and Dr. Bao as well as Dr. Menzemer, for their support. Finally, this thesis and all the work leading up to it would not be possible without the amazing support and guidance of my advisor, Dr. Shiva Sastry.
TABLE OF CONTENTS

LIST OF TABLES . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . viii
LIST OF FIGURES . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . ix

CHAPTER

I. INTRODUCTION . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
   1.1 Importance of the Cloud Conveyor System . . . . . . . . . . . . 4
   1.2 The Simulation Platform . . . . . . . . . . . . . . . . . . . . . . 5
   1.3 Organization of this Thesis . . . . . . . . . . . . . . . . . . . . . 6

II. BACKGROUND . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
   2.1 Cloud Conveyor System Model . . . . . . . . . . . . . . . . . . . 7
   2.2 Real-time Scheduling View . . . . . . . . . . . . . . . . . . . . . 11
   2.3 Optimization, Planning and Pedagogy View . . . . . . . . . . . . 12
   2.4 Role of the CCS Simulator in Future Research . . . . . . . . . . 13

III. CLOUD CONVEYOR SYSTEM SIMULATOR DESIGN . . . . . . . 15
   3.1 Object-Oriented Approach . . . . . . . . . . . . . . . . . . . . . 16
   3.2 Model-View-Controller (MVC) Framework . . . . . . . . . . . . 21
   3.3 Class and Object Relationships . . . . . . . . . . . . . . . . . . . 25
   3.4 Algorithms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 28
3.5 Threads and Thread Safety ........................................... 29

IV. IMPLEMENTATION, PERFORMANCE ANALYSIS AND SYSTEM PROFILING ................................................. 31

4.1 Implementation ...................................................... 31

4.2 Using the Simulator .................................................. 32

4.3 Class Relationships .................................................. 36

4.4 Multi-thread Design .................................................. 39

4.5 Method Invocations .................................................. 45

4.6 Memory Usage ....................................................... 48

4.7 Code Coverage ....................................................... 49

4.8 Discussion ........................................................... 50

V. CONCLUSIONS ......................................................... 52

BIBLIOGRAPHY ........................................................ 54
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>View overhead. This table shows the overhead for the major application threads when displaying the graphic view.</td>
</tr>
</tbody>
</table>

viii
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Cloud Conveyors System. Conveying units move periodically between the two ends of fixed tracks, one grid unit for each time tick. Entities that arrive at an Input <em>physically move</em> toward their Output by riding on the units; a <em>cyber transfer</em> can occur when two units rendezvous.</td>
</tr>
<tr>
<td>3.1</td>
<td>Attributes of the <strong>CCS_Unit class</strong>. These attributes are all tagged <em>private</em>. This means that only the methods of this class can change values of these attributes.</td>
</tr>
<tr>
<td>3.2</td>
<td>Behaviors of the <strong>CCS_Unit Class</strong>. Methods of the class provide an interface for interaction. Behavior of an object is the expression of these interactions.</td>
</tr>
<tr>
<td>3.3</td>
<td>Architecture of the <strong>CCS Simulator</strong>. There are three dominant relationships among classes illustrated here. The classes grouped by architectural concern, model classes are on the top, controller classes in the middle, and view classes are on the bottom. The view classes must go through controller classes to get to model data.</td>
</tr>
<tr>
<td>3.4</td>
<td><em>Main object interaction</em>. <em>mController</em> maintains system time and coordinates all activities. As time advances, the units move on tracks. These tracks intersect and when multiple units meet at these intersections, a rendezvous occurs. It is at these rendezvous that cyber-transfers can happen, dictating conveying behavior of the CCS</td>
</tr>
<tr>
<td>4.1</td>
<td>Starting the simulator. <strong>CCS Simulator start screen</strong> provides the input panel for entering CCS configuration.</td>
</tr>
<tr>
<td>4.2</td>
<td>Unit configuration. <strong>CCS Simulator unit configuration window</strong> allows users to specify which tracks have units enabled and what the offset of the unit is.</td>
</tr>
</tbody>
</table>
4.3 *IO configuration.* The input/output configuration screen allows users to specify the number and location of inputs and outputs. The rate at which new entities arrive at the input are also configurable here. ... 35

4.4 *The Debug View.* This view displays a textual simulation of the configuration for debugging simulator behavior. ... 36

4.5 *The Track View.* This view shows simulator behavior through Rendezvous highlighting and Patterns of Rendezvous Events. ... 37

4.6 *Verifying Has relationship.* Allocation Stack Trace shows that the CCS.vController object instantiated the CCS.mController object in column 1. Column 4 (Live Objects) shows that there is only one CCS.mController object as intended in the design. Column 5 (Allocations) confirms that only one such object was ever instantiated in the execution. ... 38

4.7 *Verifying Uses relationship.* A filtered view of the Heap Dump shows that CCS.LinearTrackView uses CCS.mController as intended in the design. ... 39

4.8 *Thread performance.* Thread execution time with (top) and without (bottom) a view. Notice from the two figures that as the number of units increase, the execution time of the mController increases as expected. Notice from the top figure that the Linear Track View has the largest execution time when the view is used. This confirms that the intended separation of concerns have been achieved in the thread-safe design. Error bars indicate the standard deviation in the sample. ... 41

4.9 *Memory Usage with (top) and without (bottom) threads.* The figure on top shows that the number of bytes on the Heap for the two implementations are similar. The figure on the bottom shows the number of object instances on the heap. In both cases, notice that the overhead of using threads is not excessive. The small increase shown is expected. ... 43

4.10 *Benefits of multi-thread design.* The figure on top shows that the execution time of the mController is significantly less when using a multi-threaded design. Similarly, the figure on the bottom shows that the execution time of the vController is also improved in a multi-threaded design. Error bars indicate the standard deviation in the sample. ... 44
4.11 *Multi-thread verification.* Profiler view showing multiple instances of CCS_LinearTrackView and one instance of CCS_DebugView . . . . . 45

4.12 *Verification of method invocations.* Verification of invocations of the CCS_Input::injectPart method for a simulator configuration where $M = 7$ and the simulation ran for 100 time units . . . . . . . . . 46

4.13 *Source code of the CCS_Input::injectPart method.* The source code shows that for every invocation of CCS_Input::injectPart, and instance of CCS_Entity is created . . . . . . . . . . . . . . . . . . . . . . 47

4.14 *Verifying object allocation.* Instances of the CCS_Entity class show the expected number of object instances for the number of times CCS_Input::injectPart is called . . . . . . . . . . . . . . . . . . . . . . 47

4.15 *The Hot Spots view of the profiler shows code that executes often.* By examining hot spots in the profiler, the software was redesigned to eliminate an unintended sequence of calls to the same method . . . 48

4.16 *Code Coverage.* The figure is a view of the code coverage plugin and shows the lines of code that were executed (light gray) and the lines of code not executed (dark gray) during a test case created for the CCS_Model::removeUnit method . . . . . . . . . . . . . . . . . . . . . 51
CHAPTER I

INTRODUCTION

Many engineered systems today involve a tight integration between multiple sub-disciplines. These systems are pervasive in a variety of application domains such as advanced manufacturing, intelligent transportation, and healthcare. For example, a modern automation system requires the coordinated use of techniques from computer engineering/science, embedded systems, communications, control theory, electric machines, hydraulics, pneumatics, and others. The vision of the Cyber-Physical Systems (CPS) community is to foster concentrated research and education in such inter-disciplinary areas. Here, the focus is on understanding the principles that enable tight integration between the cyber activities such as computation, communication and control, and physical processes that constrain these activities.

Several challenges that need to be addressed to enable the CPS vision are presented in [22]. Among these, there is a need for systematic methods to model, analyze and exploit patterns of events that evolve in space and time to improve Quality of Service (QoS) of a CPS application. In this context, the Cloud Conveyors System (CCS) was proposed in [25] as a well-structured example of a CPS application that exhibits rich spatio-temporal behaviors that arise from the interaction of mobile conveying units.
CCS comprises a collection of conveying units that each move periodically between two ends of fixed tracks at a constant velocity. As illustrated in Figure 1.1, there are two sets of tracks — the horizontal and the vertical. The system-level objective here is to transport physical entities (e.g., material or people) from some Input port to some Output port, where each entity has its own target output port, deadline, and QoS constraints. As can be seen in Figure 1.1, entities enter the system via some Input that is located at the left end of a horizontal track, ride on one or more conveying units in CCS, and exit via an Output at the right end of a horizontal track.

Figure 1.1: Cloud Conveyors System. Conveying units move periodically between the two ends of fixed tracks, one grid unit for each time tick. Entities that arrive at an Input physically move toward their Output by riding on the units; a cyber transfer can occur when two units rendezvous.

Entities transfer through the CCS by riding on the units; when an entity moves from one grid location to another by riding on a mobile unit, we call it a
physical move. When two units rendezvous at a location, a cyber transfer can occur; here, an entity riding on one unit can instantaneously transfer to the other unit as long as the capacity constraint is not violated. A cyber transfer need not occur at every rendezvous because the decision on such a transfer would depend on many factors, including the target Output of the entity, deadline, and remaining time before which it must be delivered, QoS constraints of the entity, the current load on the downstream units in the system, etc. In order to account for these dynamic factors, every cyber transfer in CCS involves cyber activities such as decision-making, computation, communication, sensing, actuation, and control. Because each entity must be physically moved from one location (i.e., its Input) to another (its Output), we can say that every transport task involves an intertwined sequence of cyber transfers and physical moves. The system has to decide on the specific sequence of transfers for each entity so as to achieve the overall QoS objectives which may be quantified in terms of end-to-end latency, throughput of the entities, system availability, etc.

The interesting cases of transport in CCS involve entities that enter on horizontal track $H_j$ and exit on $H_{j'}$. In such a case, the entity must first ride along the horizontal track $H_j$, get transferred to a unit moving along some vertical track to reach the unit that moves along $H_{j'}$. Such a transport could, in general, also involve transfers over a sequence of units. Hence, in CCS a precise characterization of the pattern of rendezvous events that can occur is important for end-to-end transport of entities.
1.1 Importance of the Cloud Conveyor System

CCS enables a systematic investigation of issues that are central to several core issues in cyber-physical systems. This is a well-structured, precise problem in which computation, communication, control, sensing, and actuation interact tightly with the physics of the mobile units conveying entities. The arrival of entities via the Inputs is not deterministic; the mobile conveying units and low power communication links that are critical for the coordinated operation of these units can fail randomly. Thus, CCS presents a dynamic and uncertain operational environment. Composition in a physical realization of CCS is easily achieved by activating (or adding) more horizontal and vertical units. However, a small change in the configuration of CCS fundamentally affects the pattern of rendezvous events that can occur in the system and the period of this pattern. Consequently, the Input-Output connectivity for transport is changed and QoS of the system will be affected. Thus, while the idea of composition can be easily stated in CCS, this abstract problem demands rigorous and well-reasoned techniques to effectively leverage composition. Computational and physical notions of time and space are tightly integrated in the semantics of this intriguing and deceptively simple problem. The simulation platform described in this thesis enables a full exposition of the subtleties of the CCS.
1.2 The Simulation Platform

The primary objectives for the design of this platform were:

1. Reliable (correct) and robust model for CCS,
2. Incremental feature addition,
3. Easy maintenance, and
4. Efficient and detailed monitoring at multiple levels.

To address the above need, the CCS simulator was designed using an object-oriented approach. To enable future growth and easy maintenance, it was necessary to incorporate tools that could guide future changes and evolution of the simulator. For this reason, the simulator was designed using the Model-View-Controller (MVC) framework. This framework enabled better organization of the classes, provided deep insights into the class relationships, and simplified the design verification activities necessary to make the platform robust.

The specific contributions of this thesis are:

1. Object-Oriented Design of the Cloud Conveyor System in the Model-View-Controller Framework,
2. Implementation of the design and validation,
3. Detailed profiling of the implementation, and
4. Validation of the implemented simulator as a foundation for CCS research.
1.3 Organization of this Thesis

This thesis gives relevant background into the *Cloud Conveyors System* in Chapter 2. In Chapter 3 the initial design of the CCS Simulator is presented. In the following Chapter 4, the design is verified using tools of the proposed CCS Platform. This thesis presents a conclusion of the work in Chapter 5.
CHAPTER II

BACKGROUND

This chapter presents the system-model for the Cloud Conveyor System and related work. It also addresses future work in which the simulator presented in this thesis will be a part.

2.1 Cloud Conveyor System Model

The software simulator described in this thesis was designed to embody the description of the CCS as envisioned in [25]; the following description closely follows the original description and is reproduced here with permission.

The example instance of the CCS illustrated in Figure 1.1 has 10 vertical units and 6 horizontal units and the underlying grid has $8 \times 12$ lattice points that provide a discrete coordinate space for locations in CCS. Given, $M$ horizontal units and $N$ vertical units, and an underlying grid of $(M+2) \times (N+2)$, the horizontal tracks $H_1, H_2, \ldots, H_M$ are on rows $1, 2, \ldots, M$; similarly the vertical tracks $V_1, V_2, \ldots, V_N$ are on columns $1, 2, \ldots, N$. CCS may have a maximum of $M$ inputs and $M$ outputs, one at each end of every horizontal track $H_j$; the $j^{th}$ input, $1 \leq j \leq M$ is located at $(0, j)$ and the $j^{th}$ output is located at $(N + 1, j)$. 

7
2.1.1 Periodic Movement of Mobile Units

We represent the horizontal units as $h_1, h_2, \ldots, h_M$; unit $h_j$ moves along the track $H_j$. Similarly, the vertical units are $v_1, v_2, \ldots, v_N$ and unit $v_i$ moves along track $V_i$. System time evolves in discrete ticks and in each tick, every unit concurrently moves one unit along its track. Initially, the horizontal unit $h_j$ is at the grid location $(0, j)$ and the vertical unit $v_i$ is at location $(i, 0)$. The horizontal units, $h_j$, move from $(0, j)$ to $(N+1, j)$ and return toward $(0, j)$. Similarly, the vertical units $v_i$ move from $(i, 0)$ to $(i, M+1)$ and return to $(i, 0)$. All these units periodically move from one end of the track to the other at a constant velocity. Each mobile unit can be started with an initial delay (offset). We use $\delta^r_j$ to represent the offset of the horizontal unit $h_j$ and $\delta^v_i$ to represent the offset of the vertical unit $v_i$.

2.1.2 Rendezvous Events

A rendezvous event (or rendezvous) is a spatio-temporal event in CCS and is essential for transporting entities across two different horizontal tracks. At every rendezvous, a horizontal unit and a vertical unit meet at a particular location in the underlying grid at the same time. We refer to the portion of the grid that is bounded by the locations $(1, 1), (1, M), (N, M)$ and $(N, 1)$ as the Rendezvous Zone because any rendezvous in CCS can only occur in this part of the grid. By physically adjusting the actual positions of the units on the tracks, we can assure that the two units will not collide when a rendezvous event occurs; the units can be close enough so that an entity riding on one unit can be transferred to the other unit. Such transfers of entities
are necessary to improve the connectivity between inputs and outputs and the QoS that can be achieved in the CCS. Several rendezvous events can occur simultaneously at different locations of the underlying grid. Because the units move in a periodic manner, the locations and the times at which two units meet is also periodic and, hence, this is referred to as a pattern of rendezvous events.

2.1.3 Configuring CCS

The pattern of rendezvous is constrained by the CCS configuration.

**Definition 1.** A configuration of CCS is a specification of values for the number of horizontal units $M$, the number of vertical units $N$, the velocity of each unit, and the initial offset of each unit.

From a control perspective, most of the parameters of the configuration can be changed. The number of tracks cannot be changed dynamically without causing the system to reset. The velocities of the units can be changed (including setting the velocity of a specific unit to zero). While it is technically possible to introduce arbitrary delays (i.e., stalls) during the execution of units, it raises schedulability issues that are similar in spirit to clairvoyant scheduling. As long as the velocities of the units can be expressed as integers or rational values, the spatio-temporal pattern of rendezvous events is periodic; this periodic pattern determines the structure of paths in the system from the inputs to the outputs and, hence, affects the QoS that can be achieved.
2.1.4 Transporting Entities

The system-level objective in the abstract CCS is to transport entities from inputs to their outputs; the user must specify the QoS attributes, such as the maximum End-to-End Latency (Deadline) and Throughput that must be satisfied when delivering an entity to its output.

For example, suppose an entity $e_s$ arrives at input $I_j$ and must be transported to the output $O_j$. In this case, the transport of $e_s$ begins with a cyber transfer from $I_j$ to unit $h_j$, a physical move on $h_j$ from location $(0, j)$ to $(N + 1, j)$ and finally another cyber transfer from $h_j$ to $O_j$. Suppose another entity $e_x$ arrives at input $I_k$ and its target output is $O_j$, $k \neq j$. The transport of $e_x$ would begin with a cyber transfer from $I_k$ to unit $h_k$, a physical move on $h_k$ from $(0, k)$ to $(p, k)$, a cyber transfer from $h_k$ to unit $v_p$, a physical move on unit $v_p$ from $(p, k)$ to $(p, j)$ where another rendezvous must occur with unit $h_j$. A cyber transfer must occur at $(p, j)$ to transfer $e_x$ from $v_p$ to $h_j$. Finally, $e_x$ must physically move on $h_j$ to $O_j$ to complete the transport of $e_x$ with a cyber transfer to output $O_j$. This example illustrates that an end-to-end transport of an entity in the CCS always begins and ends with a cyber transfer. A physical move is always preceded by and succeeded by cyber transfer. When a unit reaches the end of the track and does not transfer to the output, it remains on the unit until it is involved in a cyber transfer.
2.2 Real-time Scheduling View

Using an analogy from real-time systems [9], the transport of an entity in the CCS can be viewed as a task \( \tau_i \); this task is released when a new entity, \( e_i \), arrives at its input. Tasks can have a relative deadline \( D_i \) before which \( e_i \) must be delivered to its output. A task is periodic when a sequence of entities that arrive at the same input with an inter-arrival time of \( T_i \). The \( j^{th} \) instance of task \( \tau_i \) is the \( j^{th} \) entity, \( e^j_i \), in such a periodic sequence. This task model can be extended in a natural manner to capture periodic, aperiodic, and sporadic arrival of entities at an input as used in the real-time systems literature [9, 17].

The horizontal and vertical units that are involved in the transport of entity \( e_i \) are the resources required to “execute” the task \( \tau_i \). Fixed priorities in CCS can be assigned to each input or each entity. Thus, if input \( I_i \) has priority \( \text{prio}(I_i) \), every entity that arrives via \( I_i \) will be assigned a fixed priority \( \text{prio}(I_i) \) for transport in the CCS. When each entity can be assigned its own priority, dynamic priority schemes can be used to efficiently transport entities in the CCS. In the latter approach, the priority of an entity can change when it is enroute; for example, the dynamic priority assigned to an entity can be based on the remaining time left before which its relative deadline would be violated.

Recent investigations related to scheduling have focused on utilizing load to handle critical and non-critical sporadic tasks [18], improving the feasibility of sporadic tasks [5], managing preemption overheads [4, 6], and resources [20, 15, 16].
Determining a sequence of physical and cyber transfers for transporting entities in CCS has many issues that are similar in spirit to real-time scheduling problems. In CCS, the scheduling is deeply connected to the spatio-temporal structure of the pattern of rendezvous events.

2.3 Optimization, Planning and Pedagogy View

Many problems related to the efficient transport of entities in the CCS and the optimal operation of the CCS can also be approached from the perspective of optimization and coordination [21] of multiple agents [24, 8]. A better understanding of these relationships can significantly improve cyber-physical systems education [14].

The CCS is intended to be a reference problem for cyber-physical systems similar to problems such as the Inverted Pendulum [2], Prisoner’s Dilemma [23], and Dining/Drinking Philosophers [10]. These problems have served as a basis for theoretical or experimental exploration of a variety of techniques in the domains of control, cooperation and conflict among agents, and distributed computing. In general, such problems help to formulate and analyze critical issues in a precise context that is relevant to several applications. Designers and theoreticians have worked within the context of such problems in different domains to gain useful insights. Similarly, the CCS is a precise, well-structured, context systematically addressing some of the critical issues in cyber-physical systems [25].
2.4 Role of the CCS Simulator in Future Research

The complex interplay between the structure of the CCS, i.e., the CCS configuration, the dynamic arrival of entities at the inputs, and the system-level objectives that must be satisfied, cannot be readily captured in mathematical models or analyzed in rigorous settings. For example, given a configuration of the CCS, it is necessary to understand what QoS can be achieved for the tasks. Second, given a collection of CCS tasks and the desired QoS for these tasks, it is necessary to determine what CCS configuration(s) can achieve the target QoS [25].

The CCS Simulator described in this thesis will enable researchers to empirically explore potential solutions to the above two problems. By observing the animations and analyzing event sequences in the CCS log files, such researchers can formulate hypotheses that relate the pattern of rendezvous events to QoS. The connections between the pattern of rendezvous events and real-time scheduling techniques can also be explored.

The CCS Simulator will also allow researchers to plan for prototype implementation of the CCS based on prior work reported in the Complex Engineered Systems Lab at the University of Akron [3]. Such a prototype system can fully expose the subtleties of the CCS that cannot be captured in the present simulator — however, the simulator is an important step toward designing and implementing such a prototype system.
The CCS Simulator will also enable researchers to explore *virtual network* embedding in the context of reconfigurable networks that transport physical entities. Similar ideas have been explored in the information realm recently [13]; however, adapting these ideas and the *Platform-as-a-Service* idea of cloud computing to the domain of material handling is a challenging and interesting problem [14]. Using the prior work in the area of Composable and Reconfigurable Conveyor Systems [1] as a starting point, critical issues in resource virtualization can be explored using the CCS Simulator.
CHAPTER III
CLOUD CONVEYOR SYSTEM SIMULATOR DESIGN

Procedural and object-oriented approaches are two fundamentally different ways to design a software system. Both these approaches aim to capture the essential characteristics of a system as a model in software data structures, and update this model to reflect operational constraints and behaviors.

In the procedural approach, the fundamental tasks are to identify a set of data structures and define functions to manipulate these structures in coherent and efficient manner. When the number of these functions increases, methods such as functional decomposition can be used to efficiently identify and organize the functions. The dynamic invocations that affect these functions can be organized as a function call tree. While this approach was used as the basis for software design for several decades, it is well-known that this approach does not scale efficiently. The level of abstraction provided and the support for type safety is limited and, consequently, the software implemented tends to be brittle and error prone. Since the semantics of the application domain is dispersed in the data structures and algorithms used, domain knowledge and experience cannot be readily used to design verification procedures.
3.1 Object-Oriented Approach

An object-oriented approach offers a different method to software design, implementation and maintenance. In this approach, objects are created to directly mimic artifacts in an application domain. These objects interact by exchanging messages and this sequence of object interactions represents the evolution of computation. For this reason, the object-oriented design approach is sometimes referred to as one in which *Simulation is a metaphor for Computation* [7]. The cloud conveyor system (CCS), which is the focus for simulation in this thesis, is a novel conception for a material handling system. The application domain has several artifacts that have specific behaviors and, hence, the object-oriented approach is well suited for designing the CCS Simulator.

There are key principles that are central to the object-oriented design approach. The essential ideas are to utilize *Abstraction and Encapsulation, Polymorphism, Inheritance*, and *Object Interactions* to achieve software system objectives. A challenge in object-oriented design is to ensure that these principles manifest in the design; further, the objects created must mimic domain artifacts in a simple and natural manner. These ideas are reflected in classical object-oriented designs that are well-documented in the literature [12, 19].

In object-oriented design for a system, every class is an *Abstraction*. An abstraction is the purposeful suppression of certain details so as to emphasize other details. For example the class `CCS_Unit` suppresses all the sensing and electromechan-
ical actuation details of the mobile conveying units and emphasizes *only* the behavior of the units CCS. The power of this abstraction is fully realized in the class structure by encapsulating the attributes and behaviors of units necessary to achieve the system objectives. The attributes are variables that are declared and used in the class, and the behaviors are codified as methods. Figure 3.1 shows the attributes of the class `CCS_Unit`. Notice that these attributes directly correspond to physical attributes in the domain such as velocity, offset and position.

```java
private int id;
private String label;

private boolean isHome;

private int offset;
private int direction;
private double velocity;

private CCS_Track track;
private CCS_Entity entity;

private CCS_Position position;
private CCS_Position homePosition;
```

Figure 3.1: *Attributes of the CCS_Unit class.* These attributes are all tagged `private`. This means that only the methods of this class can change values of these attributes.

Figure 3.2 shows the methods of the class `CCS_Unit`. The method with the same name as the class is the mandatory constructor method. This method is invoked by the run-time whenever a new instance of the class is created. Notice that there are a set of access methods that are used to get and set values of the attributes of the class. These methods ensure that the state of every `CCS_Unit` object is only updated
as intended. The methods `CCS_Unit::move` and `CCS_Unit::goToHomePosition` represent the behaviors of the `CCS_Unit` objects. Recall that these are the horizontal and vertical units that move along tracks in the CCS.

```
public CCS_Unit(int id, String label, int x, int y, int offset, double velocity) {}  

public void move() {}  
public void addEntity(CCS_Entity e) {}  
public void removeEntity() {}  

public void setOffset(int offset) {}  
public void setVelocity(double velX) {}  
public void setTrack(CCS_Track track) {}  
public void goToHomePosition() {}  

public int getID() {}  
public int getOffset() {}  
public int getX() {}  
public int getY() {}  
public int getDirection() {}  

public double getVelocity() {}  
public boolean hasEntity() {}  
public boolean isHome() {}  

public String getLabel() {}  
public CCS_Track getTrack() {}  

public CCS_Entity getEntity() {}  
```

Figure 3.2: Behaviors of the CCS_Unit Class. Methods of the class provide an interface for interaction. Behavior of an object is the expression of these interactions.

The `direction` attribute of the `CCS_Track` object on which a `CCS_Unit` moves determines whether the unit is a horizontal unit \( h_i \) or a vertical unit \( v_i \).
Such a design offers the flexibility to reuse the same unit irrespective of the shape and the geometry of the tracks.

It is common practice in the object-oriented approach to also think of a class as a template. Thus, a class can only be used to instantiate objects; each instance of the class has a private collection of the class attributes and offers the set of behaviors encoded in the class definition. After instantiating one or more objects, these objects interact by invoking methods in other objects. In this manner, when all the classes defined in the software system have direct and natural mapping to domain artifacts, the corresponding instances of these artifacts interact by invoking methods in other objects. An object may also invoke its own behaviors. In the CCS Simulator, there is only one class definition for the class \texttt{CCS\textunderscore Unit} while there is a separate instance for each horizontal unit $h_1, h_2, \cdots h_M$ and each vertical unit $v_1, v_2, \cdots v_N$. Similarly, there is a class definition for \texttt{CCS\textunderscore Track} and several instances for each horizontal and vertical track.

Inheritance simplifies the implementation by allowing attributes and methods declared in a parent class to be used by objects of one or more child classes. Thus, the child classes do not have to re-implement methods, or redefine attributes, that have already been implemented in parent classes. While eliminating redundancy in code, inheritance also ensures that child classes have access to correct implementations from parent classes. For example, the \texttt{CCS\textunderscore Track} class is a parent class from which the class \texttt{CCS\textunderscore LinearTrack} inherits attributes (e.g., Input and Output objects, Units, and Position) and behaviors.
Polymorphism enables run-time substitution of objects in the system. This idea is explained through a specific example. The CCS Simulator has a `CCS_Observer` and a `CCS_Subject` interface. An object that implements the `CCS_Observer` will register itself with a subject whose changes it needs to be notified of. When an object implementing the `CCS_Subject` interface makes changes, it will notify all of its registered observers. For example, the simulator allows two different views of the CCS — one called a *Debug View* and the other called a *Track View*. The debug view presents a text-based representation of the CCS and the track view presents an animation of the units and entities moving along the tracks as illustrated in Figure 1.1. Objects that wish to be notified must inherit the `CCS_Observer::update()` method and may offer a specialized implementation of the behavior. The debug view may use the notification in a manner that is different than how the track view uses the same information. However, the CCS system-model classes do not need to know which object is actually receiving the update notification and what local behavior will be executed in response. Thus, through the use of polymorphism, the actual objects that receive messages during run time can be anything that can accept the notification. The power of this feature will become clear later when the full complexity of CCS is revealed [26].
Despite its advantages in offering a rich set of abstractions to design complex systems, the number of classes involved and the multiple object instances of each class can be overwhelming, especially when the size and complexity of the software system increases.

3.2 Model-View-Controller (MVC) Framework

The MVC Framework is an architectural rubric for organizing complex software systems that involve user interaction. Without enforcing or imposing any design constraints, the framework offers a style for the design and implementation of large software systems. Since user interaction is a very important function for the CCS Simulator, MVC is well suited for the design of this simulator. In this approach, there is three-way separation of concerns as follows:

1. *Represent* the underlying application (system-model),

2. *Present* the system-model to user(s), and

3. *Enable* user(s) to interact with the system-model.

The objectives of the above three concerns are achieved by behaviors that are encapsulated in different classes. The system-model classes abstract domain artifacts and the interactions among the objects of these classes reflect direct domain interpretation of the behavior of the artifacts. There is a subtle connection between the classes that present a view of the system-model to the user and the interactions that the user can effect in that view. For this reason, View-Controller pairs were designed
by noting that the controller was specific to a particular view. Thus, to enable new views of the same system-model, a new controller was also designed.

3.2.1 The Model Classes

The model objects in the MVC Framework are a representation of the CCS System that has been instantiated. These objects exists as long as the instantiation is not freed, i.e., as long as the CCS Simulator is executing. The design objective is to put as little information as necessary in this model. It is also critical, simultaneously, to make sure that all the information that will ever be needed in any View-Controller can be derived from the information in the model classes and objects. Thus, the design of the model classes is a challenging task that must balance efficiency of implementation and account for future evolution of the system.

The model classes enumerated below were designed for CCS. For each class, the description shows the relevance of the abstraction to the application domain.

1. **CCS.Track**. The geometry of the tracks affect the behavior of the units. The horizontal and vertical units move along the tracks.

2. **CCS.Entity**. The objective of CCS is to transfer entities from Inputs to Outputs. Entities ride on the units that move along the tracks.

3. **CCS.Input**. Entities arrive on CCS via Inputs.

4. **CCS.Outputs**. Entities leave CCS via Outputs.
In addition to the above classes that had direct physical correspondence in CCS, several other model classes were designed. For example the class CCS_Model is necessary to abstract the model. Other classes designed were CCS_Grid, CCS_Field, CCS_Position, CCS_Intersection, CCS_Rendezvous, CCS_RendezvousPattern and CCS_LinearTrack.

3.2.2 Model Controller Classes

The CCS system-model contains the configuration for a CCS. For each instance of the system-model, there is an CCS_mController object created. This is the only object that can interact with the system-model objects. The CCS_mController is instantiated by a view-controller representing the control panel and is then used by the view-controller instances to get and set system-model attributes. The mController is responsible for keeping system time during execution of CCS activity. It implements the CCS_Subject interface so that classes implementing the CCS_Observer interface can register themselves for updates.

3.2.3 View-Controller Classes

Users require multiple views to fully understand and explore CCS. For example,

1. a view that presents an animation of the units moving and entities transferring through CCS may derive its own system-model objects to support the animation and user interaction.
2. a configuration view will allow users to specify a CCS configuration; the user specifications will be held in a local system-model and this system-model will be updated before the simulation is started.

3. the simulator must always maintain a default view called the *Debug View* that can be used to diagnose the simulator behavior if the intended objectives of the view are not achieved.

Each View-Controller works with a system-model through the *mController* that has been instantiated by a configuration View-Controller that allows users to configure the CCS. Some view-controllers may have their own model objects that transform system-model data into a more suitable formats for the view. For instance, the Debug View has a *grid* object of *CCS/Grid*. The *grid* takes the object-oriented system-model and transforms it into a string matrix for display. To demonstrate the approach and establish a simulation platform, the following views have been designed and validated for CCS:

1. Config View,

2. Debug View, and

3. Animation View with Rendezvous Pattern Tracker

A view-controller was designed for each of the above views. Each view-controller is a service-object for the *mController*. In the current design, the configuration view-controller owns the *mController* object and all the view-controllers use this instance to accomplish the tasks necessary for the view.
The following views are examples of view-controllers that can be designed and implemented in the future to enhance this CCS Simulation platform:

1. **Design View.** Users can view Input-Output Connectivity graphs, minimum latencies for each Input-Output pair, maximum achievable transfer rates between subsets of Inputs and Outputs, Utilization, Schedulability, and other non-performance measures that can be achieved in a particular configuration of CCS.

2. **Virtual Conveyor System View.** This view would allow users to specify a virtual conveyor system topology in a manner that is described in [25]. Such virtual topologies would be embedded to the actual CCS using transformation techniques.

Further, the above view-controllers, the current Animation View can be extended by augmenting it with additional system-models and *mControllers* for real-time control of the units, verification, and planning.

3.3 Class and Object Relationships

Classes have different relationships among themselves. A clear understanding of these relationships and their consistent use is critical for the efficient design of software systems. Among such relationships, the important ones are *has-a*, *is-a*, and *uses*. The *has-a* relationship is important because it represents owner-member relationships. Suppose class *A* *has* another class *B*; then an object of class *A* must instantiate an
object of class \( B \) before it can be used. In the CCS Simulator, \texttt{CCS\_Model} \textit{has-a} \texttt{CCS\_Field}; the \texttt{CCS\_Field} class \textit{has-a} \texttt{CCS\_Track}, etc.

The \textit{is-a} relationships are also important because these represent the inheritance relationships in the system. For example, a \texttt{CCS\_LinearTrackView} \textit{is-a} \texttt{CCS\_View}. Similarly the \texttt{CCS\_DebugView} \textit{is-a} \texttt{CCS\_View}.

![Diagram](image_url)

Figure 3.3: Architecture of the CCS Simulator. There are three dominant relationships among classes illustrated here. The classes grouped by architectural concern, model classes are on the top, controller classes in the middle, and view classes are on the bottom. The view classes must go through controller classes to get to model data.

In addition to the above two relationships between classes, the design must also identify \textit{uses} relationships. From a software implementation perspective, an
object that uses another object must have a reference to the target object. It is necessary to make sure that such references are correctly created and systematically propagated in the system.

Figure 3.3 represents the three kinds of relationships between the classes that were designed. Note that in this figure, the system-model classes and the view-controller classes are shown in different groups. The system-model classes are on top; the model controller, \texttt{CCS\_mController}, facilitates interaction between the system-model and the view-controllers; there is only one instance of this class per system-model. The \textit{mController} is responsible for creating, updating and maintaining the classes that comprise the system-model.

Figure 3.4: \textit{Main object interaction}. \texttt{mController} maintains system time and coordinates all activities. As time advances, the units move on tracks. These tracks intersect and when multiple units meet at these intersections, a rendezvous occurs. It is at these rendezvous that cyber-transfers can happen, dictating conveying behavior of the CCS.

While the \textit{uses} relationship between classes is generic and only identifies that objects of the two classes interact during execution, the relationships between
objects that are instantiated can, in fact, be quite complex. Figure 3.4 illustrates the dominant interactions between objects in the CCS Simulator.

Notice from Figure 3.4 that the \textit{mController} object is responsible for advancing system time. The units \textit{ride on} tracks and the tracks \textit{meet at} intersections. A rendezvous \textit{occurs} between two units at some intersection. While this paragraph is easy to read in English, it is important to realize that the domain artifacts described here are in fact classes that have been designed as described in the previous section.

3.4 Algorithms

The heart of the CCS lies in the \textit{mController} object. All interaction between the view, which presents the model state to the user, and the model, which stores the system state, must occur through the \textit{mController}. The algorithm for the \textit{mController} thread is listed in Algorithm 1. While the application is running, the \textit{mController} advances \textit{units}, increments system time, checks for and creates \textit{rendezvous}, signals cyber-transfers, and updates \textit{input} objects to increase their part queue. Having one controller allows for data integrity as changes can only be made to the model by one thread. However, since multiple threads read the model, thread safety is still a concern and is addressed in Section 3.5.
Algorithm 1: CCS_mController thread

while true do
    if not Paused then
        /* move units and increase system time */
        \( u_i.correction() \forall u_i \in U; \)
        \( t = t + 1; \)
        for \( 1 \leq j \leq M \) do
            for \( 1 \leq i \leq N \) do
                /* check for two units at the same position */
                if \( h_j.position == v_i.position \) then
                    /* create a rendezvous if two units are at the same position */
                    create \( r^j_{j,i} \);
                    /* attempt a cyber transfer between the two units at the rendezvous */
                    if transfer scheduled then
                        cyber-transfer entity between \( h_j \) and \( v_i \);
                    /* update the input entity count */
                    if \( h_j.inputConfigured \) AND \( h_j.inputRate \neq 0 \) then
                        if \( t \mod h_j.inputRate \) == 0 then
                            entityCount++;
                    /* update views to the new model configuration */
                    notify observers;
    /* update views to the new model configuration */
    notify observers;

3.5 Threads and Thread Safety

Thread safety is implemented in the CCS model using Java synchronization primitives. Public methods encapsulate class data attributes in a synchronized(lock) block to protect values from being written by two threads simultaneously, or being read while being written by independent threads. This approach is more efficient than locking the entire method since it allows for more concurrent executions.
Instances of $mController$ and $vController$ were implemented as separate threads. The instance of the class $CCS.mController$ keeps track of time and manipulates the state in the instances (objects) of the system-model. If multiple instances of the system-model are needed, then new instance $CCS.mController$ will be created for each; this could happen, for example, when the user launches two separate processes of the CCS Simulator. The $vController$ object manages the views of the CCS for a given configuration. If multiple configurations of $M$ and $N$ are needed, then multiple $vController$ instances will be created for each such configuration.

The $CCS.View$ class implements the Java Runnable Interface [26]. When a new view is requested, by undocking the current view from $vController$ or switching between linear track view and matrix debug view, a new thread is created to run the view. If the view is an undocked window, the thread will terminate when the window closes. If the view is docked, it will terminate when the docked view is switched or the application is exited.
CHAPTER IV
IMPLEMENTATION, PERFORMANCE ANALYSIS AND SYSTEM PROFILING

The design of a software system provides a road map for the implementation. In the context of an object-oriented design, all the characteristics of the design are captured in the relationships between classes and the interactions between the objects. Further, since the simulator was designed in the Model-View-Controller (MVC) framework, it was necessary to ensure that the separation of concerns intended in the framework was in fact achieved.

4.1 Implementation

The object-oriented design described in the previous chapter was implemented using the Java programming language. An integrated development environment for Java called NetBeans [11] was used to develop the simulation platform. NetBeans offered a simple graphical user interface to configure user views and interactions. The Java code necessary to operationalize these views and interactions was generated by NetBeans directly and hence this reduced the potential for errors and bugs in this part of the code. In addition, NetBeans enabled easy integration of several tools that are necessary to verify the design and performance of the simulator and maintain a robust CCS Simulation Platform to support future investigations.
A software tool called the *Profiler* that is integrated with NetBeans was used to examine the objects that were instantiated during the execution of the simulator. This examination made it possible to verify that the object and class relationships intended in the design were in fact realized in the implementation. The profiler also made it possible to monitor the amount of time spent in executing various threads of the software system. This level of detail made it possible to confirm that the implemented software executed as intended in the design. Finally, an auxiliary software plugin called the *JaCoCo Code Coverage Analysis* tool, was used to examine the portions of the code that were executed in a particular execution. This approach makes it possible to design a comprehensive test suite to verify and maintain the CCS Simulator in the future. The complete design for the CCS Simulator resulted in 25 classes and 3,885 lines of code in Java; the number of object instances depends on the actual configuration of the CCS, i.e., Tracks, Units, Entities, Offsets, etc.

4.2 Using the Simulator

Figure 4.1 shows the initial configuration panel that is displayed to the user when the simulator is launched. Users must provide values for $M$ and $N$, which are the number of horizontal and vertical tracks, respectively, in the particular configuration that the user wishes to explore. When the *Generate* button is clicked, the system records these values, creates the object instances corresponding to these values, and enables users to further specify attributes of this configuration. For example, the input text boxes through which users can configure the units are enabled.
Users must now specify the number of units in the configuration. For this purpose, users must click in the boxes immediately below the values of $M$ and $N$. The user click will launch an instance of the unit configuration dialog shown in Figure 4.2. Through this dialog, users can choose either to enable or to disable a unit riding on a specific track; for each such unit selected, users may also provide a value for the initial offset if the default value of 0 is not desirable. Users must complete this interaction by clicking on the OK button; results of the user’s interaction in the dialog will be presented on the CCS Configuration Panel.

After configuring the number of tracks and units on these tracks, users must now configure the locations of the Inputs and the Outputs. This can be accomplished by clicking on the *Configure IO* button. The CCS Simulator will now launch the configuration window for inputs and outputs that is shown in Figure 4.3. In this
dialog, users may select one or more inputs as required. For each Input, users can also specify the constant rate (period) with which new entities arrive on the CCS via that specific Input. The entities that are generated arrive at each Input, remain on the Input in a Queue. Each entity can only be transferred to a unit via a cyber-transfer. Users can only select the location of the Outputs and no other configuration parameters are required.

After the configuration for the CCS simulation has been specified, users can select one of several views that are supported. The Debug View is shown in Figure 4.4; this view will display a textual representation of the units in a form that looks like a matrix and is useful for debugging the system behavior. The Track View is shown in Figure 4.5. This view presents an animation of the units moving along the tracks with entities as the system time, $t$, evolves. Each Rendezvous event is highlighted
Figure 4.3: IO configuration. The input/output configuration screen allows users to specify the number and location of inputs and outputs. The rate at which new entities arrive at the input are also configurable here.

with an orange circle. Users can choose to display a graphical view of the Pattern of Rendezvous Events by clicking on the Show Pattern button in the Configuration Panel. When the system time has evolved sufficiently, i.e., when $t \geq (2 \times \text{period})$, the CCS Simulator will display the complete Pattern of Rendezvous Events.

In addition to the above features, the Configuration Panel also allows users to record an event log to an output file. Users can step through a simulation by using the Step button. An executing simulation can be paused using the Pause button and resumed using the Go button. By clicking on the Clear button, users can delete all object instances in the current simulation; they must now enter a new configuration as outlined above to continue exploring the CCS.
Figure 4.4: The Debug View. This view displays a textual simulation of the configuration for debugging simulator behavior.

The remainder of this chapter describes the results from verification activities that were carried out to ensure that the design intent was indeed captured in the implementation.

4.3 Class Relationships

In most of the computing environments, the Java Virtual Machine (JVM) is responsible for executing programs that are written using the Java language. \(^{1}\) The CCS Simulator was executed on a JVM in a variety of computing environments that included Linux, MacOS, and Windows.

\(^{1}\)The exception is when the code is compiled to execute in native mode for embedded and other special-purpose environments.
The JVM allocates all instances of objects in a special area of the memory of the operating system process that executes the simulator called the Heap. The profiler allows users to dump, i.e., record this part of the memory in a file, and examine the Heap using special tools. The recorded file is commonly called a Heap Dump. Special tools such as a Heap Walker can be used to examine the Heap Dump.

The netBeans profiler also offers a Heap Dump tool and a memory profiling tool called the Allocation Stack Trace to examine the heap.

The netBeans profiler was used to verify that the class relationships were implemented as designed as illustrated in the following. Consider the has-a and uses relationships of the CCS_mController. The has-a relationship was verified using the Allocation Stack Trace. The uses relationship was verified using the Heap Dump.

The simulator was executed under control of the profiler and CCS was con-
figured during runtime. A profiling snapshot was captured to record the memory allocations completed until a specific time; simultaneously, the Heap was dumped. Figure 4.6 shows the Allocation Stack Trace focused on the CCS.mController. Notice from the column labeled Allocations and the column labeled Live Objects that there is only one mController object; thus, this observation confirms that the design intent of having a single controller to manage the CCS system-model was achieved. There were no inadvertent errors the code that caused more (needless and unintended) objects to be instantiated.

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Allocation Call Tree</th>
<th>Bytes Allocated</th>
<th>Objects Allocated</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCSController.CCS.mController</td>
<td></td>
<td>40 B (100%)</td>
<td>1 (100%)</td>
</tr>
<tr>
<td>CCSView.CCS.Linear_vController.&lt;init&gt; ()</td>
<td></td>
<td>40 B (100%)</td>
<td>1 (100%)</td>
</tr>
<tr>
<td>CCSView.CCS_GuiContainer.&lt;init&gt; ()</td>
<td></td>
<td>40 B (100%)</td>
<td>1 (100%)</td>
</tr>
<tr>
<td>CCSView.CloudConveyorsSim.main(String[])</td>
<td></td>
<td>40 B (100%)</td>
<td>1 (100%)</td>
</tr>
</tbody>
</table>

Figure 4.6: Verifying Has relationship. Allocation Stack Trace shows that the CCS.vController object instantiated the CCS.mController object in column 1. Column 4 (Live Objects) shows that there is only one CCS.mController object as intended in the design. Column 5 (Allocations) confirms that only one such object was ever instantiated in the execution.

Since the Heap Dump can be quite large, it was filtered to focus on CCS.LinearTrackView objects. Figure 4.7 shows that the CCS.LinearTrackView object indeed uses a CCS.mController object. Since the Allocation Stack Trace confirmed that there was only instance of CCS.mController, this instance could have been allocated only by the vController as intended in the design.
4.4 Multi-thread Design

Threads are important to achieve performance on modern many-core computer systems or cloud execution platforms. An examination of the execution time of each thread also makes it possible to understand whether the implementation is consistent with the design of the system.

Recall that in the design described, the CCS system-model is maintained by the mController. This was implemented as a separate thread. The system-model is presented to the user by the vController thread to allow the users to configure an instance of the CCS. All user actions such as keyboard input and mouse movements are intercepted by the vController view-controller and posted to the EventQueue that is maintained by the JVM. The main thread is only responsible for initiating the simulator and launching initial simulator thread.
Figure 4.8 shows the execution time of the above threads in the CCS simulator. The figure on top shows the situation when the animation view is enabled. The figure on the bottom shows the same situation without the animation view. Notice that as the number of units ($M$ and $N$) increase, the execution time of the main thread does not increase. This is because the main thread only launches the simulation and does little else. In both cases the execution time required by the $mController$ increases linearly. This increase is expected because there are more instances in the system-model as the number of units in increased. On closer examination, the execution time of this thread can be noticed to increase slightly faster than a linear increase. This shows that the implementation is indeed efficient because as the number of units increase, the number of intersections increase as the product of the number of units along each set of tracks. Even when there is no rendezvous in an intersection, the $mController$ must check to make sure that no rendezvous has occurred. Consequently, the execution time of the $mController$ increases. Despite this increase, the execution time of the $mController$ is close to linear irrespective of whether the view is used or the view is not used.

When no view is used, the system-model is executed and no output is presented to the users. This mode of execution is useful when a simulation must be carried out involving hundreds of units. Notice from the figures that the time required by the $vController$ is minimal because there is no significant difference in the execution time when the view is used (top) or not (bottom). However, notice from the figure on top that the execution time of the $LinearTrackView$ increases as the size
Figure 4.8: Thread performance. Thread execution time with (top) and without (bottom) a view. Notice from the two figures that as the number of units increase, the execution time of the mController increases as expected. Notice from the top figure that the Linear Track View has the largest execution time when the view is used. This confirms that the intended separation of concerns have been achieved in the thread-safe design. Error bars indicate the standard deviation in the sample.

This confirmed that the separation of concerns between the vController and LinearTrackView were indeed achieved as intended in the design.

The overhead incurred in each thread when presenting the view to the users is shown in Table 4.1. For each thread, the ratio of the execution time without the view to the execution time with the view is shown. When this number is close to 1, there is no penalty for showing the view. A number less than 1 indicates a penalty for running the view and a number greater than one indicates a penalty for not running a view.
Table 4.1: View overhead. This table shows the overhead for the major application threads when displaying the graphic view.

<table>
<thead>
<tr>
<th>Thread</th>
<th>( M = 7, N = 3 )</th>
<th>( M = 10, N = 6 )</th>
<th>( M = 13, N = 9 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>0.992</td>
<td>0.976</td>
<td>0.981</td>
</tr>
<tr>
<td>EventQueue</td>
<td>1.05</td>
<td>1.04</td>
<td>1.11</td>
</tr>
<tr>
<td>mController</td>
<td>1.17</td>
<td>1.07</td>
<td>1.08</td>
</tr>
<tr>
<td>vController</td>
<td>0.9055</td>
<td>0.826</td>
<td>0.701</td>
</tr>
</tbody>
</table>

Table 4.1 shows that the main thread stays consistent whether or not a view is shown. The vController thread runs consistently when no view is shown, and increases with the grid size when the view is shown. Therefore its ratio is decreasing because the denominator of the ratio increases as the numerator stays flat. The mController and EventQueue threads run on the processor longer when there is no view. This is likely because there is no contention with the view thread for the CPU resource.

To ensure thread-safety in the CCS system-model, i.e., to make sure that there were no read or write conflicts between threads, Java synchronization primitives were used. Synchronized statements were added to critical sections of code whenever multiple threads could access the system-model simultaneously. Figure 4.9 shows the memory usage with and without thread implementations.

As shown in Figure 4.9, there is a small overhead for using multiple threads. The benefit of using threads is illustrated in Figure 4.10. Notice here that when in a multi-threaded design, the two main threads mController and vController have considerably smaller execution times in the threaded implementation. This confirmed that the multi-threaded design indeed delivered the performance as expected.
Figure 4.9: Memory Usage with (top) and without (bottom) threads. The figure on top shows that the number of bytes on the Heap for the two implementations are similar. The figure on the bottom shows the number of object instances on the heap. In both cases, notice that the overhead of using threads is not excessive. The small increase shown is expected.
Figure 4.10: Benefits of multi-thread design. The figure on top shows that the execution time of the mController is significantly less when using a multi-threaded design. Similarly, the figure on the bottom shows that the execution time of the vController is also improved in a multi-threaded design. Error bars indicate the standard deviation in the sample.
To confirm that the different threads were indeed launched and supported on the system, the NetBeans profiler was used. Figure 4.11 shows a snapshot of the profiler. Notice that there are two instances of `LinearTrackView` (on lines 1 and 3) and one instance of `DebugView` (on line 7). There is one instance of `vController` (on line 5), one instance of the `mController` (on line 6). The `main` thread is on line 4 and the JVM `EventQueue` is on line 2. Thus this confirms that the design is indeed a multi-threaded design as described in this section.

Figure 4.11: *Multi-thread verification.* Profiler view showing multiple instances of `CCS_LinearTrackView` and one instance of `CCS_DebugView`

4.5 Method Invocations

The number of times methods in various objects were invoked was also examined to analyze the performance of the CCS Simulator. The invocation count of a method revealed whether it is being invoked unnecessarily. The *Call Tree* extracted using the NetBeans profiler was useful for this task. To extract a call tree for a specific execution of the CCS Simulator, it was necessary to configure and execute the simulator.
An example call tree is shown in Figure 4.12. In this execution, the simulator was configured with $M = 7$ and $N = 3$. The simulation was executed for 100 time units; each input injected an entity into CCS every time unit. The call tree shows that the `CCS::Input::injectPart` method was invoked 700 times (fourth line from the bottom of the figure) because there were seven inputs in the configuration. By double-clicking on an entry revealed the code (Figure 4.13) that was in fact executed for each invocation. The Heap was also inspected to confirm (as shown in Figure 4.14) there were 707 instances of `CCS::Entity` class allocated. This includes the initial loading of the `Input` at system time 0.

![Call Tree](image)

**Figure 4.12: Verification of method invocations.** Verification of invocations of the `CCS::Input::injectPart` method for a simulator configuration where $M = 7$ and the simulation ran for 100 time units.

A “Hot Spot” is an area of code that is executing often. An examination of the hot spots can quickly reveal potential issues. The NetBeans profiler offers a view to list program hot spots in decreasing order of the processor execution time as shown in Figure 4.15. An examination of hot spots for the CCS Simulator revealed that the `CCS::RendezvousPattern::initializeRendezvous` method was invoked being five times. In contrast, the design intended this method to be
public void injectPart() {
    CCS_Entity e = new CCS_Entity(_entityID++);
    e.setInput(this);
    synchronized(this) {
        _entities.offer(e);
    }
}

Figure 4.13: Source code of the CCS_Input::injectPart method. The source code shows that for every invocation of CCS_Input::injectPart, an instance of CCS_Entity is created.

Figure 4.14: Verifying object allocation. Instances of the CCS_Entity class show the expected number of object instances for the number of times CCS_Input::injectPart is called.

invoked only once. This method is used to set up an empty list of rendezvous for every tick in period of the pattern of rendezvous events. The hot spots revealed that this method was invoked in the CCS_RendezvousPattern constructor and every time $M$ or $N$ is updated. Thus, when the model was created with $M = 0$ and $N = 0$, the method was invoked; when $M$ was initialized, the rendezvous pattern was cleared and the method was invoked again; when the user initialized a value for $N$, the method was invoked once again. After this examination through the profiler, the CCS_RendezvousPattern::initializeRendezvous call was removed from
the CCS::resetModel method to eliminate unnecessarily initializing the rendezvous pattern multiple times.

<table>
<thead>
<tr>
<th>Hot Spots - Method</th>
<th>Invocations</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCSModel.CCS_Position.getX()</td>
<td>102,215</td>
</tr>
<tr>
<td>CCSModel.CCS_Position.getY()</td>
<td>35,415</td>
</tr>
<tr>
<td>CCSModel.CCS_Intersection.equals(Object)</td>
<td>28,456</td>
</tr>
<tr>
<td>CCSModel.CCS_Unit.getX()</td>
<td>21,240</td>
</tr>
<tr>
<td>CCSModel.CCS_Unit.getY()</td>
<td>18,125</td>
</tr>
<tr>
<td>CCSModel.CCS_Unit.getY()</td>
<td>8,640</td>
</tr>
<tr>
<td>CCSModel.CCS.Track.getMax()</td>
<td>5,760</td>
</tr>
<tr>
<td>CCSModel.CCS_Intersection.getY()</td>
<td>5,525</td>
</tr>
<tr>
<td>CCSModel.CCS_Track.getMin()</td>
<td>3,977</td>
</tr>
<tr>
<td>CCSModel.CCS_Track.getNumberOfUnits()</td>
<td>3,676</td>
</tr>
<tr>
<td>CCSModel.CCS_Track.getNumberOfUnits()</td>
<td>3,270</td>
</tr>
<tr>
<td>CCSModel.CCS_Track.getOutput()</td>
<td>2,535</td>
</tr>
<tr>
<td>CCSModel.CCS_LinearTrack.getDirection()</td>
<td>2,529</td>
</tr>
<tr>
<td>CCSModel.CCS_Intersection.getUnits()</td>
<td>1,806</td>
</tr>
<tr>
<td>CCSModel.CCS_Intersection.clearUnits()</td>
<td>1,743</td>
</tr>
</tbody>
</table>

Figure 4.15: The Hot Spots view of the profiler shows code that executes often. By examining hot spots in the profiler, the software was redesigned to eliminate an unintended sequence of calls to the same method.

4.6 Memory Usage

The memory footprint of various components of the CCS Simulator and how this changes when the number of units change was another detail that was examined to analyze the performance of the system. The Heap Dump was used to carry out this task. Nominally, the Heap visualization tools can list the ratio of the number
of instances of a specific classes with respect to the total number of instances in the system. This can be further filtered to focus on specific instances for a specific application.

As noted in the preceding section, the call tree was used to confirm that the \texttt{CCS\_Input::injectPart} method was not being invoked needlessly; as illustrated in the Heap Dump, Figure 4.14, there are 700 instances of \texttt{CCS\_Entity}. Each such object is created when a new entity is injected into CCS. This confirmed that entity instances are not being created anywhere else in the code and hence the implementation realized the design as intended. It is also evident that the correct number of \textit{units} and \textit{tracks} have been allocated.

4.7 Code Coverage

Because the CCS Simulator implementation spans several classes and methods, it is important to make sure that the software is adequately exercised via the tests. This task was carried out using the \textit{JaCoCo Code Coverage} plugin for NetBeans. A second plugin for NetBeans called \textit{JUnit} was used to generate test suites for the Simulator. Unit tests generated by \textit{JUnit} provided shell test scripts for every method of a selected class. The shells must be filled in with code that can exercise the corresponding method — these are known as test cases. A test case can instantiate an object, invoke object methods and test the results of the method calls.

The code coverage tool records the actual lines of code that were executed during the execution of a specific test case. By capturing all the lines of code executed
in a test suite, it is possible to conclude whether or not the test suite adequately exer-
cises the designed software. Such a quantitative examination of coverage is important
for the CCS Simulation Platform.

The code coverage plugin can be activated from the NetBeans IDE. Once
coverage is activated and the tests for the project are executed, the code coverage
statistics can be viewed.

Test cases were designed for every method in the CCS system-model and
statistics for each method were gathered. Code coverage results are shown in Fig-
ure 4.16. Notice that lines of code that were executed in the JUnit tests are indicated
with light gray highlighting whereas lines of code not executed are highlighted in dark
gray.

4.8 Discussion

The results presented in this chapter demonstrate the effectiveness of the design for
the CCS Simulation Platform. Examination of the Heap Dump and the Allocation
Stack Trace helped to confirm the relationships between classes and objects that
were described in the preceding chapter. By examining the number of instances of
individual objects, and the number of invocations of specific methods in the objects,
it was possible to refine the implementation. Results from the code coverage tool
demonstrated that when a suite of tests are rigorously developed it is possible to
validate the design of the simulation platform.
public void removeUnit(int track, int offset, int velocity,   
CCS_Track.direction direction) {

    CCS_Track t;
    CCS_Unit u;
    synchronized(this) {
        switch (direction) {
            case HORIZONTAL:
                t = _hTracks.get(track);
                u = new CCS_Unit(t.getID(),("h"+t.getID()),t.getMin().getX(),
                                t.getMin().getY(), offset, velocity);
                _hUnits.remove(u);
                t.removeUnit(u);
                break;
            case VERTICAL:
                t = _vTracks.get(track);
                u = new CCS_Unit(t.getID(),("v"+t.getID()),t.getMin().getX(),
                                t.getMin().getY(), offset, velocity);
                _vUnits.remove(u);
                t.removeUnit(u);
                break;
            default:
                break;
        }
    }
}

Figure 4.16: Code Coverage. The figure is a view of the code coverage plugin and shows the lines of code that were executed (light gray) and the lines of code not executed (dark gray) during a test case created for the CCS_Model::removeUnit method
CHAPTER V

CONCLUSIONS

This thesis described a comprehensive simulation platform for the Cloud Conveyor System, which is a novel approach to material handling. The object-oriented approach was well suited for the design of this simulator because it helped to directly capture domain artifacts in software. The encapsulation and abstraction provided by the classes made it possible to focus on how the system-level behaviors could be achieved simply from the interactions between object instances. Key object-oriented design ideas such as inheritance and polymorphism were effectively exploited in the design and implementation of the system.

The Model-View-Controller framework was an effective rubric for designing this platform. The core CCS system-model classes represent the artifacts of the domain — units, entities, tracks, positions, and rendezvous. A single \textit{mController} object managed the system-model to ensure that users could interact with this model through multiple views. The need for multiple view-controllers was discussed and two different view-controllers were implemented to demonstrate the effectiveness of the design. The critical relationships between classes and objects in the system were discussed and presented.
Based on the profiling results described in the preceding chapter, it can be concluded that the design intentions were indeed achieved. By examining the Heap Dump and the Allocation Stack Trace, it was possible to conclude that the design intentions and relationships were realized as intended. Examination of the number of instances of different classes and the number of invocations of specific methods helped to refine the implementation. The code coverage analysis demonstrated the effectiveness of the development environment selected.

The platform described in this thesis is comprehensive. The core CCS system-model can support many more views. For example, different track geometries and different strategies for control of the units to modify speed or distance traveled are some of the options that can be added by designing new view-controllers. The NetBeans profiler is an effective tool for verifying that the design intentions were not violated in the implementation. Finally, by designing a comprehensive test-suite to verify the functional capabilities of the system, this CCS simulation platform can be easily maintained and integrated into ongoing research investigations.


