Production Cell Simulation and Control Software

Author: Oleksandr Shynkariuk
Supervisor: Welf Löwe
Date: 2013-11-21
Course Code: 5DV00E, 30 credits
Level: Master
Department of Computer Science
Abstract

Control software plays an important role in industrial automation. Several domain-specific languages for control software implementation were designed and accepted in industry. Structured Text (ST)[1] is one of these languages. It is standardized by IEC 61131-3 standard and designed for programmable logic controllers.

In this thesis two implementations of control software were developed: Java implementation as a reference model, and ST implementation as a test scenario for ST to C compiler. The functionality of both implementations was compared using the simulation model. It was concluded that resulting ST control software provides the same functionality as its reference model.

Key words: Control software, production cell simulation, Java, Structured Text, TCP communication.
# Table of content

1 Introduction ...................................................................................................................... 1  
   1.1 Background .................................................................................................................. 1  
   1.2 Goal .............................................................................................................................. 1  
   1.3 Goal criteria .................................................................................................................. 1  
   1.4 Focus and limitations ................................................................................................. 2  
   1.5 Outline of report ......................................................................................................... 2  
2 Related work ..................................................................................................................... 3  
3 Architecture description .................................................................................................. 5  
   3.1 System architecture ..................................................................................................... 5  
   3.2 Control Software architecture ...................................................................................... 6  
      3.2.1 “4+1” View Model description ............................................................................... 6  
      3.2.2 Logical view ......................................................................................................... 7  
      3.2.3 Development view ............................................................................................... 7  
      3.2.4 Physical view ...................................................................................................... 11  
      3.2.5 Process view ........................................................................................................ 11  
      3.2.6 Scenarios ............................................................................................................ 12  
   Summary .......................................................................................................................... 13  
4 Implementation description ............................................................................................ 14  
   4.1 Java implementation .................................................................................................. 14  
      4.1.1. Process notification mechanism ........................................................................ 14  
      4.1.2 Region lock implementation ............................................................................... 16  
   4.2 Structured Text implementation .................................................................................. 17  
   Summary .......................................................................................................................... 19  
5 Communication protocol description ............................................................................ 20  
   5.1 Protocol messages ..................................................................................................... 20  
   5.2 Communication scenario ........................................................................................... 21  
   Summary .......................................................................................................................... 23  
6 Results ................................................................................................................................ 24  
7 Conclusions and Future work ....................................................................................... 27  
References .......................................................................................................................... 28  
Appendix A – Control Software installation instructions ....................................................... 29  
Appendix B – Semantically equivalent code comparison ....................................................... 30
1 Introduction

In this chapter a small introduction covering what is control software, why it is so important and how to test it is given. Goal and goal criteria are also defined in this chapter as well as the limitations of this work. Finally, the outline of the report, describing the content of this thesis is given.

1.1 Background

Control software is a type of software that monitors and controls industrial processes that exist in the real physical world. Nowadays control software plays an important role in industrial automation, it is typically used in industries such as oil, electrical and water. Control software sends control commands to the remote stations as a response to the information fetched from them.

Several high-level, domain-specific languages enhanced for implementing control software were defined and standardized. One of these languages is Structured Text (ST) programming language [1], which is supported by the IEC 61131-3 standard and is widely used for programmable logic controllers (PLCs).

The Company (ARiSA AB) implements ST to C compiler, which is used by the Customer (Sauer-Danfoss AB) to write control software for industrial machines at their factories. It is extremely important to make sure that the compiler is valid before starting to use it. We suggest validating the compiler with a set of good test programs. But what is a good test program? We will assume that a good test program for compiler validation is one that is as close as possible to the real world program, which has to be processed by this compiler. In our case a good test program will be a piece of control software that performs the control of industrial machines on Customer’s factories. But unfortunately the Company does not have access to the Customer’s control software and machines. In this case a smart move will be to implement the simulation of the whole system: the simulation model of machines and the control software performing the control over that simulation model.

1.2 Goal

The goal of this master thesis is to design and implement control software for a simulation model of production cells, which can be used as Customer’s production cell simulation. Two key objectives have been set:

1. Test the correctness of ST to C compiler using the implemented control software as test case.
2. Market Company as a modern and highly technologic compiler construction company.

1.3 Goal criteria

The main focus of this work is testing the correctness of ST to C compiler, which can be achieved by comparing the behavior of semantically equivalent control software written in ST and Java. In order to evaluate this goal, the following goal criteria were set:

a. The control software implemented in ST language should produce the same output (the sequence of model states) as its Java analogue on the equivalent simulation scenarios.

b. Control software should not hamper simulation flow.

c. Easy to design new control scenarios.
d. The secondary, but still very important goal of this project is the following: the results of this work must be shown to the existing and potential customers of the Company. The next goal criterion was set to evaluate this goal: the resulting control software must work with the simulation model, which is displayed on the web page.

1.4 Focus and limitations

The study is focusing on a real time production cell case study developed in the Forschungszentrum Informatik (FZI), Karlsruhe, Germany [2]. The real time production cell case study (see Figure 1.1) is composed of two conveyor belts (a feed belt and a deposit belt), a barcode reader, which is located at the picking position of the feed belt, four processing units and two portal cranes.

![Figure 1.1. FZI Real Time Production Cell top view](image)

The procedure of item processing takes the following steps. Item is inserted at the beginning of the feed belt and is conveyed to the front of the barcode reader. When the barcode reader notifies the item, the feed belt is stopped and Crane1 is notified, and it picks up this item and places it on a free processing unit (workspace). When the processing unit finishes processing the item, the second portal crane is notified. As a response to this notification, it takes the item from workspace and brings it to the deposit belt. In general, every processing unit has its own processing time; the moving speed of portal cranes and belts also differs.

1.5 Outline of report

The second chapter discusses existing researches in this field and outlines the case study used in this work. In the third chapter, the general overview of the whole system is presented together with a detailed architecture project of Control Software, which is described using “4+1” View Model. The fourth chapter elaborates on the implementation details of Control Software. It considers both Java and Structured Text implementations. In the fifth chapter, the communication protocol between Control Software and Simulation Software is described in detail. The conclusion regarding goal and goal criteria evaluation is given in the sixth chapter. The final results of the work are presented and discussed and some future directions and recommendations for improvement are mentioned in the seventh chapter.
2 Related work

This chapter describes related work in the field of production cell. It mentions two major production cell case studies, describes them and gives comments on the goals of both case studies and related researches.

“Production Cell” case studies
Two major “Production Cell” case studies were developed: FZI Production Cell and Production Cell with Focus. Both of them have a common primary goal to show the usefulness of formal methods in software development and to prove their applicability to real-world cases. Both case studies are easy to understand but in the same time they are not trivial, which allows applying different formal methods of software development on them. Additionally, the example for both case studies was taken from a real metal processing plant near Karlsruhe. This fact certainly increases the practical value of both case studies. As noticed before, this case study is used in this work as a test scenario.

FZI production cell
FZI Production Cell functioning principle is shortly described in section 1.4. More detailed description is presented in the technical report of A. Lötzbeyer [2]. This case study is used in a number of researches, such as in the paper of Zorzo [3], which has a goal to provide a mechanism to deal with concurrent exceptions.

Production cell with Focus
The Production Cell with Focus case study is originally developed as one of two major case studies of the KORSO project, aiming to design an improved software development technology that is based on a practical use of the formal foundations of correct software development [4].

The production cell consists of two conveyor belts, a positioning table, a two-armed robot, a press, and a travelling crane (see Figure 2).

Figure 2.1. FZI Real Time Production Cell top view
Metal plates are inserted at the feed belt and then are moved to the press, where they are forged. Afterwards they are brought out of the production cell via the travelling crane and the deposit belt. More detailed description of this case study can be viewed in a book of Lindner and Lewerentz [5].

This case study is also widely used in number of researches, such as in the paper of Lewerentz [6], which aims to give the evaluation of applied methods by measuring the quality of the resulting software.

**Key differences**

As a conclusion of related work overview we should emphasize, that both case studies and related researches are aimed to investigate and apply different formal software development methods for improving the quality of the resulting software. The goal criteria of this research are different, since we are focusing on verification of functional requirements of resulting control software. This means that the problem we are aimed to solve is unique and there is no ready-made solution or a framework for solving it.
3 Architecture description

In this chapter the general overview of the whole system is given and an architecture project of Control Software is described using the “4+1” View Model [7].

3.1 System architecture

The architecture of the system is built by Three-tier software architecture pattern [8]. This is a classic client-server model, where an additional component to separate logic and communication handling was introduced on the server side. Figure 3.1 illustrates the general architecture of the whole system.

![Three-tier architecture of the system](image)

Figure 3.1. Three-tier architecture of the system

This architecture has the following advantages:

- Server (Control Software and Communication component) can work with multiple instances of Simulation Software (clients), which can be run on different machines.
- The logic of the server does not depend on communication protocol between server and client.
- Any of these tires can be independently replaced or upgraded in response to requirement or technology changes.

One important drawback of this architecture is that all the computations are transferred to the server, which makes server a bottleneck of the system. However, several tactics can eliminate this drawback, for example: transferring a certain small amount of computations to clients or introducing redundant backup servers.

The server is concurrent and offers on-demand thread (in case of Java implementation) and process (in ST implementation) spawning. Additionally, I should mention that client and server communicate via socket communication channel, which makes it possible to access server from any machine in the network. Communication protocol between client and server is described in more details in Chapter 5.

Having described architecture of the whole system, it is possible to describe separately the architecture of Control Software.
3.2 Control Software architecture

In this section the Control Software architecture is outlined using generally recognized “4+1” View Model. The architecture is described from four concurrent views of the model and use case scenarios are used to illustrate the architecture design.

3.2.1 “4+1” View Model description[7]

The ”4+1” View Model is aimed to describe the architecture of the software system using four concurrent views and scenarios or use cases (the fifth view). The four views of this model are:

- **Logical view** – describes logical organization of the software system components. It is focused on showing the functionality which software system provides to the users. Usually described using UML Class, Component and/or Sequence diagrams.

- **Development view** – describes software systems from the software developer’s point of view. Serves as a technical documentation of the software system design. Usually described using UML Class and/or Component diagram.

- **Physical View** – describes software system from the system engineer’s point of view. Gives detailed description of the software system components physical organization, deployment and topology as well as connections between them. Usually described using UML Deployment diagram.

- **Process View** – describes the dynamic side of the software system: system processes and communication between them. Usually described using UML Activity diagram.

The *Scenarios* give the description of interaction between software components. As a result, they help to identify key elements of the software architecture and serve as a basis for test cases.

Figure 3.2 illustrates the outline of “4+1” View Model.

![Figure 3.2. “4+1” View Model illustration](image)

The benefit of using this model is that it is described from the viewpoint of every stakeholder interested in this software system: end-users, software engineers and project
managers. This allows mitigating the risks associated with software architecture and simplifies software system architecture understanding.

3.2.2 Logical view
From a logical point of view the control software consists of six logical components: Processes, Events, Control, Simulation Interface, Device Manager and Region Lock. These components are connected between each other (see Figure 3.3.) and have their own responsibilities and functions.

![Diagram of logical view of Control Software architecture](image)

**Figure 3.3. Logical view of Control Software architecture**

*Control* – the purpose of this component is to process the incoming message from Communication Component. Depending on the message, this component may launch or stop the execution of some processes or it may perform some changes in the state of devices, using Device Manager component.

*Device Manager* – performs changes in simulation devices by asking Simulation Interface to send appropriate messages to the actual Simulation Software (client). It is also responsible for dealing with region locks, which are used to speed up the entire working process. Region lock and their implementation are discussed in details in Chapter 4.

*Processes* – this component consists of a number of different processes (e.g. “Feed Belt Process”), which can be started by Control component and updated by a different kind of events (e.g. “Item In Picking Position”). The purpose of processes is to simplify the control over belts and portal cranes. Processes may generate events.

*Events* – is the component, which contains a bunch of events that may happen during the simulation.

*Region Lock* – provides functionality to implement region locks in the workflow simulation process.

*Simulation Interface* – is responsible for providing Communication Component with control messages, which have to be sent to Simulation Software.

All these components are designed as separate entities of the system (classes or packages). The detailed overview of these entities is presented in the “Development View” section.

3.2.3 Development view
From the development point of view the Control Software consists of seven packages. The package diagram of Control Software component is presented on Figure 3.4.
Below each package is described in detail.

Package simulation consists of one class. This package acts like an adaptor between Control Software and Communication Component. Its aim is to hide the complexity of message construction by providing the common interface.

Class SimulationInterface provides a generic interface of Simulation Software to Control Software. Class contains 13 methods.

- Method startBelt() uses Communication Component to send the message to Simulation Software for starting its feed belt.
- Method stopBelt() uses Communication Component to send the message to Simulation Software for stopping its feed belt.
- Method firstPortalUp() uses Communication Component to send the message to Simulation Software for lifting its first portal.
- Method firstPortalDown() uses Communication Component to send the message to Simulation Software for taking down its first portal.
- Method secondPortalUp() uses Communication Component to send the message to Simulation Software for lifting its second portal.
- Method secondPortalDown() uses Communication Component to send the message to Simulation Software for lowering its second portal.
- Method firstPortalMagnetOn() uses Communication Component to send the message to Simulation Software for turning on the magnet on its first portal.
- Method firstPortalMagnetOff() uses Communication Component to send the message to Simulation Software for turning off the magnet on its first portal.
- Method secondPortalMagnetOn() uses Communication Component to send the message to Simulation Software for turning on the magnet on its second portal.
- Method secondPortalMagnetOff() uses Communication Component to send the message to Simulation Software for turning off the magnet on its second portal.
- Method moveFirstPortalTo(Position) uses Communication Component to send the message to Simulation Software for moving its first portal to a desired position. Method takes one parameter – a target position of the first portal.

Figure 3.4. Control Software package diagram
- Method `moveSecondPortalTo(Position)` uses Communication Component to send the message to Simulation Software for moving its second portal to a desired position. Method takes one parameter – a target position of the second portal.

- Method `setWorkspaceActive(Integer, Boolean)` uses Communication Component to send the message to Simulation Software for starting or stopping the work at a given workspace. Method takes two parameters. The first parameter is the number of the workspace, an integer value. The second parameter indicates whether the workspace should start or stop processing the item (a Boolean value).

Package `shared` is aimed to contain all the common data, which is used by Control Software and Communication Component.

Class `Messages` contains the definition of all the messages used in the communication process.

Package `logging` contains one class.

Class `ProductionLogger` is responsible for logging the messages. Class contains one method.

- Method `log(String)` performs actual logging of the given message. Method has 1 string parameter – the message to be logged.

Package `control` contains two classes: `ControlComponent` and `DeviceManager`. The aim of the classes of this package is to implement the actual control over the Control Software.

Class `ControlComponent` contains the logic of the control process. The class has one method.

- Method `processMessage(String)` processes the message received from Simulation Software and acts accordingly to it.

Class `DeviceManager` executes major operations with devices (e.g. picking items from workspaces). The class has four methods.

- Method `pickItemFromFeedBelt()` calls a sequence of methods in SimulationInterface class needed for picking item from a feed belt.

- Method `pickItemFromWorkspace(Integer)` calls a sequence of methods in SimulationInterface class needed for picking item from a given workspace. Method has one integer parameter – the identifier of the workspace, from which the item has to be picked.

- Method `placeItemOnTrayBelt()` calls a sequence of methods in SimulationInterface class needed for placing the picked item on the deposit belt.

- Method `placeItemOnFreeWorkspace()` calls a sequence of methods in SimulationInterface class needed for placing the picked item on a free workspace.

Package `control.locks` contains one class responsible for providing the region lock mechanism for portals.

Class `RegionLock` provides region locks for the portals. Class contains two methods.

- Method `release()` releases all previously acquired locks.

- Method `acquire(Integer)` acquires a lock for a specified region. Method has one integer parameter – the identifier of the region to be locked.
Package `control.events` contains descriptions of all the events, which may occur during simulation. Package contains seven classes.

Class `Event` is an abstract class that generalizes all the system events, which may happen during the workflow.

Class `WorkspaceEvent` is an abstract class that generalizes the events, which may happen on the workspace. The class contains two methods.
- Method `setWorkspace(Integer)` associates this event with the workspace, where this event has happened. Method has one integer parameter – a workspace identifier, where this event has happened.
- Method `getWorkspace()` returns the identifier of the workspace, where this event has happened. Method returns Integer value - a workspace identifier.

Class `ItemProcessedAtWorkspace` represents the event that happens when any of the workspaces finishes processing the item. The class inherits from `WorkspaceEvent` class.

Class `ItemReleasedAtWorkspace` represents the event that happens when the second portal picks up the item from any of the workspaces. The class inherits from `WorkspaceEvent` class.

Class `ItemPlacedOnWorkspace` represents the event that happens when the first portal places the item for processing on any of the workspaces. The class inherits from `WorkspaceEvent` class.

Class `ItemInPickingPosition` represents the event that happens when the item appears in the picking position on the feed belt, i.e. ready to be picked by one of the portals.

Class `ItemPickedFromFeedBelt` represents the event that happens when the item is picked from the feed belt by the first portal.

Package `control.processes` is designed to assist in the logic of control software. The package consists of six process classes, which are aimed to encapsulate the logic of a concrete process, which may happen during workflow execution.

Class `Process` represents an abstract process, which can be run in the system. The class is abstract.

Class `PortalProcess` represents an abstract process, which may occur in the work of the first and the second portal. The class is abstract and inherits the `control.processes.Process` class. This class contains one field and six methods
- Field `queue_size` has type Integer and represents the number of request the portal can store in its buffer. All the requests are executed in FIFO (first-in-first-out) order.
- Method `moveToWorkspace(Integer)` emulates the movement of the portal to a specified workspace. Method has one integer parameter – a workspace identifier.
- Method `moveToRegion(Integer)` emulates the movement of the portal to a specified region. Method has one integer parameter – region identifier.
- Method `up()` emulates the lift of the magnet on the portal.
- Method `down()` emulates the lowering of the magnet on the portal.
- Method `magnetOn()` emulates turning on the magnet of the portal.
- Method `magnetOff()` emulates turning off the magnet of the portal.

Classes `FirstPortalProcess` and `SecondPortalProcess` emulate the processes of first and the second portal respectively.

Class `TrayBeltProcess` represents the process of deposit belt. The class has two methods.
- Method `startBelt()` emulates the start of deposit belt.
- Method `stopBelt()` emulates the stop of deposit belt.
Class *FeedBeltProcess* represents the process of feed belt. The class has two methods.
- Method *startBelt()* emulates the start of feed belt.
- Method *stopBelt()* emulates the stop of feed belt.

### 3.2.4 Physical view

The physical view of the model describes the hardware configuration to which the software is mapped. In our case the physical view is simple: the Control Software and all its processes run on the same node, i.e. on the same machine. The hardware configuration may vary according to the architectural tactics used in order to satisfy non-functional requirement changes. But this topic is out of the scope of this work. We just note that having running server and all its components on one machine makes it easier to add new features and test server functionality.

### 3.2.5 Process view

The architecture of the server from a process view is described using activity diagram (Figure 3.5).

![Activity diagram](image)

**Figure 3.5. Control Software activity diagram**

The diagram from Figure 3.4 illustrates the process of server initialization when a new client (simulation software) makes a request to use it. This request is made by SIMULATION_MESSAGE sent to the Control Software. The message triggers the process of server initialization. First, the server generates id for a new client and
initializes *DeviceManager* component. Afterwards, all server processes are started. After that server initializes *SimulationInterface*, and after being initialized, sends SIMULATION_START message and new id to the client.

### 3.2.6 Scenarios

In this chapter the description of the architecture is illustrated using two scenarios. The first scenario describes how the item is delivered to the workspace in order to be processed. The second scenario focuses on the rest part of the workflow process, specifically, it describes how the item is picked from the workspace and delivered to the deposit belt.

**Scenario 1: delivering items to the workspaces**

Scenario triggers when *ControlComponent* receives message that item appears on picking position on the feed belt. In this case the task to pick the item from feed belt is added to the task queue of the first portal. When the task is picked from the queue, the following steps will be executed:

1. The feed belt stops. The first portal releases all acquired locks and moves to the feed belt.
2. Having arrived to the feed belt, the portal picks the item from it. When the item is picked, the belt continues moving.
3. The first portal searches the nearest free workspace, where the item can be loaded.
   4.a. If the target workspace has id 1 or 2, the first portal acquires a lock to the first region.
   4.b. If the target workspace has id 3 or 4, the first portal acquires a lock to the first region. When the lock is acquired, portal moves to the first region. When the portal arrives to the first region, it acquires the lock to the second region.
   4.c. Else there is no free workspaces at the moment. The first portal waits for a certain amount of time and repeats step 3.
4. When a lock to a desired workspace region is acquired, the portal moves to the workspace.
5. Having arrived to the workspace, the first portal releases the item. When the item is released, the portal releases all acquired locks, starts moving to the feed belt and picks the next task from its task queue. The workspace starts processing the item.

**Scenario 2: picking processed items and delivering them to the feed belt**

Scenario triggers when *ControlComponent* receives message that item is processed on one of the workspaces. In this case the task to pick the item from the workspace is added to the task queue of the second portal. When the task is picked from the queue, the following steps will be executed:

1. The second portal acquires the lock to the second region.
2. When the lock to the second region is acquired:
   2.a. If the id of the destination workspace is 3 or 4, the second portal moves to the workspace.
   2.b. If the id of the destination workspace is 1 or 2, the second portal acquires the lock to the first region. When the lock to the first region is acquired, the second portal moves to the destination workspace.
3. When the second portal arrives to the workspace, it picks up the item from it, releases all acquired locks and moves to the deposit belt.
4. Having arrived to the deposit belt, the second portal places item on it and takes next task from its task queue.
Both scenarios give a high level description of the workflow process. Its detailed description presented in the 5th chapter of this thesis.

Summary
In this chapter the general architecture of the whole system and the architectural overview of the Control Software were given. The architecture of the Control Software was described using widely-used “4+1” View Model. The architecture was described from 4 different views: logical, development, physical and process views. In addition, the descriptions of two important scenarios of the system behavior were given.

The architecture of the software system provides a solid base for system construction. The next section describes the following step in the system development: implementation details.
4 Implementation description

In this chapter the most important issues of the Control Software component implementation are given. The Control Software component was implemented in two languages: Java and Structured Text. Java was chosen because it provides all the necessary tools for control software implementation.

4.1 Java implementation

Implementing Control Software component in Java, I have used object-oriented approach [9]. Usage of this approach gave the following advantages:

- The Control Software component was introduced as a number of minor components with simple interfaces, making the system easy to understand.
- The complexity of development and maintenance processes was significantly reduced.
- The usage of object oriented principles like inheritance and polymorphism allowed avoiding code duplication.

In this section two the most interesting implementation details are outlined. The first one is about designing the process notification mechanism. The second one describes region lock implementation and its usage.

4.1.1. Process notification mechanism

During the implementation I have faced the following issue: the processes of Control Software component must notify each other about changes they do in the system. For example the workspace process notifies others about the processed item. To deal with this issue I have chosen to use Observer behavioral pattern of the “Gang of Four” [10]. This pattern allows a number of observers to see the published event. Figure 4.1 demonstrates the class diagram of Observer pattern usage in the system.

```
<<interface>>
Observable
+registerObserver(Observer): void
+removeObserver(Observer): void
+notifyObservers(Event): void

<<interface>>
Observer
+update(Event): void

StateMachine
-listOfObservers: Observer[]
```

Figure 4.1. Observer pattern usage

The key entity in this pattern is a StateManager class, which contains list of observers and provides methods to:
- subscribe observer: registerObserver(Observer),
- unsubscribe observer: removeObserver(Observer),
- notify observers with event: notifyObservers(Event).
StateManager implements Observable interface by implementing its methods. The implementation of StateManager class is presented in Listing 4.1.

Listing 4.1. StateManager.java
```java
package control.processes;
import java.util.ArrayList;
import java.util.List;
import control.processes.events.Event;
public class StateManager extends Observable {
    // observers collection
    private List<Observer> observers = new ArrayList<Observer>();
    /****** Observer-specific methods ******/
    public void registerObserver(Observer o) {
        observers.add(o);
    }
    public void removeObserver(Observer o) {
        int i = observers.indexOf(o);
        if (i >= 0)
            observers.remove(i);
    }
    public void notifyObservers(Event event) {
        for (Observer obs : observers)
            obs.update(event);
    }
}
```

Class Observer provides update method that updates a concrete observer according to this event. Abstract class Process inherits this interface and update method is implemented in its concrete successor process classes (e.g. WorkspaceProcess). Listing 4.2 shows the example of update method implementation in the control.processes.FirstPortalProcess class.

Listing 4.2. FirstPortalProcess.java
```java
package control.processes;
import java.util.Queue;
import java.util.concurrent.ArrayBlockingQueue;
import control.ControlComponent;
import control.processes.events.Event;
import control.processes.events.ItemInPickingPositionEvent;
import control.processes.events.ItemPickedFromFeedBelt;
...
public class FirstPortalProcess extends PortalProcess {
    private Queue<Runnable> tasks = new ArrayBlockingQueue<Runnable>(TASK_STACK_SIZE);
    ...
    @Override
    public void update(Event event) {
        if (event instanceof ItemInPickingPositionEvent) {
            tasks.offer(pickItemAtFeedBeltTask);
        } else if (event instanceof ItemPickedFromFeedBelt) {
            tasks.offer(movePortalToWorkSpace);
        }
    }
    ...
}
```

Listing 4.3 shows how the StateManager class can be used in order to notify all the processes with specific event.

Listing 4.3...
Listing 4.3. StateManager usage example
StateManager stateManager = new StateManager();
Process firstPortalProcess = new FirstPortalProcess();
stateManager.registerObserver(fpp);
// register other processes as observers
...
// notify all observers:
stateManager.notifyObservers(new ItemInPickingPosition());

This implementation enables usage simplicity and dynamic subscription, deletion and notification of the observers.

4.1.2 Region lock implementation
This section describes the implementation of thread-safe region lock. Region locks are implemented in control.locks.RegionLock class. Listing 4.4 shows the implementation of this class. This implementation uses double-checked locking mechanism to implement thread-safe region lock acquisition.

Listing 4.4. RegionLock.java
package control.processes.locks;
public class RegionLock {
    private int lock = 0;// represents a lock of the region:
    // 0 - not locked; 1 - locked by 1st portal crane; 2 - locked by 2nd
    // portal crane
    /**
     * Acquire a lock for this region
     * @param craneNumber number of the crane (1 or 2)
     * @return true if lock was successfully acquired false otherwise
     */
    public boolean acquire(int craneNumber){
        if(lock == 0){
            synchronized(RegionLock.class){
                if(lock == 0){
                    lock = craneNumber;
                    return true;
                } else {
                    return false;
                }
            }
        }
        return false;
    }
    /**
     * Releases the lock of the region
     */
    public void release(){
        lock = 0;
    }
}

RegionLock class represents the lock for one of the regions. In order to control the locks for both regions, it is required to have two instances of the RegionLock class. Listing 4.5 shows the usage of RegionLock class.

Listing 4.5. RegionLock usage example
public class DeviceManager { ...
    private RegionLock region1 = new RegionLock(); // the first region with
    // workspaces 1 and 2
    private RegionLock region2 = new RegionLock(); // the second region with
    // workspaces 3 and 4
The example from Listing 4.5 how the first portal moves to the 4th workspace. First, it acquires lock to the first region (method moveFirstPortalToRegionOne), moves there when the lock is acquired and then acquires a lock to the second region (method moveFirstPortalToFourthWorkspace) and moves to the fourth workspace when the lock is acquired.

In order to acquire lock to the region, one must use appropriate RegionLock instance (region1 or region2 in Listing 4.5) and pass workspace identifier to acquire method.

4.2 Structured Text implementation

In Structured Text (ST) implementation of the Control Software component I have used a structured programming approach, where the whole ST program is divided in a number of function blocks (subroutines) that perform its own specific tasks. For example, in this section the example for the function block is shown, which is responsible for processing ‘FIRST_PORTAL_IS_DOWN’ message.

Listing 4.6. ST implementation of ‘FIRST_PORTAL_IS_DOWN’ message processing

```
FUNCTION_BLOCK PROCESS_FP_IS_DOWN
  VAR_INPUT
    MESSAGE: STRING;
  END_VAR
  CLIENT_ID := LEFT(MESSAGE, 1);
  IF(FIRST_PORTAL_LOADED <> TRUE) THEN
    MESSAGE_FOR_CLIENT := CONCAT(CLIENT_ID, ':', 'FIRST_PORTAL_TURN_ON_MAGNET');
    PRINT(MESSAGE_FOR_CLIENT);
    MESSAGE_FOR_CLIENT := CONCAT(CLIENT_ID, ':', 'FIRST_PORTAL_UP');
    PRINT(MESSAGE_FOR_CLIENT);
    FIRST_PORTAL_LOADED := TRUE;
  ELSE
    MESSAGE_FOR_CLIENT := CONCAT(CLIENT_ID, ':', 'FIRST_PORTAL_TURN_OFF_MAGNET');
    PRINT(MESSAGE_FOR_CLIENT);
    MESSAGE_FOR_CLIENT := CONCAT(CLIENT_ID, ':', 'FIRST_PORTAL_UP');
    PRINT(MESSAGE_FOR_CLIENT);
    FIRST_PORTAL_LOADED := FALSE;
  END_IF
END_FUNCTION_BLOCK
```
In the same procedural way the rest of the ST Control Software was implemented. Note that non-standard PRINT function for sending messages to clients from ST code is used.

Besides, in order to accept connections from clients, a small piece of server software was written in C. This component accepts connections from clients, forks a new process for a new client and delegates this process to serve the client. Listing 4.7 shows the most important implementation details of this component.

Listing 4.7. Connection listener component written in C

```c
int main(int argc, char *argv[]){
    int sockfd, newsockfd, portno, pid;
    socklen_t cli_len;
    struct sockaddr_in serv_addr, cli_addr;
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if ( sockfd < 0 )
        error("ERROR opening socket");
    portno = 8190;
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(portno);
    if (bind(sockfd, (struct sockaddr *) &serv_addr,
        sizeof(serv_addr)) < 0)
        error("ERROR on binding");
    listen(sockfd,5);
    cli_len = sizeof(cli_addr);
    while (1) {
        newsockfd = accept(sockfd,
            (struct sockaddr *) &cli_addr, &cli_len);
        if ( newsockfd < 0 )
            error("ERROR on accept");
        pid = fork(); //fork to process multiple connections
        if ( pid < 0 )
            error("ERROR on fork");
        if ( pid == 0 ) {
            close(sockfd);
            processRequests(newsockfd); //process requests from client
            exit(0);
        } else close(newsockfd);
    }
    close(sockfd);
    return 0;
}
```

When a new connection is established in ‘main’ function, a new process is forked, in which the processRequests function is called. This function receives messages from its client and calls ST function fb__PROCESS_MESSAGE_Body to process them. Listing 4.8 shows the implementation of processRequests function.
Listing 4.8. *processRequests* function implementation

```c
void processRequests (int sock){
    int n;
    char buffer[256];
    uint16_t *__ERR = 0u;
    bzero(buffer,256);
    while(1){
        n = read(sock,buffer,255);
        if (n < 0) error("ERROR reading from socket");
        struct fb__PROCESS_MESSAGE param;
        struct fb__PROCESS_MESSAGE *pointerParam = &param;
        struct __string s; struct __string st;
        struct __string* strBuf = &s;
        struct __string* strSim = &st;
        strcpy(strBuf->value, buffer);
        strcpy(strSim->value, SIMULATION_INITIALIZED);

        pointerParam->__ST__MESSAGE = s;
        pointerParam->__ST__SIMULATION_INITIALIZED = st;
        fb__PROCESS_MESSAGE_Body(__ERR, pointerParam);
    }
}
```

To respond to clients with messages, ST control software sends message to the named pipe, where it is read by reused Java message sender and redirected to the client.

**Summary**

In this section the descriptions of Java and Structured Text implementations of the Control Software were given. In Java implementation the object-oriented approach was used, which gave certain advantages. Structured Text implementation of Control Software was implemented in procedural manner. Besides, the auxiliary C code was added to the native ST implementation to provide necessary functionality. The next section describes the details of communication protocol between Control Software and Simulation Software.
5 Communication protocol description

In this chapter messages used in communication between Control Software and Simulation Software are described. Besides, the typical scenario of client-server communication is introduced. For shortness, we will call Control Software as CS and Simulation Software as SS.

5.1 Protocol messages

The structure of the messages is presented in figure 5.1. This structure applies to all the messages described later, except for SIMULATION_INITIALIZED, because at the time when this message is sent, there is no client id generated for the client. Messages consist of three parts: client identifier (for distinguishing different clients), command (to represent the meaning of the message) and a separator (to split client id value from the actual command). Technically, client id is an 8-character length string, which is generated on the server when SIMULATION_INITIALIZED message is received.

![Message Structure Diagram](image)

The description of message commands, which can be sent by Simulation Software and received by Control Software are listed below:

- SIMULATION_INITIALIZED – notifies CS that the simulation is ready to start.
- SIMULATION_STOPPED - notifies CS that the simulation is stopped.
- ITEM_IN_PICKING_POSITION – notifies CS that the item is appeared in picking position on the feed belt.
- ITEM_PICKED_FROM_FEED_BELT - notifies CS that the item was picked from feed belt by the first portal.
- ITEM_PLACED_ON_WORKSPACE – notifies CS that the item is placed on the workspace.
- ITEM_PROCESSED_AT_WORKSPACE N – notifies CS that the item is processed on the workspace, where N is the workspace identifier (positive integer from 1 to 4).
- FIRST_PORTAL_IS_UP – notifies CS that the first portal has risen up.
- SECOND_PORTAL_IS_UP – notifies CS that the second portal has risen up.
- FIRST_PORTAL_IS_DOWN – notifies CS that the first portal went down.
- SECOND_PORTAL_IS_DOWN – notifies CS that the second portal went down.
- FIRST_PORTAL_AT_FEED_BELT – notifies CS that the first portal arrived to the feed belt.
- FIRST_PORTAL_ARRIVED_AT_FREE_WORKSPACE – notifies CS that the first portal arrived to a free workspace.
- SECOND_PORTAL_AT_TRAY_BELT – notifies CS that the second portal arrived to the deposit belt.
- SECOND_PORTAL_ARRIVED_AT_FREE_WORKSPACE – notifies CS that the second portal arrived to the workspace with processed item.
The description of message commands, which can be sent by Control Software and received by Simulation Software are listed below:

- **SIMULATION_START** – requests SS to start the simulation process.
- **STOP_FEED_BELT** – requests SS to stop the feed belt.
- **START_FEED_BELT** – requests SS to start the feed belt.
- **ITEM_INSERT** – requests SS to insert the item of the feed belt.
- **MOVE_FIRST_PORTAL_TO_FEED_BELT** – requests SS to move the first portal to the feed belt.
- **MOVE_SECOND_PORTAL_TO_TRAY_BELT** – requests SS to move the second portal to the feed belt.
- **FIRST_PORTAL_MOVE_TO_WORKSPACE_N** – requests SS to move the first portal to the specified workspace, where N is the workspace identifier (positive integer from 1 to 4).
- **SECOND_PORTAL_MOVE_TO_WORKSPACE_N** - requests SS to move the second portal to the specified workspace, where N is the workspace identifier (positive integer from 1 to 4).
- **FIRST_PORTAL_MOVE_TO_REGION_1** – requests SS to move the first portal to the first region.
- **SECOND_PORTAL_MOVE_TO_REGION_2** – requests SS to move the second portal to the second region.
- **FIRST_PORTAL_UP** – requests SS to rise up the magnet on the first portal.
- **FIRST_PORTAL_DOWN** – requests SS to lower the magnet on the first portal.
- **SECOND_PORTAL_UP** – requests SS to rise up the magnet on the second portal.
- **SECOND_PORTAL_DOWN** – requests SS to lower the magnet on the second portal.
- **FIRST_PORTAL_TURN_ON_MAGNET** – requests SS to turn on the magnet on the first portal crane.
- **FIRST_PORTAL_TURN_OFF_MAGNET** – requests SS to turn off the magnet on the first portal crane.
- **SECOND_PORTAL_TURN_ON_MAGNET** – requests SS to turn on the magnet on the second portal crane.
- **SECOND_PORTAL_TURN_OFF_MAGNET** – requests SS to turn off the magnet on the second portal crane.

In this section the definitions of the messages used in the communication are outlined. The next section shows how these messages are used in a typical working scenario.

### 5.2. Communication scenario

The communication scenario description is outlined below:

**Initialization phase:**
In order to start simulation, SS sends SIMULATION_INITIALIZED message to CS, which means that the simulation is ready to receive control messages. In response CS sends SIMULATION_START message to SS, on which SS turns on both belts and starts generating items.

**Working phase:**
When the item appears to be in picking position SS notifies CS by sending ITEM_IN_PICKING_POSITION message. In response CS sends FEED_BELT_STOP message, which forces SS to stop the feed belt. Afterwards CS sends MOVE_FIRST_PORTAL_TO_FEED_BELT message. Having received this message
SS sends the first portal to the feed belt. When the portal is finally relocated to the feed belt, the message **FIRST_PORTAL_AT_FEED_BELT** is sent to CS, this means that the portal is at the feed belt and is ready to pick item.

CS sends **FIRST_PORTAL_TURN_OFF_MAGNET** and then **FIRST_PORTAL_DOWN** message to SS. SS forces first portal to go down and notifies CS that the portal is down by **FIRST_PORTAL_IS_DOWN** message. CS turns portal magnet on by sending **FIRST_PORTAL_TURN_ON_MAGNET**. SS receives this message and turns the magnet on. Meanwhile CS sends a command to lift the first portal sending **FIRST_PORTAL_UP** message to SS. SS forces first portal to go up and notifies CS that the portal is up by **FIRST_PORTAL_IS_UP** message.

When at least one workspace is free, CS sends a **FIRST_PORTAL_MOVE_TO_WORKSPACE_N**, where N is either 1, 2, 3 or 4, indicating the identifier of the vacant workspace which was selected by CS. Having received this message SS sends the first portal to the desired workspace and reports with **FIRST_PORTAL_ARRIVED_AT_FREE_WORKSPACE** message to CS. CS responds on this message with **FIRST_PORTAL_DOWN** message sent to SS. SS forces first portal to go down and notifies CS that the portal is down by **FIRST_PORTAL_IS_DOWN** message. CS turns portal magnet off by sending **FIRST_PORTAL_TURN_OFF_MAGNET**. SS receives this message and turns the magnet off. Meanwhile CS sends a command to lift the first portal sending **FIRST_PORTAL_UP** message to SS. SS forces first portal to go up and notifies CS that the portal is up by **FIRST_PORTAL_IS_UP** message.

CS sends **WORKSPACE_N_START_PROCESSING** message after receiving **FIRST_PORTAL_UP** message from SS. SS starts to process the item on a workspace. When the item is processed on the workspace, SS sends **ITEM_PROCESSED_AT_WORKSPACE_N** message to CS. Having received this message, CS sends **SECOND_PORTAL_MOVE_TO_WORKSPACE_N**, where N is either 1, 2, 3 or 4, indicating the identifier of the workspace which has just processed the item. Having received this message, SS sends the second portal to the desired workspace and reports with **SECOND_PORTAL_ARRIVED_AT_FREE_WORKSPACE** to CS. Note, that CS may send **SECOND_PORTAL_MOVE_TO_REGION_2** in case if the second portal has to be sent to the workspace 1 or 2 and the first region is blocked. The same happens with **FIRST_PORTAL_MOVE_TO_REGION_1**, which is sent when the first portal has to be sent to workspace 3 or 4 and the second region is blocked.

When the first portal reaches the first region SS sends **FIRST_PORTAL_AT_REGION_1** message. When the 2nd portal reaches the second region SS sends **SECOND_PORTAL_AT_REGION_2** message.

When the simulation on the client side is finished (e.g. stopped by user), the **SIMULATION_STOPPED** is sent to CS. In response to this message, CS releases all the resources that were allocated for this client and removes it from the list of clients.

Communication scenario example is illustrated in Figure 5.2.
Figure 5.2. Communication scenario example

Summary
In this section the communication protocol between Control Software and Simulation Software was described. First, the list of used messages together with their purposes was given. Afterwards, the communication scenario (rules) was introduced. The typical scenario consists of two basic phases: initialization phase and working phase. In initialization phase the connection is established between Control Software as a server and Simulation Software as a client. In the working phase both parties interact with each other using messages described in this chapter.

This chapter finalizes the development description of the system. The next chapters discuss the results of this work, consider some improvements for a future work and make conclusions about the work done.
6 Results

As a result of this work, two implementations of Control Software were developed: Java implementation (as the reference model) and Structured Text implementation. As a result of this work one minor bug in ST to C compiler was found. The bug was in ST FIND function that returns a position of a substring in a given string. The function takes two arguments: first argument is a string in which to find sub string the second argument is a sub string that will be searched for. We intended to use this function in order to compare the incoming message from simulation with a certain message type. Here is a typical usage of this function:

```
FOUND := FIND(MESSAGE, 'SIMULATION_INITIALIZED');
```

Where variable MESSAGE represents the message received from simulation and can be, for example, initialized as following:

```
MESSAGE := 'a26acb38:SIMULATION_INITIALIZED';
```

With such a parameters the function was stuck and no error message was shown. This bug was reported and fixed in the following releases of compiler.

In order to check if the goals of this work were achieved, we have to evaluate every goal criteria from the section 1.3:

a. To evaluate this goal criterion, we have tested the Simulation Software working with the same settings with both implementations. The testing was made in a systematic way: settings of the Simulation Software were different for each test. There were basically three classes of tests depending on the settings:
- The speed of both portal cranes is the same.
- The speed of the first crane is two times faster than the speed of the second crane.
- The speed of the first crane is two times slower than the speed of the second crane.

In each of the classes we have created cases with varying every workspaces processing speed. In all the cases both implementations produce the equal output of its states. Here is an extract example of one of the testing result:

<table>
<thead>
<tr>
<th>Java</th>
<th>ST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Got message from simulation: SIMULATION_INITIALIZED</td>
<td>Here is the message: SIMULATION_INITIALIZED</td>
</tr>
<tr>
<td>Initializing new session</td>
<td>Starting Simulation!</td>
</tr>
<tr>
<td>Processing message: SIMULATION_INITIALIZED</td>
<td>Writing message d1501kf4:SIMULATION_START to pipe</td>
</tr>
<tr>
<td>Sending message: a26acb38:SIMULATION_START</td>
<td>Here is the message: d1501kf4:ITEM_IN_PICKING_POSITION</td>
</tr>
<tr>
<td>Got message from simulation: a26acb38:ITEM_IN_PICKING_POSITION</td>
<td>Writing message d1501kf4:MOVE_FIRST_PORTAL_TO_FEED_BELT to pipe</td>
</tr>
<tr>
<td>Processing message: ITEM_IN_PICKING_POSITION</td>
<td>Here is the message: d1501kf4:FIRST_PORTAL_AT_FEED_BELT</td>
</tr>
<tr>
<td>Sending message: a26acb38:MOVE_FIRST_PORTAL_TO_FEED_BELT</td>
<td>Writing message d1501kf4:FIRST_PORTAL_AT_FEED_BELT to pipe</td>
</tr>
<tr>
<td>Got message from simulation: a26acb38:FIRST_PORTAL_AT_FEED_BELT</td>
<td>Writing message 1:FIRST_PORTAL_DOWN to pipe</td>
</tr>
<tr>
<td>Processing message: FIRST_PORTAL_AT_FEED_BELT</td>
<td>Here is the message: d1501kf4:FIRST_PORTAL_US_DOWN</td>
</tr>
<tr>
<td>Sending message: a26acb38:FIRST_PORTAL_TURN_OFF_MAGNET</td>
<td>Writing message d1501kf4:FIRST_PORTAL_TURN_OFF_MAGNET to pipe</td>
</tr>
<tr>
<td>Sending message: a26acb38:FIRST_PORTAL_DOWN</td>
<td>Writing message 1:FIRST_PORTAL_UP to pipe</td>
</tr>
<tr>
<td>Got message from simulation: a26acb38:FIRST_PORTAL_DOWN</td>
<td>Here is the message: d1501kf4:FIRST_PORTAL_DOWN</td>
</tr>
<tr>
<td>Processing message: FIRST_PORTAL_DOWN</td>
<td>Writing message d1501kf4:FIRST_PORTAL_DOWN to pipe</td>
</tr>
<tr>
<td>Sending message: a26acb38:FIRST_PORTAL_TURN_ON_MAGNET</td>
<td>Writing message 1:FIRST_PORTAL_UP to pipe</td>
</tr>
<tr>
<td>Sending message: a26acb38:FIRST_PORTAL_TURN_ON_MAGNET</td>
<td>Here is the message: d1501kf4:FIRST_PORTAL_UP</td>
</tr>
</tbody>
</table>

Here is the message:

```
SIMULATION_INITIALIZED
Starting Simulation!
```

Writing message d1501kf4:SIMULATION_START to pipe

```
SIMULATION_INITIALIZED
```

Writing message d1501kf4:ITEM_IN_PICKING_POSITION to pipe

```
d1501kf4:ITEM_IN_PICKING_POSITION
```

Writing message d1501kf4:MOVE_FIRST_PORTAL_TO_FEED_BELT to pipe

```
d1501kf4:MOVE_FIRST_PORTAL_TO_FEED_BELT
```

Writing message 1:FIRST_PORTAL_DOWN to pipe

```
d1501kf4:FIRST_PORTAL_DOWN
```

Writing message d1501kf4:FIRST_PORTAL_AT_FEED_BELT to pipe

```
d1501kf4:FIRST_PORTAL_AT_FEED_BELT
```

Writing message 1:FIRST_PORTAL_UP to pipe

```
d1501kf4:FIRST_PORTAL_UP
```

Writing message 1:FIRST_PORTAL_DOWN to pipe

```
d1501kf4:FIRST_PORTAL_DOWN
```

Writing message 1:FIRST_PORTAL_UP to pipe

```
d1501kf4:FIRST_PORTAL_UP
```

Writing message 1:FIRST_PORTAL_DOWN to pipe

```
d1501kf4:FIRST_PORTAL_DOWN
```
b. This goal criterion is evaluated using the counter evidence. During the development and test phases we have never seen the simulation stuck because of slow work of control software. In order to make it stuck we have introduced artificial delays to the control software and tested it with one user, who was not aware about how simulation flow must go. Gradually increasing artificial delays we aimed to find out at which point the user will notice that one of the components of the simulation flow goes slow or stuck. Found that at the point of 7 seconds delay the user noticed that both cranes were pending whilst there were items to pick. This means that control software has around 7 seconds to receive request from client, process it and respond with an answer. We also have implemented stress testing of the Control Software component by connecting 10 web clients to it. Five clients were running on the same machine as the server and other five clients were running on a different machine in the same network. The test was lasting for 5 minutes and there were noticed no delays in any of the clients.

c. The easiness to design new scenarios is provided by the software architecture described in section 3.1. The three-tier architecture allows splitting the logic of Control Software from its clients. Having developed the communication protocol between Control Software and Simulation software, now it is easy to change the logic of the Control Software in order to design new control scenarios, without affecting other parts of the system. In order to implement new scenarios (create new control rules), for the current design it is necessary to understand only the Control Software internals and there is no need to be aware about how simulation engine or communication protocol works. This significantly minimizes the time required to understand the Control Software system and implement the required changes comparing to the monolith solution, where control software and simulation engine are merged into one software component.

d. Figure 6.1 illustrates how the last goal criterion is satisfied: the resulting control software works with simulation model, which is displayed on the web page. The web page uses Unity Web Player for running 3D model in client’s browser. Unity Web Player plugin is required in order to run the model. If the plugin is not installed in the browser, the system will provide the installation link. Once the plugin is installed it is possible to run the model. It uses JavaScript for running the model in a browser and communicates with a remote Control Software via TCP socket channel. During the startup of the model it will ask to provide the IP address of the machine where Control Software is installed. The Control Software installation instructions are outlined in appendix A.
Figure 6.1. Simulation Software working in Google Chrome web browser
7 Conclusions and Future work

In chapter 6 we have shown that all the goal criteria are satisfied, which means that all the goals of this thesis were achieved. The correctness of the ST to C compiler was illustrated using the developed ST control software and comparing it with its Java reference example. The Company was presented as highly technologic compiler construction company by placing the simulation model on the Company web site and running control software on the server.

A big variety of enhancements can be done to improve this work. It actually depends on concrete requirements and goals. Here are some possible future improvements:

- Increase reliability and performance of the server (Control Software). As mentioned before in section 3.1, this can be achieved with the usage of software quality tactics.
- Add support for handling exceptional situations in simulation. For example, what should happen if the item falls from the belt or crane? Such situation should be detected and dealt appropriately.
- Allow user to interact with simulation model “on-the-fly”. For example, we can allow user to enable/disable some workspaces, change portal location e.t.c. System should detect such a changes, apply them and keep doing a stable job.

Note that last two improvements are likely to affect all parts of the system, making changes in communication protocol, control and simulation software.
References

Appendix A – Control Software installation instructions

Unity webplayer requires a socket served policy in order to connect to a remote server. In order to allow Simulation Software to fetch this policy, the server has to run sockpol.exe program. It can be found in the Unity install folder. This is a simple Mono executable file, meaning that it can be run on any platform supported by Mono. First the socket policy provider has to be launched by typing the following command:

```
mono /path/to/sockpol.exe
```

Note: do not terminate sockpol.exe process until the control software is finished working. Otherwise you will not be able to accept new connections from client.

Now we have to run the provided Control Software executable file named “STControlComponent” (or “STControlComponent.exe” for Windows platform). Run the server using the following command (Unix):

```
./path/to/STControlComponent
```

This will launch the Control Software as a server on the machine. From now the server waits for new connections. The next step is to connect to the server from Unity Simulation Software. First, it is needed to open a Unity client. This work focuses only on the webplayer Unity client build, which is an HTML page that can be opened in the browser. But it is also possible to use build for any other platform that is supported by Unity. Having opened the HTML file, the window similar to one on figure A.1 should appear. Figure A.1 shows the starting window of simulation.

![Figure A.1. Simulation starting window](image)

In this window server IP address (where sockpol.exe and STControlComponent are launched) has to be entered. The port numbers should remain with default values. Having pressed “Connect” button, the client will establish connection with server and will start simulation. To stop simulation close the browser tab.
Appendix B – Semantically equivalent code comparison

This appendix shows semantically equivalent implementations of processing SECOND_PORTAL_IS_DOWN message in both Java and ST languages.

**ST implementation:**

```st
FOUND := FIND(MESSAGE, SECOND_PORTAL_IS_DOWN);
IF(FOUND <> 0) THEN
    CLIENT_ID := LEFT(MESSAGE, 1);
    IF(SECOND_PORTAL_LOADED <> TRUE) THEN (*just pick up the item*)
        MESSAGE_FOR_CLIENT := CONCAT(CLIENT_ID, ':', 'SECOND_PORTAL_TURN_ON_MAGNET');
        PRINT(MESSAGE_FOR_CLIENT);
        MESSAGE_FOR_CLIENT := CONCAT(CLIENT_ID, ':', 'SECOND_PORTAL_UP');
        PRINT(MESSAGE_FOR_CLIENT);
        SECOND_PORTAL_LOADED := TRUE;
    ELSE (*release the item*)
        MESSAGE_FOR_CLIENT := CONCAT(CLIENT_ID, ':', 'SECOND_PORTAL_TURN_OFF_MAGNET');
        PRINT(MESSAGE_FOR_CLIENT);
        MESSAGE_FOR_CLIENT := CONCAT(CLIENT_ID, ':', 'SECOND_PORTAL_UP');
        PRINT(MESSAGE_FOR_CLIENT);
        SECOND_PORTAL_LOADED := FALSE;
    END_IF
END_IF
```

**Java Implementation:**

```java
else if(msg.equals(SECOND_PORTAL_IS_DOWN)){
    isSecondPortalDown = true;
    if(deviceManager.secondPortalIsLoaded()){ sim.secondPortalTurnOnMagnet();
        sim.secondPortalUp();
    } else {
        sim.secondPortalTurnOffMagnet();
        sim.secondPortalUp();
    }
}
```