CHALMERS

HMI debugger/monitor

Implementation of a debug interface

Master of Science Thesis in the programme
System Control and Mechatronics

OLA EDWARD
CHRISTOFFER HINDELID

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, October 2010

HMI debugger/monitor
Implementation of a debug interface

Ola Edward
Christoffer Hindelid

Examiner: Jan Skansholm

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden October 2010

Abstract

The company Mecel AB have a software product called Populus, which is a complete tool-chain for designing, developing and deploying user interfaces for distributed embedded systems. A manufacturer can use the program suite to make customized HMIs (Human Machine Interfaces) in a time-span that is much lower than in conventional methods.

At the start of the project work, the Populus system did not have any easy way for the user to debug the HMI that was being designed. This was a problem as the program suite include a lot of features and a HMI design could grow to be quite complex. Mecel wanted to implement a debugger that could be shipped with the product.

The main goal of the project was to implement different tools that could be used by HMI designer to get a better understanding of why a potential problem did occur. The end result produced a start of a functional debug-interface, which could be used even by a HMI designer that was not familiar with the Populus system. This report will describe how this was achieved, by going into detail about the different implemented solutions. In addition to this, motivation of different design choices will also be presented. The work was carried out in both C++ and Java, as both of this programming languages are used in the Populus system.

Contents

1 Abbreviations

Display - A device that displays graphical content to the user.
Faceplate - The mechanical buttons connected to a HMI.
FU - Functional Unit.
GUI - Graphical User Interface.
HMI - Human Machine Interface.
ODI - Open Display Interface. This is the communication protocol used between connected applications and the Populus Engine.
Popolus Editor - Java platform that is used to create HMIs.
Popolus Engine - C++ platform that runs the HMIs.
Target - The actual hardware platform for which the software are developed.
TCP/IP - Transmission Control Protocol/Internet Protocol.
XML - eXtensible Markup Language.

2 Introduction

2.1 Background

The company Mecel AB have a software product called Populus, which is a complete tool-chain for designing, developing and deploying user interfaces for distributed embedded systems. A manufacturer can use the program suite to make customized HMIs (Human Machine Interfaces) in a time-span that is much lower than in conventional methods.

At the start of the project work, the Populus system did not have any easy way for the user to debug the HMI that was being designed. This was a problem as the program suite include a lot of features and a HMI design could grow to be quite complex. Mecel wanted to implement a debugger that could be shipped with the product.

2.2 Method

The project was divided into three distinct phases:

1. **Initial phase** - Learning about how the existing code in the Populus project functions and decide exactly what should be implemented in the project. Knowledge about the existing system was mainly learnt by studying the relevant protocol descriptions in Mecels documentation of the system.

2. **Theoretical work** - To be able to implement the more advanced functions that are required in the project, some code evaluation was preformed. In this phase different ways of solving a given problem was considered from a theoretical point of view, to find best possible implementation.

3. **Programming** - The programming stage of the project did consist of practical implementations of the tasks specified in the project. The implementation was done using Eclipse and Visual Studio 2005 as development environments.

2.3 Goal

The main goal in the project was to implement the basic functions that were needed to get a working debug-interface in the Populus editor. This work was focused around getting the background functions working in a satisfactory way. This means that the visual presentation in the Populus editor only consisted of a small part of the total work done in the project. The tasks that needed to be fulfilled in order to get the basic functionality was divided into working packages that are presented in this section.

In addition to this, the master thesis needed to be documented in a written report. An oral presentation where also prepared and preformed, both at Chalmers and at Mecel AB.

Initial phase and planning
The first part of the project consisted of getting an introduction to the workplace and the task at hand. To be able to better understand the environment in which the project was done, existing document that describes this was studied. In addition to this the work that was preformed the following weeks had to be planned.

Port the existing Trace-program
The Populus system has functionality to log different events that occur and send them to connected devices. At the start of the project, an existing trace program where used to receive this information. This program was a standalone program written in C++. This task consisted of porting the existing program functionality to Java, which would let the Populus editor handle this trace information. This task consists of three subtasks:

1. Use the existing Trace-program and get familiar with it.
2. Interpret the binary data message that are sent from the engine and translate them to readable data.
3. Save the data that were derived in task 2 to a file and/or the HMI-editor.

Writing debug protocol
All the functions needed in the debug-interface were documented.

Requesting data from the engine
When debugging the user should be able to get information about: data, indicators, guards, priority stack, and history stack. To be able to do this, a way to request and receive specified data from the engine was derived.

Setting data into the engine
When debugging the user could want to set the following: data, indicators, guards, priority stack, and history stack. This could be done to force the engine into a specific state that is interesting for the user. As this could be problematic for continued execution, this step was not sure to be fully implemented.

Evaluating of difficult code parts
It was considered desirable to be able to run the Populus Engine at different speeds, and to pause the execution. To be able to do this, some way to slow down the execution speed of the Populus engine was needed.

Another feature that was wanted was to be able to save and recreate a session performed in the Populus engine. This was done by saving all needed data to a file that could later be executed in the Populus system. Both these problems were considered to be complicated to perform in practice, and both can be solved in many different ways. Because of this, some time was spent evaluation the different possibilities.

Implement speed control of the execution speed of the Populus engine
The solution that was developed in the evaluation part had to be implemented.

Implement record and playback of sessions
The solution that was developed in the evaluation part had to be implemented.

Visualize in editor
A simple user interface that displays the implemented functionalities was implemented in the Populus editor.

Demo/Presentation
Some time was spent setting up a presentation of the project.

Writing the project report
The work that was done in the project was documented in the project report.

3 The Populus System

This section will give a brief overview of the system in which all the code described in this report was implemented. It will also explain the main features of the Populus system, and how the editor and the engine interact. The discussion will not go into great detail, but will instead focus on the key points needed to understand the work done in the project.

3.1 Populus Editor

Figure 1: An overview of the Populus editor.

The Populus editor is shown in figure 1 and is a desktop program written in Java and based on the Eclipse platform. The program is used to create HMIs for distributed embedded systems by using a user-friendly graphical interface. This way, complete HMIs can be created and verified without having to write any program-code, which can significantly shorten the development time for new products. The Populus editor saves the complete HMI in a binary database that are saved in the XML[2] format. This database contains all graphical components that are needed to render the interface on the screen. It also includes all logical connections between different buttons and the events that they trigger.

3.2 System layout

The main layout of the Populus system is shown in figure 2. As can be seen in this figure the Populus engine is the central part of the system, and has a lot of different responsibilities. Firstly it is used to interpret the HMI-database that is created in the Populus editor and to render the result on one or more connected

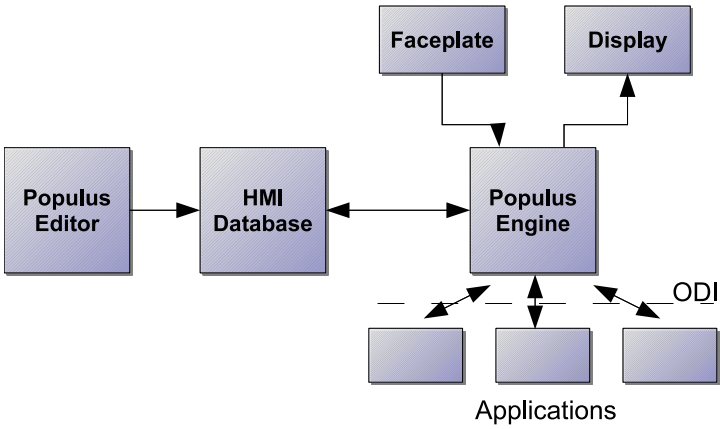

Figure 2: The layout of the Populus system.

displays. It is also responsible for priority handling, which means that the user gets to see the information that is most important at each given time-instance. Above that it is also in charge of input handling, where information can be passed both from other connected devices and the faceplate.

3.3 Populus Engine

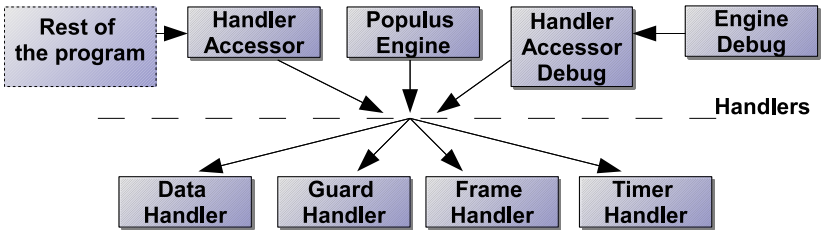

Figure 3: An overview of the layout of the Populus engine.

The internal composition of the Populus engine is shown in figure 3. The main part of the program is the *PopulusEngine*-class. This class instantiates handler-classes, which are used to handle different kinds of data- and state-information. Each handler-class has a specific area of use, were for example the *GuardHandler* contains all information needed to keep track and update the guard information.

To make the different handlers accessible from the rest of the program the *HandlerAccessor*-class is used. This class holds pointers to all Handler-classes and returns the references to the interface-classes, which contains all public functions for each Handler.

For this project the most important part of the Populus system is the *EngineDebug*-class. As the name implies this class is used to implement debug-features.

3.3.1 The debug-interface

One important consideration during the project work was to minimize the impact that the new functions have on the existing code. This was done by separating the debug-functionality using specialized debug-interfaces that where only used by the *EngineDebug*-class. The debug-interfaces where implemented using the *HandlerAccessorDebug*-class (see figure 3). This class is used as the connection between the *EngineDebug*-class and other important classes of the Populus system.

By using this method, the new functions, which were introduced by the debug-functionality, are not accessible from the other classes in the program. This means that the debug-functions will act like an encapsulated part of the program, which minimized its interference with the rest of the program.

3.3.2 The main loop

The main loop in the Populus system is what drives the program forward. This loop is placed in the *PopulusEngine*-class in figure 3, and is responsible for updating the state of all other classes in the program. A flowchart of the design of this can be seen in figure 4.

As can be seen in this figure there are three different events that can change the state of the engine. The first one is timers which are used to control events where the timing is important. One example of this is to be able to display an animation, where different picture-files are drawn depending on the system time. Another important use of timers is to be able to implement reoccurring events. This is done by using a special type of timers which schedules itself for a new execution, with a predefined delay, after it is handled by the system. This type of timers are for example used to tell the system when to redraw the screen, in this case the period of the timer is dependent on the desired refresh rate of the display. This is why it follows the redraw-timer in figure 4.

It is usual to develop a software-emulation of a physical system for which a HMI is developed. The reason for this could be many, but are mainly related to ease of use in development. Because of this, events which are spawned by the operating system can also be used as an input to the Populus system. This is

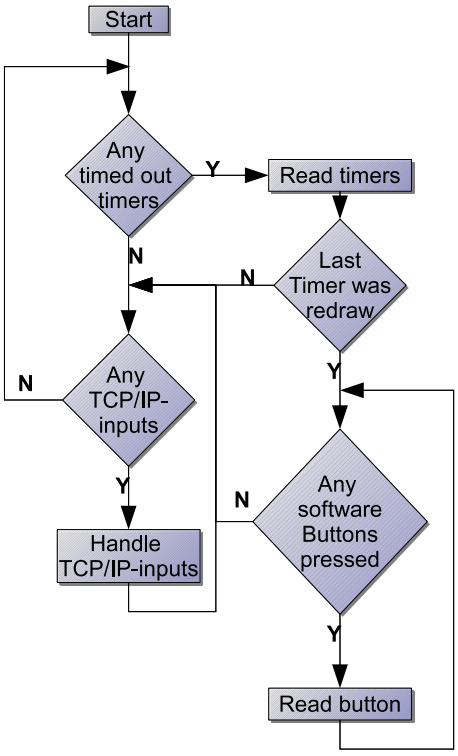

Figure 4: A flowchart of the design of the main loop in the Populus system.

commonly used to be able to imitate the behavior of real buttons by using icons in the emulated system. These types of events are usually generated by the operating system when a window is redrawn, which means that their generation is heavily dependent on the timer that controls the refresh rate of the display.

The third kind of event that needs to be handled by the Populus system is inputs from connected FUs. This connection is usually done by using a TCP/IP-connection where each remote device uses a separate socket.

3.4 Data and indications

The *DataHandler*-class is used to save and process different kinds of data information. This is, for example, used to let FUs communicate with the engine or other connected devices. It also contains a set of indications for each connected FU. Indications are an efficient method of storing, and transferring, boolean values.

An overview of the *DataHandler*-class is shown in figure 5. As can be seen,

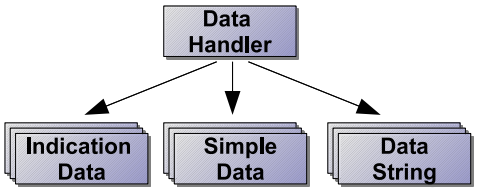

Figure 5: An overview of the *DataHandler*-class.

there are three different kinds of data that were of interest in this project work, all which is accessed through the *DataHandler*.
The *DataHandler* is also responsible for a subscription-interface where other parts of the Populus program can listen to changes in data values. This functionality exists both for changes in the different data-types as well as for indication-data.

3.5 History stack and Priority stack

The history stack is used to be able to remember the order in which previous screens have been displayed to the user. This can be useful in a menu system, for example if the HMI designer wants to include a back-button.
The priority stack is used to decide what information that will be displayed on the screen. If different parts of the system want to access the display at the same time, the *PriorityHandler* uses the priority stack to determent which request that will be granted.

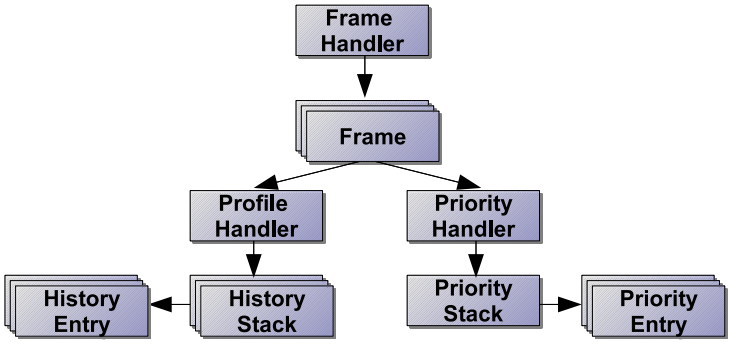

Figure 6: An overview of the *FrameHandler*-class.

The Populus system usually contains more than one set of priority and history stacks. This is due to the fact that the Populus engine can have multiple frames. Frames are used to simplify the case when the engine is run against

multiple screens. In these cases a separate frame can be used for each of the displays.

An overview of the layout of the *FrameHandler*-class is shown in figure 6. As can be seen, each frame has a separate instance of the *PriorityStack*-class. Each *PriorityStack* consists of multiple instances of the *PriorityEntry*-class. This class is used to store information about which part of the program that should have access to the display.

The situation is quite similar in the case of the *HistoryStack*, but with a big difference. This dissimilarity is due to the fact that the *HistoryStack*-class is accessed through the *ProfileHandler*, and that each *Frame* can be connected to more than one profile. Each profile can then have one or more instances of the*HistoryStack*-class, which consist of multiple instances of the *HistoryEntry*-class. The *HistoryEntry*-class is the information holder, that contains the needed data to use the *HistoryStack*.

3.6 Guard values

Guards are boolean values which are used to symbolize different states in the engine. This information is used to control the behavior of the system. One example of this could be a guard representing if a mobile phone is connected or not, which could affect the different menus that are displayed.
The rest of the system accesses Guard values by using the *GuardHandler* (see figure 3). It is also possible for other parts of the system to subscribe to changes in the guard values, this functionality is also managed by the *GuardHandler*.

3.7 Timers

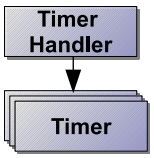

Figure 7: An overview of the *TimerHandler*-class.

In the Populus system timers are used to make events happen at a specified time, or to arrange recurring events. This is handled by the *TimerHandler*-class, of which an overview can be seen in figure 7. The *TimerHandler* has multiple instances of the *Timer*-class, one for each timer that is present in the system. The *Timer*-class is used to hold all information that is needed for a timer.

4 Overall design guidelines

A couple of key issues where considered during the design of the program-code. Some of these issues are related to good programming practice, others were given by our supervisors at Mecel. This section will give a brief overview of the guidelines that have been considered during this project work.

4.1 Observer-pattern

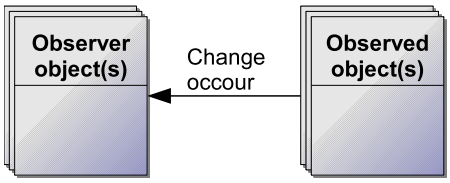

Figure 8: An overview of the observer design pattern.

The observer pattern is described in figure 8 and is used to let one or more object(s) (the observer) listen to events generated by another object (the observed). This means that the observed and the observer form a publish/subscribe-relationship, where the observer can remain passive and wait for new information created by the observed object [4]. There can be many observers to one object and one observer to many objects. The normal way of implementing this functionality is to have a function in the observed object that notifies all subscribing observers whenever a change is made to a data source of interest.

It is then the observers responsibility to handle the data that are generated by the observed object. The main reason to use the observer-pattern is to avoid polling-situations, where an observer instead checks for data-updates. This is bad for two reasons; firstly it is an inefficient method as unnecessary calls are made, even when no new data is available. Secondly, the time between new data-information is generated, and when it is available to the subscribing observer can be longer than necessary. Because of this observer-pattern were used on numerous occasions during the project work.

4.2 Keep work intensive changes out of the engine

The Populus engine is designed to be able to run on target systems where the hardware specifications can be relatively low. Because the debug-functions will be used on these kinds of systems, it was important to avoid changes that would have a negative impact on the performance of the engine. The Populus editor, on the other hand, is designed as a desktop program, where it is safe to assume that the available hardware is much better. In addition to this, the editor does not have the same type of real-time demand as the engine. Because of this,

an important factor, when evaluating different programming-solutions, where to put as much of the computations as possible in the editor.

4.3 Minimize the amount of TCP/IP traffic

Information is sent between the editor and the engine using the TCP/IP protocol. This could present a problem, for example if the editor and the engine do not run on the same machine. In these cases the response-time and the bandwidth of the network system could be limiting factors. This was considered throughout the project, as the needed amount of data traffic was kept to a minimum.

4.4 Use short TCP/IP messages if possible

The TCP/IP messages that are used to transfer data between the engine and the editor consist of byte arrays. The theoretical size limit of each of these arrays is 4 294 967 296 (2^{32}) bytes, but during the course of the project the size of the TCP/IP messages where kept as small as possible. This lead to that larger data-chunks were divided into as small parts as possible. The main reason for this design choice where to make debugging and maintenance easier. Another consideration was also that the messages would be able to fit into a single-frame CAN-bus message, where the size of each message should not exceed 8 bytes [3]. This was a consideration, as it was possible that the debug-interface could be used on the CAN-bus, and the overhead is lower when using this size of messages.

5 Reading information from the engine

One of the key features of a debug-interface is to be able to read the value of key data-variables. Because of this, this functionality needed to be implemented in the debugger developed in this project. This section of the report will describe how these parts of the program are designed, and how messages are sent, and dealt with, in the different parts of the system.

5.1 Design overview

An overview of the design of the debug-interface, that was implemented in the editor, is shown in figure 9. This design uses two layers of observers, where the second layer listens to the first layer, and sends updates to the main part of the program.

Message-passing between the Populus editor and the Populus engine is done in

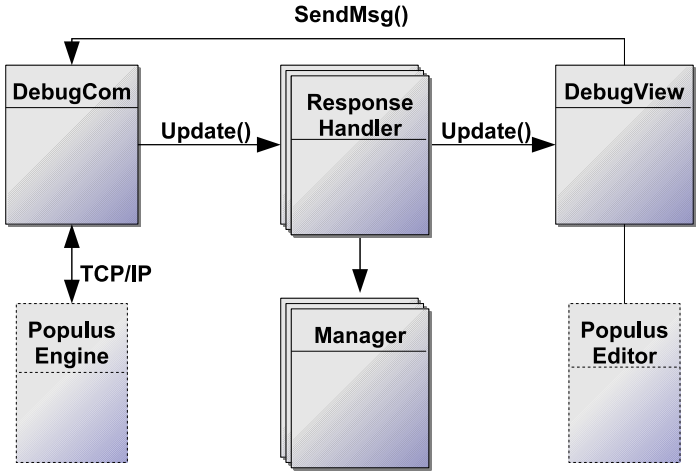

Figure 9: An schematic view of the design of the debug interface.

the *DebugCommunication*-class. This class sets up a TCP/IP connection, where the data is transferred as packages coded as bytearrays. The *DebugCommunication*-class is also able to communicate with the engine by sending byte arrays representing different commands over the TCP/IP connection. These messages can be used to request certain information or specify which debug-information that engine should send. When a message is received from the engine the *DebugCommunication*-class removes all overhead-information from the message and sends it to all available *ResponseHandlers*.

The design described in figure 9 contains multiple instances of different *ResponseHandler*-classes, one for each type of data that can be requested from the engine. The reason for this is to divide the code into logical, object oriented, parts to make it easier to maintain. Also, the different response handlers handle their data in slightly different ways, which make this a natural separation. The main functionality of the *ResponseHandler*-classes is to take the data received from the engine and parse it into meaningful information. This set of classes is also responsible for recognizing different message-types and notify the main program when certain events occur. Each response handler have a *Manager*-class which is used to keep track of the information needed in each response handler.

The *DebugView*-class is responsible for the GUI (see figure 10), and is also the connection between the debugger and the rest of the Populus system. It is also responsible for passing user-inputs to the other parts of the debug-system.

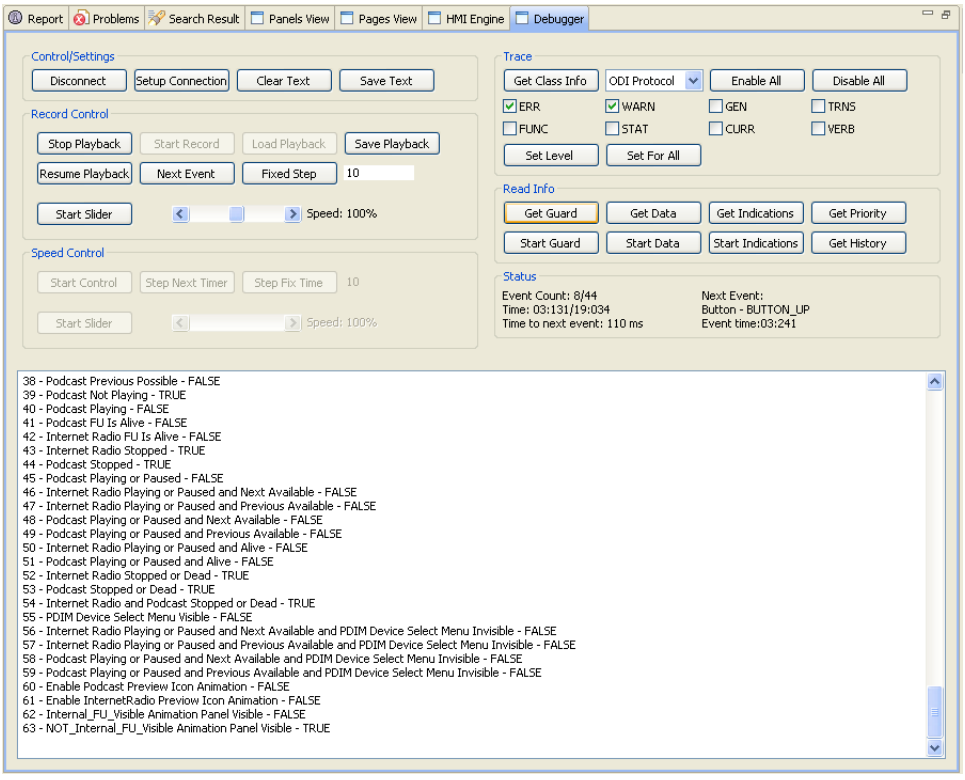

Figure 10: The debugger user interface.

5.2 Trace program

The Populus engine has functionality to generate debug-information when different events occur. The trace part of the debugger is used to read this information and display it to the user. The trace program is also able to communicate with the engine to specify which debug-information that engine should send.

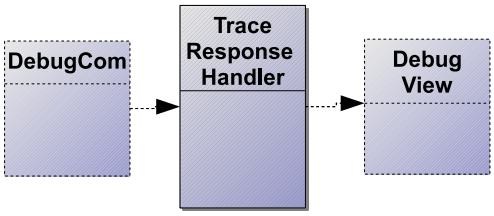

Figure 11: An overview of the trace program.

The design of the Trace program is shown in figure 11. The main part of the trace program is the *TraceResponseHandler*-class. This class is used to convert the binary input from the engine into clear-text messages, which can be sent to the editor to be displayed to the user. It also handles debug-commands which are sent from the engine. These message are used to specify what trace information that the engine should send.

5.3 History Stack

The history stack in the Populus system is used to keep track of the order in which different pages have been displayed to the user. As this is often used in HMIs to make transitions in menus, it can be important to be able to derive this information if the HMI do not work as intended. The implementation of the history stack in the Populus editor are based around four classes, which are displayed in figure 12.

The *HistoryStackManager*-class initializes an array of the *HistoryStack*-class for each profile that is used at the present time. It is then responsible to update this array when new information is sent from the engine. As a history stack is sent in many different TCP/IP-messages, another important functionality in the *HistoryStackManager* is the ability to keep track of when a history stack is ready to use. This is done by invalidating all history stack that are requested, and then keep track of the length of each new history stack that are sent.

The *HistoryStack*-class is used to store all information needed in a history stack. To do this a array of the *HistoryEntry*-class is used. A *HistoryEntry* contains all information needed to take one history-step. The *HistoryStack*-class also have some logic that signals if the program tries to add a *HistoryEntry* to

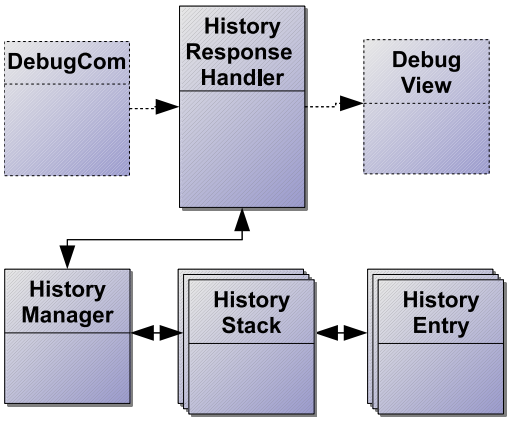

Figure 12: The overall design of the history stack in the editor.

an full stack or if the last added *HistoryEntry* fills the last spot in the stack.

5.3.1 History stack request

As the response to a history stack-request is passed in more than one TCP/IP-message, a specified sequence must be used to keep the debugger and the engine in sync. How this is done is explained in figure 13 and in the text below.

1. The user of the Populus editor requests the history stack for one or more profiles.
2. The engine receives the request and sends information about which profiles that will be updated.
3. The *HistoryResponseHandler* receives the initial message and invalidate the history stacks that will be received.
4. The engine generates a start message for the history stack of each profile that has been requested, which contains the length of every stack.
5. The *HistoryStackManager* creates a new history stack class for each start message with the correct length.
6. The engine sends every history entry for all the requested profiles.
7. The *HistoryStackManager* fills the history stack with the corresponding entrys. It also checks if all requested stacks have been received.
8. If all stacks are valid the *HistoryStackManager* sends a message to the *HistoryResponseHandler*, which notifies the main part of the debug GUI.

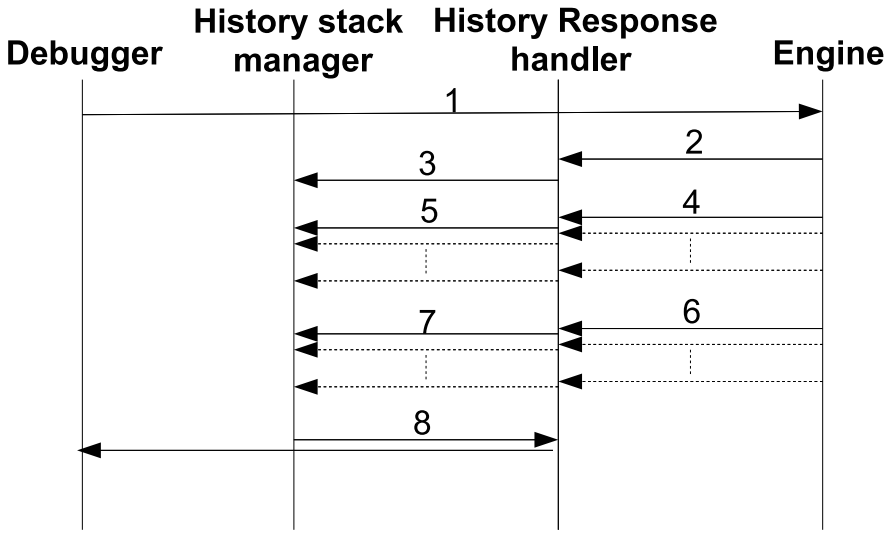

Figure 13: A flowchart of a request of history information.

5.4 Priority stack

The priority stack is used to control what contents that will be shown in the display. It is also responsible to make sure that pop-up messages are displayed in the right way, and in the right priority order. It is, for example, crucial that a message signaling an engine-failure is displayed rather than a message from the CD-player. This is especially true because the Populus system only allows for one simultaneous pop-up message.

The design of the editor side of the priority stack implementation is shown in figure 14. The *PriorityManager*-class uses multiple instances of *PriorityEntry*, where each entry represents one device that wants to access the display. The *PriorityManager* is also in charge of keeping track of when an update of the priority stack is finished, and to make information available to the rest of the system.

5.4.1 A priority stack request

In the Populus system the priority stack is implemented as a linked list, where the first instance in the list corresponds to the element which has the highest priority at the current time. One of the reasons for this is that the length of the priority stack is changing during runtime, as different parts of the program-code requests the use of the display. When the user of the Populus editor makes a request to get the priority stack, information regarding all priority entries needs to be transmitted from the engine. How this is done is explained in figure 15.

1. The user of the Populus editor requests the information in all the priority

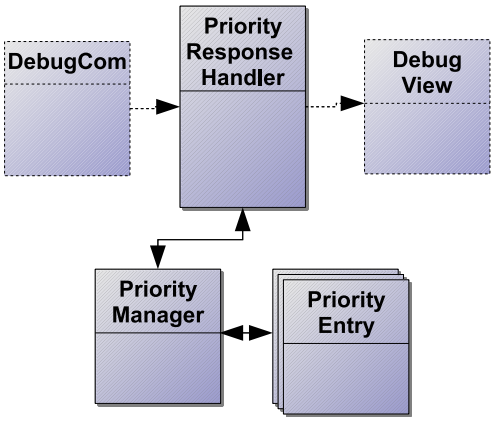

Figure 14: An overview of the priority stack implementation.

stacks.

2. The engine go through all slots in the linked list to get the current size of the stack.
3. A start message is generated containing the size of the stack and other general information.
4. The start message is sent to the editor.
5. The *PriorityResponseHandler* sends the start message to the *PriorityStackManager*.
6. On reception of the start message the *PriorityStackManager* empties the current priority stack. It then creates an array of the size of the stack that will be sent.
7. Every slot in the priority stack is used to generate a entry message, which is sent to the editor.
8. Each entry is placed in the priority stack array in the *PriorityStackManager*.
9. If the array is full after the last received message, the *PriorityResponseHandler* is notified.
10. The *PriorityResponseHandler* notifies all observers that the priority stack transfer is complete.

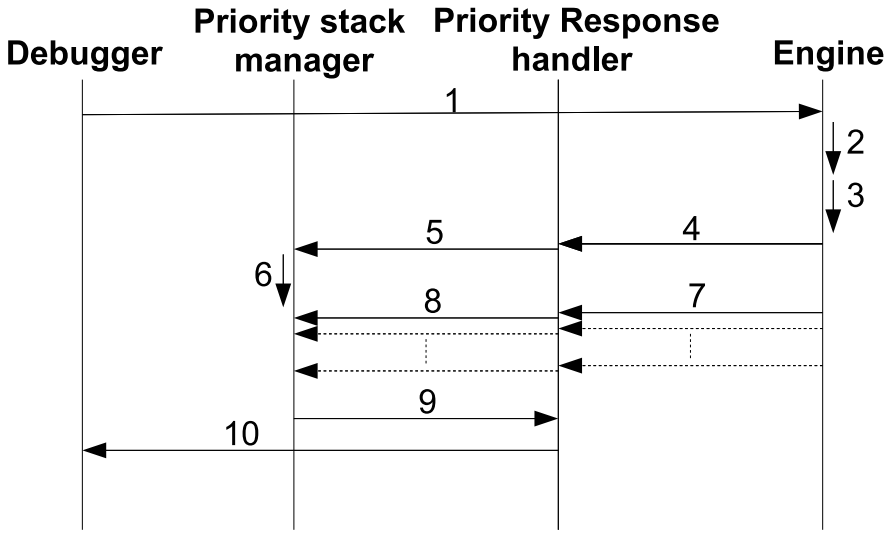

Figure 15: A flowchart of a priority stack request.

5.5 Guards

In the Populus system guards are used to be able to control the behavior of a HMI depending on boolean values. This is done by letting a guard represent a specific state and updating it to match the situation at the current time. For example, a guard that corresponds to if a CD is in the CD-player can be used in a media application. If this guard has the value false it would be reasonable to disable an Eject-CD button.

As can be seen in figure 16, the only class that are used by the *GuardResponseHandler* in the guard implementation is the *GuardManager*. This class is mainly used to translate the received guard-messages to guard information. As guard-data can both be sent one at a time or all at once, the *GuardManager* have methods to deal with both of these cases.

5.5.1 A guard request

The maximal number of guards that are allowed in the Populus system is 65 536 (2^{16}), even though it is unusual that any HIM uses more than maybe a couple of hundred. Because of this relatively large number, a response to a guard-request needed to be split into several messages. A schematic view of a guard request/response can be seen in figure 17.

1. The user of the debugger requests a list of all guards and their values.

2. The engine receives the request and generates a Guard-start-message containing the total number of guards that will be sent.

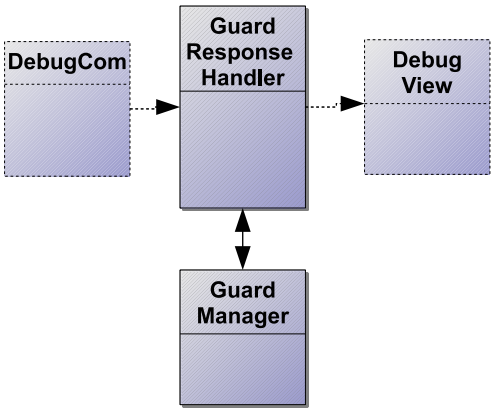

Figure 16: An overview of the Guard implementation.

3. The engine sends the start message to the editor.
4. The start message is passed to the *GuardManager*.
5. The *GuardManager* creates a new boolean-array with the same size as the number of guards.
6. The engine creates enough 8-byte messages to contain the value of all guards.
7. The engine sends all guard data messages to the editor.
8. All guard data messages are sent to the *GuardManager*.
9. The editor adds each guard-value to the guard-array. The ID of the guard corresponds to which place it has in the array. This means that the first guard in the first message will always have Guard ID 1.
10. If the last added guard have the same ID as the size that was specified in the start-message, the transfer is complete and the *GuardManager* notifies the *GuardResponseHandler*.
11. The *GuardResponseHandler* notifies the debugger.

5.5.2 Subscribing to guard-changes

As guards are designed to be changed during runtime, it is important to be able to keep an updated list of all guards in the editor. As frequent requests of all guard-values would lead to an undesirable amount of data-traffic, a subscription system is used. Using this system the editor can choose to only receive information about updates in guard values, which considerably reduces the data-traffic. A schematic view of a guard subscription is shown in figure 18.

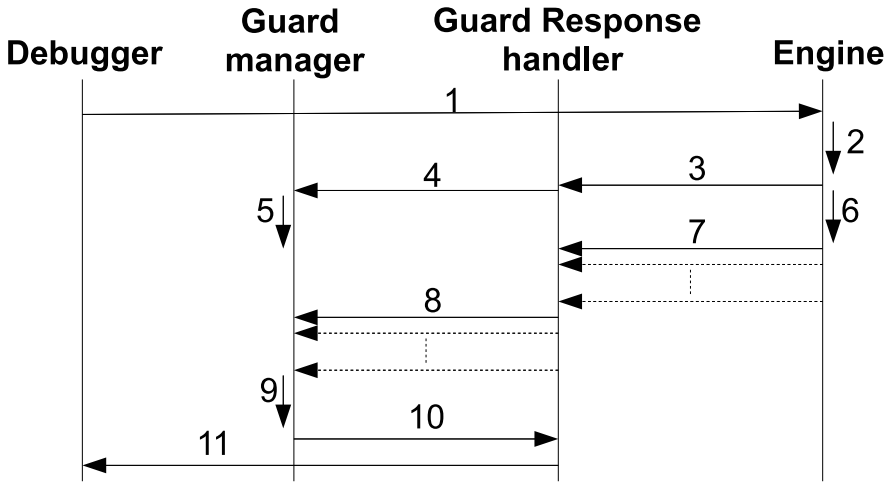

Figure 17: A flowchart of a guard request.

1. The user of the debugger starts a guard-subscription.
2. The engine use internal functions to register the editor for a guard-subscription.
3. The engine gets information about a guard change and generates a guard-update-message.
4. The guard-update-message is sent to the editor.
5. The editor updates the guard-array according to the message.
6. The *GuardManager* notifies the *GuardResponseHandler* that the update is complete.
7. The *GuardResponseHandler* notifies and sends the updated *GuardManager* to the debugger.

If the user of the editor is no longer interested in updates of guard-values it is also possible to unsubscribe to guard-changes. When this is done the engine simply unregisters the editor as a listener to guard-changes.

5.6 Data

One of the main purposes of the *DataManager* is to let different FUs communicate with the display. On example of this could be a CD-player that wants to display the current track-number to the user. This is done by letting FUs save information in the engine. This data can then either be displayed on the screen or used in internal functions. The Populus system allows for three kinds of data to be stored:

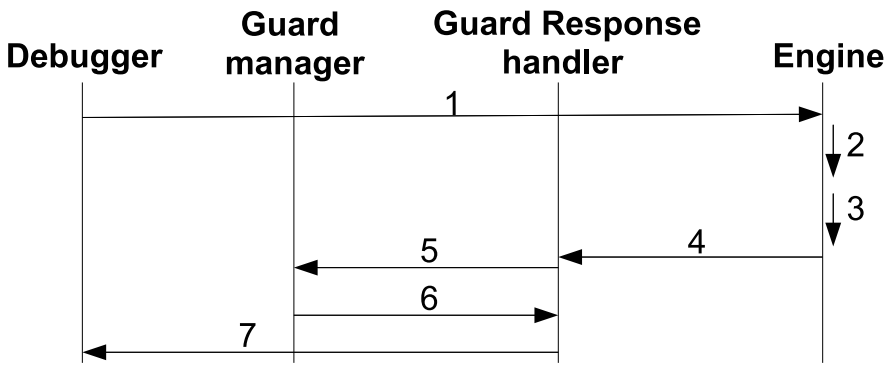

Figure 18: A flowchart of a guard subscription request.

1. **Simple data** Is used to store a 32 bits data value.
2. **String data** Is used to store strings of dynamical length.
3. **List data** A complex data-type that allows for whole lists to be stored. This could for example all posts in an address book.

Only the first two of the data-types will be implemented in the thesis work, as list data was considered to be outside of the scope of this project.

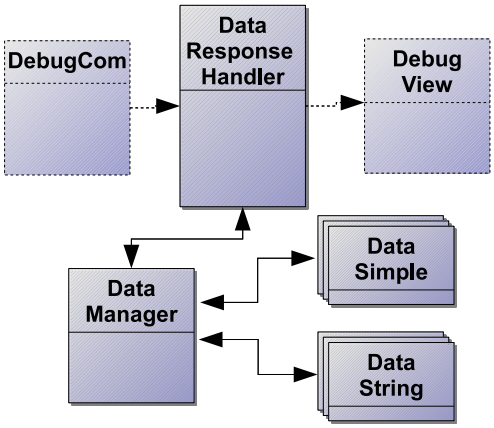

Figure 19: An overview of the Data implementation.

The implementation used when reading data from the engine is shown in figure 19. When data is received from the engine the *DataManager* is used to handle the incoming messages, and keep track of when all requested data have been received. All data are stored in one of the two different data-classes,

DataSimple or *DataString*. Both these classes inherent most of its functionality from the same base-class, and only differs in what kind of data they hold.

5.6.1 A data request

Inside the *DataManager* both simple- and string-data are saved in the same array, with a data-type variable that is used to determine what kind of data that is in each spot. The number of free spots in the data-array can be of interest to a user of the debug interface. Because of this the size of the data-array, which is set as a configuration parameter for each implementation, is sent to the debugger when it connects to the engine. The interaction between the engine and the debugger during a data request is displayed in figure 20.

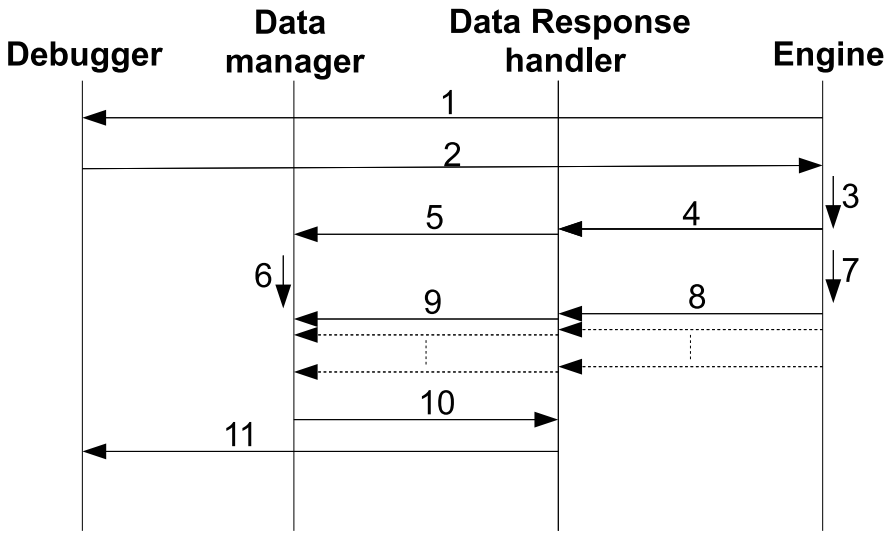

Figure 20: A flowchart of a request of data information.

1. When the debugger first connects to the engine a initializing message is sent containing the size of the data-array.
2. The user of the debug-interface requests all data present in the engine.
3. A start message is generated containing the number of used data entries and other general information.
4. The start message is sent to the *DataResponseHandler*.
5. The *DataResponseHandler* sends the message to the *DataManager*.

6. On reception of the start message the *DataManager* empties the current data-array stored in the *DataManager* and creates an array of the size of the number of data entries that will be received.

7. The engine iterates through all slots in the data-array and checks its data-type and generates an entry message.

8. The generated message is sent to the editor.

9. Each entry is placed in the array in the *DataManager*.

10. If the array is full after the last received message the *DataResponseHandler* will notice the rest of the system.

11. The debugger are notifed that the data-transfer is complete.

The data-array in the engine can change drastically in both size and appearance during runtime. The reason for this is that each profile can have its own set of data-variables, but still use the same array in the *DataManager*. Because of this, the user of the debugger needs to be cautious when requesting data. It could, for example, be dangerous to assume that the last requested data is still valid after the engine changes state.

5.6.2 Subscribing to data-changes

Much like guards (see section *5.5*) the information in the data-array is changing during runtime. To be able to have an updated list of data in the debugger a subscription system is used, which is outlined in figure 21.

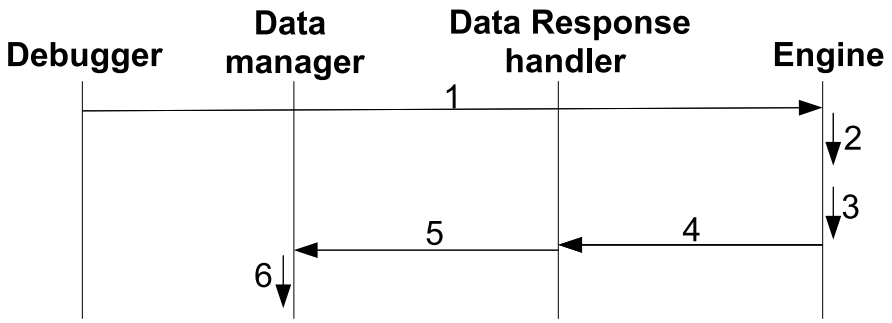

Figure 21: A flowchart of a data subscription request.

1. The debugger sends a request to start to listen to changes in the data-array.

2. The engine register the debugger as a listener for data changes.

3. When a change in data occur the engine checks the type of the data and generates an update message.

4. The update message is sent to the *DataResponseHandler*.

5. The update message is sent to the *DataManager* which updates the specific data.

6. When the update message is received the *DataManager* checks if the data ID is in the current list in the *DataManager*. If that is the case the entry is updated. If not the message is discarded.

If the user of the debugger is no longer interested in the data-array the debugger can be unsubscribed to data changes. This is done by sending an unsubscribe-message to the engine, which simply unregisters the debugger as a data-listener.

5.7 Indications

An indication is a boolean data-type that are treated a bit differently than the other data-types in the Populus system. Each FU have a set of 56 (7 bytes) indications that are saved in the engine, which it can use to send information. This design was used in the Populus system at the start of the project work. The reason for this is that some applications sends indication-data frequently, and this implementation was considered to be efficient. 56 indications were chosen to be able to transmit all indications in a single CAN-frame, which reduces the traffic on the bus.

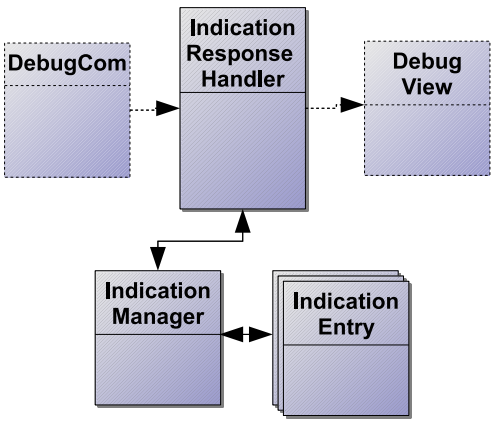

Figure 22: An overview of the read Indication implementation.

Figure 22 shows the design used in the read indications realization that are used in the editor. The main responsibility of the *IndicationManager*-class is to make sure that an update of the list of indication-data is done in the correct

way. This can be problematic as the indication-data for each FU is sent in a separate message, which means that the manager needs to keep track of an update that span over several messages. All indication data for each FU is saved in an separate instance of the *IndicationEntry*-class, which are instantiated by the *IndicationManager*.

5.7.1 A indication request

The user of the editor requests indication-data by sending a message to the engine. How this is done is displayed in figure 23.

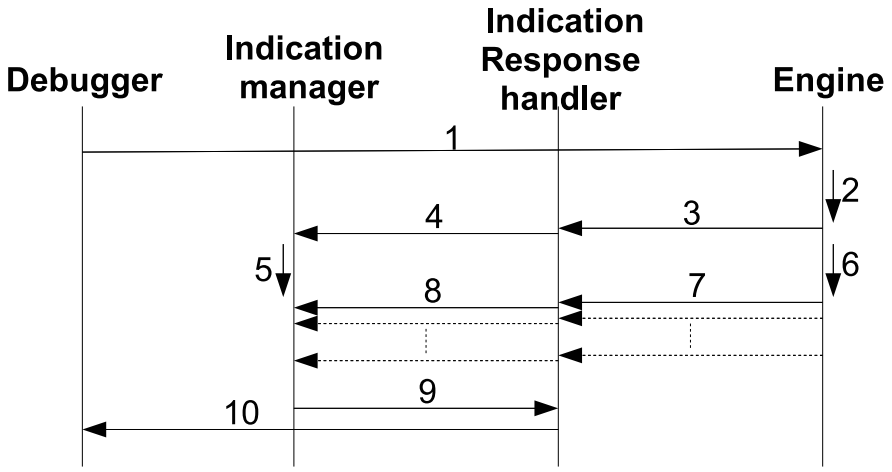

Figure 23: A flowchart of a indication request.

1. The user of the debug interface requests all indications from the engine.
2. The engine counts the number of FUs that are connected and generates a start message containing this information.
3. The start message is sent to the *IndicationResponseHandler*.
4. On reception of the start message the *IndicationResponseHandler* sends the information to the *IndicationManager*.
5. The *IndicationManager* removes all stored indication-data and saves the number of FU that will be sent.
6. Each indication-array in the engine is used to generate one indication message.
7. The engine sends each message to the *IndicationResponseHandler*.

8. The *IndicationResponseHandler* forward the message to the *IndicationManager* which add it to the corresponding FU.

9. If the array is full after the last received message, the *IndicationResponseHandler* is notified.

10. The *IndicationResponseHandler* notifies all observers that the indication stack transfer is complete.

5.7.2 Subscribing to indication-changes

The user of the editor can chose to get all indication-updates sent when they happened. This is a easy way to have an updated list of all indication-values without having to make frequent indication request. A flowchart of a indication subscription is shown in figure 24.

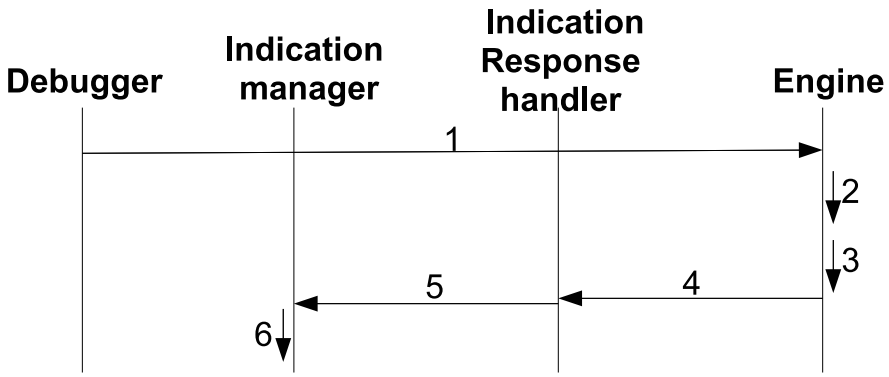

Figure 24: A flowchart of a indication subscription.

1. The editor sends a request to start to listen to changes in indication values.

2. The engine register the editor as a listener to changes in indication values.

3. When a change in indication value occur the engine generates an update message.

4. The update message is sent to the *IndicationResponseHandler*.

5. When the update message is received, it is sent to the *IndicationManager*.

6. When the update is completed the rest of the system is notified.

The user of the editor can decide to stop the subscription of indications. This is done by sending a message to the engine, which unregisters the editor as a listener to changes in indication values.

6 Controlling execution speed

One important function in a debug-interface is the ability to stop execution and evaluate different key information at a given time. Because of this, the ability to control the execution speed of the Populus engine was a desired feature in the project. This feature was divided into sub-goals which are described in the list below.

- **Start/Stop execution** The user should be able to pause and resume the engines execution at any given time.
- **Jump forward in time** When controlling the execution of the engine the user should be able to force the engine into making a controlled jump forward in time.
- **Step to next timer execution** As many changes of the engine state is generated by triggering times, a feature to step between timers was desired.
- **Run the engine at different speeds** A user of the debug system should have the ability to control the execution speed of the engine. This could for example be useful to be able to see an animation run at a lower speed.

6.1 Code implementation

In the Populus engine the execution time of a timer is setup and evaluated with the current system time as a reference. As timers are the main component that drives system forward, a logical starting point for the time-control implementation was to modify the method that returns the system time. This is done by detaching the system clock and replaces it with time-messages that are sent from the editor. This means that the engine no longer uses the built-in system clock of the target system, but rather gets time-updates from the editor.
By using this method, the engine can be stopped completely by starting the time-control and not send any time updates. It can also be set to run at different speeds by choosing different length of the time-steps and with which frequency these are sent.

6.2 Problem with skipped or delayed timers

During normal execution the engine tries to handle a timer on the exact time that it is scheduled for execution. This means that all timers will be executed at the right time, at least as long as the hardware of the target platform is powerful enough to manage. This situation changes when the time-updates are controlled by the editor, as a time-step could make the engine skip the execution time of one or more timers, which would mean that their execution would be delayed. This leads to a situation where the behavior of the engine would change when using the time control feature, something that were considered undesirable.

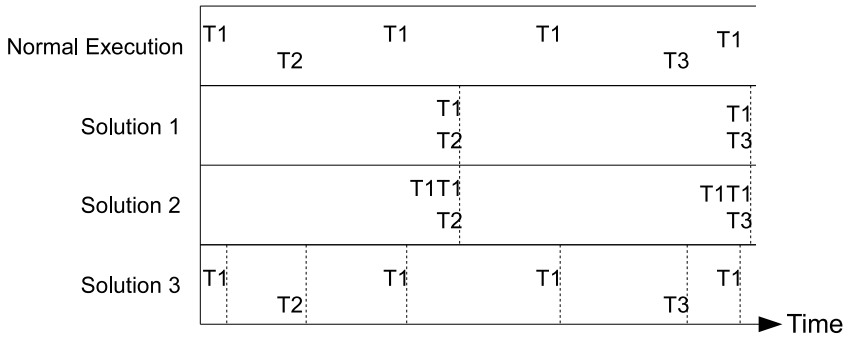

Figure 25: An illustration of the different solutions to the missed/delayed timers problem. Each dashed line in the figure represents one time-step and T# represents the execution time for a timer.

Another, similar, problem arises when using reoccurring timers. These timers are set to execute multiple times with a fixed time-delay and can, for example, be used to control the drawing of the display. Because of the way these timers are handled in the engine, a time-step that would be large enough to go by two or more executions of a single timer would only result in **one** timer execution. This means that this problem also changes the behavior of the engine in a substantional way.

As both of this problems where considered to be challenging, an investigation of different possible implementations were made. All of the solutions that were derived from this study are visualized in figure 25 and presented in the text below.

Solution 1

This solution uses a fixed time-step in each time-update. All timers that had a timeout during one time-step are executed at the end of the time-step. A timer that would have been executed more than once during this time-period will still only execute once. The main advantage with this solution is that it can be implemented in an easy way. It could also be realized with only minor changes to the engine-code, which would be desirable.

A big problem with this approach is that the execution will not exactly mimic the behaviour of normal execution. A timer which is scheduled at the beginning of a time-step will be delayed almost a complete time-step before it is executed. This means that the time-period between updates will have to be very low to get a good behaviour. This would in turn lead to a need for a lot of data-traffic. Another problem is that this solution would mean that the shortest period that will be obtainable would be the same as the used time-step.

Solution 2

This approach is similar to *solution 1* as it also uses a fixed time-step for each

time-update. The difference is that the engine keeps track of the number of times a timer would have been executed during each time-step, and executes them the right number of times at the end of each time-update. This would result in a behaviour that would better replicate the normal execution of the engine, as no timer-executions would be omitted. A problem with this approach is that all timer executions will congest and not be executed at the right time, which means that it would still not truly mimic the normal execution of the engine.

Solution 3a
The big difference between this solution and solutions *1* and *2* is that the time-step is not fixed. Instead the length of each time-step is set to match the time until the next timer is scheduled for execution. This means that the editor is trying to replicate the behavior that the engine uses during normal execution.

To be able to implement this solution the editor needs to know the time-stamp of the next timer scheduled for execution. This means that the engine needs to send a response for each time-update that is sent from the editor. This could be a problem as it will lead to a lot of traffic, and that the editor cannot continue to execute before it receives a response-message. This all boils down to a solution that is dependent on a good connection between the engine and editor.

The main advantage of this method is that each timer will be executed at the time it is scheduled, which means that it will mimic the behaviour of normal execution in a very good way. By using this approach the changes to the core parts of the engine is also kept very low.

Solution 3b
This approach is similar to that in solution *3a*, the main different consists in that the logic for the time-control is placed in the engine instead of the editor. This would minimize the data traffic needed, but means that a lot of changes have to be made to the core parts of the engine.

6.2.1 Design choice and motivation

Of the different possible solutions *3a* was the one that was implemented in the editor. The main reason for this was the fact that this, and *3b*, were the only solutions that were considered to mimic the behaviour of normal execution in a satisfactory way. Solution *3a* was chosen above *3b* as changes to the core part of the engine was deemed more undesirable than extra TCP/IP-communication.

6.3 Implementation of time control

Figure 26 shows the main design of the time-control implementation. The *TimeControlManager*-class is used to keep track of the current state of execution.

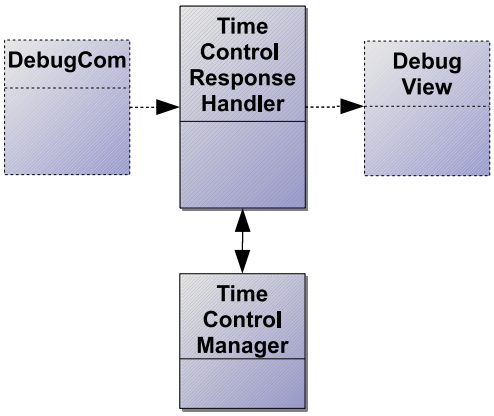

Figure 26: An overview of the time control implementation.

This is done by saving the current system time and the timestamp of the timer which is next in line for execution. This class also contains functions that are used to decide how long the next time-step will be, depending on what kind of run-mode is used.
Each time-step is sent from the *DebugView*-class, which has a separate thread that is in charge of sending time-updates to the engine. This thread uses the *TimeControlManager* to calculate the length of each new time-step and how often they will be sent.

6.3.1 Time resolution

As can be seen in figure 4 inputs from other devices are not read by the Populus system when the time is not moving forward. This means that large time-steps will result in a behavior where new inputs will only be read at the end of the time-step. This is undesirable as this does not mimic the actions that would occur during normal execution. Because of this the implementation is done in a way that each large time-step is divided into smaller steps, the size of which can be set by the user at runtime. By using small values of this setting the user can achieve more exact behavior, but this will also result in more TCP/IP traffic and puts higher demands on the used hardware.

6.4 A time control session

Figure 27 shows an detailed view of how a time-control session is executed. It should be noted that this session is synchronous as the editor needs to get a response from the engine before the next time-update is sent. The reason for this is twofold; firstly it is a good way to make sure that the engine is at the time displayed to the user. Secondly some time-control modes need to get

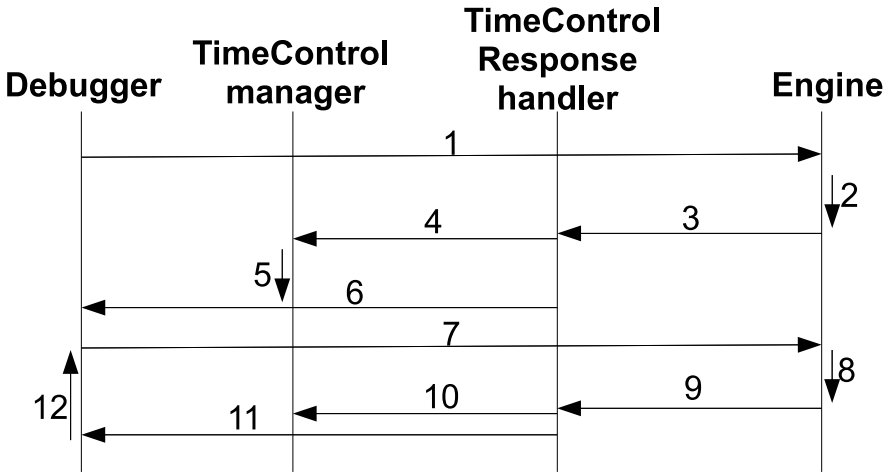

Figure 27: A flowchart of a time-control session.

information about the execution time of the next scheduled timer, which also is solved by using this method.

1. The user of the debugger sends the start control message.
2. The engine switch to use debug time instead of the time from the internal clock. The engine will pause until the next time step message arrives from the debugger.
3. An acknowledgement message is sent containing current system time and the time-stamp of the timer that is next to trigger.
4. The *TimeControlResponseHandler* creates a time control *TimeControl-Manager* containing the initial system time at start of the time control.
5. The *TimeControlManager* saves the system time as a reference of when the time control was started.
6. The *TimeControlResponseHandler* sends the *TimeControlManager* to the debugger as a acknowledgement that the engine is ready to receive time updates.
7. The debugger sends a increment time message to the engine.
8. The engine steps forward corresponding to the time step received from the debugger. All inputs will also be read.
9. A status message is sent from the engine containing the time until the next timer is scheduled for execution and the current system time.

10. The timer that was received is saved.
11. The debugger receives the time for the next timer execution.
12. The debugger will loop until the user stops the time control session.

7 Record and playback of a engine session

One of the desired functions that are specified in the project plan is the ability to record and playback a session in the engine. To be able to do this, all information that can make the engine change its state needs to be saved along with the time at which it is executed. This way, playback can be accomplished by setting the engine in a specified state and execute all the recorded commands at the right time.

7.1 Information that needs to be saved

In figure 4 it can be seen that the only events that can make the engine change its state are: *timers*, *buttons* or *TCP/IP-input*. As *timers* are either hard-coded to execute at different intervals, or triggered by input, they do not need to be saved to be able to achieve playback. This means that only *buttons* and *TCP/IP-input* was saved in a record-session.

7.2 Saving record information

To be able to recreate a recorded session the editor need to save information from the engine. As this process can be rather computation heavy and also can be done in many different ways, some time was spent researching different solutions. These solutions along with advantages and drawbacks will be presented in this section.

1. Reading data at time discrete intervals

The main idea behind this approach is that the editor request information from the engine with a fixed time delay. The engine then responds by sending all information that have been executed since the last request. This means that all events that have taken place during the time interval will be recorded as having the same time-stamp.

The larges advantage with this approach is that a lot of the logic can be put in the editor, which is the part of the code that is least critical from a performance perspective. The biggest disadvantage is the fact that the recording will not be identical to the session that is executed, as it will be dependent on the time interval. This means that the time resolution needs to be very low to accurately simulate a session.

2. The engine sends data on execution

In this solution each command that is executed in the engine is also sent to the editor. To be able to recreate a session each command will have an associated timestamp which corresponds to the execution time in the engine. Using this information playback can be achieved by sending each command from the engine according to the saved timestamp. This type of playback will simulate the execution of the recorded session closely, even if some small timing issues

could arise because of delay in the data transfer process.

One disadvantage with this method is the fact that modifications will be needed in core parts of the engine, which could be problematic as this is the most performance critical region of the system. The needed amount of transmitted data is very low in this solution, as each piece of information will only be sent once, when it is executed.

3a. Save complete states of the engine using a wrapper-class
Using this approach the engine will be placed in a wrapper-class which should make it possible to save all the internal information from the engine as a chunk of data. Each chunk would contain all information needed to recreate the state of the engine. This means that a recording can be made by saving states of the engine at a specified time interval. One big advantage with this solution is that the code in the engine could be left more or less untouched. The implementation of a wrapper-class would on the other hand lead to a big structural change of the engine, which could be deemed undesirable. This could also mean that the performance of the engine would suffer, as the extra functions introduced by the wrapper class would have to be executed even when no recording was done.

The time resolution would be needed to be very high to get a good playback using this solution. The reason for this is that it would be impossible to read information in-between states, which means that a high delay would lead to bumpy playback. This also means that the playback will not perfectly mimic the behavior of the engine, as the behavior between states is unknown.

A high refresh rate combined with the need to save the complete state of the engine also means that the amount of data traffic will be extremely high using this solution. This is mainly due to the fact that this method will have to save all information in each update, and cannot focus on changes.

3b. Save complete states of the engine using class-serialization
This method is tightly linked to the solution in *3a*; the difference is mainly in how the engine states are saved. In this solution all classes needed to recreate an engine state would have to implement a function that saves all important data in a binary form. The result from the serialization of all classes could then be sent to the editor and saved. If this is done at the same time instance in all key classes this information would be enough to make a recording.

This solution also suffers from many of the same drawbacks that is described in solution *3a*; the behavior between each saved state is unknown, and a high refresh-rate is needed to achieve smooth playback. The needed data transfer will still be high, but substantially lower than in solution *3a*, as only useful information will be sent.

The biggest difference between solution *3a* and *3b* is due to their structural

difference. *3a* leaves the engine-code intact and changes the way the engine is run. *3b* instead makes changes to the internal classes of the engine while not changing the way the execution is done.

7.2.1 Design choice and motivation

Solution *2* was the solution that was chosen to be implemented in the project. The main reason for this is the fact that this is the only method that really recreates the behavior of the engine. To minimizing the data transfer was not considered a primary concern in the evaluation of the different solutions, but the fact that solution *2* needs relatively low information exchange was also considered an advantage of this design. This solution was also considered to be a good tradeoff between complexity and performance.

7.3 Implementation of recording

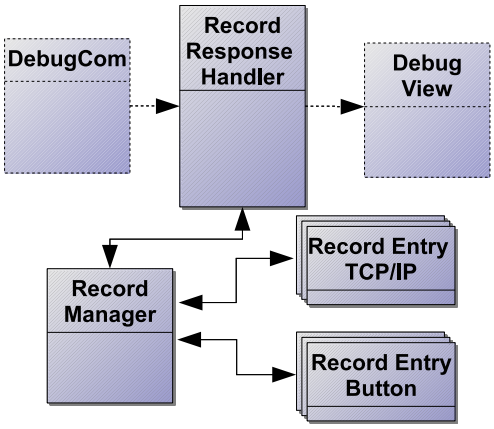

Figure 28: The overall design of the record-implementation in the editor.

Figure 28 shows the general design of the record-interface in the editor. When in recording mode, the engine will send a message for each new event that takes place. These messages contain information regarding what type of message it is, and a time-stamp. This information is sent to the *RecordManager*, which creates a instance of either the *RecordEventTCPIP*-class or the *RecordEventButton* -class, depending on the event type. Each event is saved in a list, which corresponds to a single recording. All of these classes implements the java interface *Serializable*[1], which makes them easy to save to a binary file.
When the editor is in playback-mode the editor takes control of the engine-time in the same way as described in section *6*. All recorded events are sent to the engine using a separate thread. This thread also sends time-update messages

to make the engine follow the execution of the recording. In this mode the *RecordManager*-class is used to keep track of different information regarding the playback.

7.4 A recording session

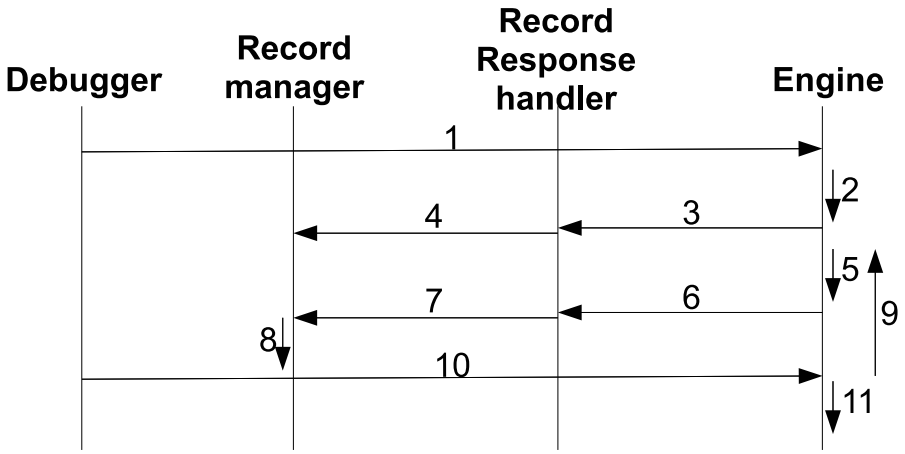

Figure 29: An overview of a recording session.

The main idea of a recording session is that the editor notifies the engine when it wants to start or stop recording. During the session the engine then sends all relevant events that takes place to the editor, where it is saved in the *RecordManager*. A detailed flow-chart of how this is done is shown in figure 29

1. A start record message is sent to the engine.
2. The engine sets itself into record mode.
3. The engine sends a start message for the recording.
4. The *RecordResponseHandler* creates a *RecordManager*.
5. When a button- or a TCP/IP-event occur a message is generated in the engine, this message also contains a time-stamp.
6. The message is sent to the editor.
7. The message is deserialized and sent to the *RecordManager*.
8. The record is added to the list of record events.
9. The send process is repeated for each event that take place during the record session.

10. When the user stops the recording a stop record message is sent to the engine.

11. The engine exits record mode and resumes normal execution.

7.5 Playback of a recorded session

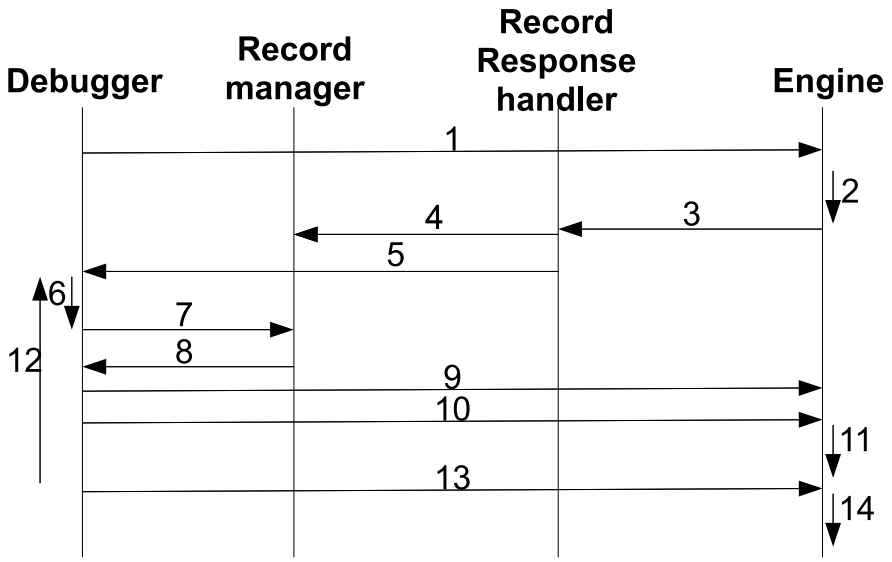

Figure 30: An overview of the playback of a recorded session.

Figure 30 shows the overall design of the playback of a recorded session. The editor goes through the list of all events that have been saved and sends them to the engine at the right time.

1. A start playback message is sent to the engine.

2. The engine goes into playback mode, which means that the time is controlled by the editor.

3. The engine sends a message that signals that it is ready to receive playback events.

4. The *RecordManager* is set to start a new playback session.

5. The updated *RecordManager* is sent to the debugger.

6. The playback thread is started.

7. The debugger checks the time until the next event will occur.

8. The *RecordManager* send a time update to the debugger.
9. The recorded event is sent to the engine.
10. The editor increases the time in the engine.
11. The engine handles the recorded event and increases its time to when the next event will occur.
12. The editor will loop until the playback is finished or the user aborts the playback.
13. The editor sends a stop playback message to the engine.
14. The engine exits the playback mode and resumes normal execution.

8 Results and Discussion

The main goal of this project was to implement an interface that would give a user of the Populus editor a set of tools to that could be used to debug the HMI that they are developing. This was considered important as it would make it much easier to introduce a new user to the Populus program, especially if they did not have a deep understanding about the system beforehand.

In the planning stages of the project work the main features that should be implemented where decided (see section 2). Overall the project must be deemed successful, as most of these features where implemented in a satisfactory way. During the course of the project some of these sub-goals where changed and refined, as more information about the system were gathered and tests were made. This section of the report will discuss each of the sub-goals in more detail and also give some insight in how continued work in this area could be done.

8.1 Port the existing Trace-program to Java

The main goal with this feature was to transfer the functionality of an existing C++-program to the java environment of the Populus editor. This was the first programming-task that were dealt with in the project work, and in many ways used to give a better understanding of the Populus system. The port was successful, as all functionality from the existing program was replicated in the Populus editor. The conversion to a Java-program also resulted in some additional benefits, as the Java-port was less likely to suffer problem with TCP/IP-buffer overflow compared to the C++-version.

8.2 Reading data from the engine

This functionality was one of the most important, as it was considered a key factor to get a feeling of an ordinary debugger. The ability to get an understanding about what takes place in the Populus engine, using only the editor, was very important. The main difficulty with this task was to be able to read information without interfere with the exciting code. This was solved by using the debug-interface described in section *3.3.1* This feature was fully implemented for all relevant data.

8.3 Setting data in the engine

The main idea behind this feature was to be able to read a value from the engine and then change it and continue execution. This functionality was implemented for one data-type, but was considered to be too unreliable to be used in the final product. The main reason for this was due to the fact that it was hard to guarantee that forcing data-changes would not interfere with other parts of the program-code. To be able to implement this feature core parts of the

engines execution would have to be changed, which were considered to be to time demanding to be in the scope of the project work.

8.4 Controlling the execution speed of the engine

The most important part of this functionality was to be able to stop the execution of the engine, and then start it again at will. This is a very useful function to have when using a debugger as it lets the user stop the execution at critical moments, and take a closer look on what is happening. Functionality to step between key-events and to take fixed time-steps was also implemented. Above this the ability to run the engine at a fixed percentage of its normal execution speed was realized. All of these features worked as intended.

8.5 Recording a session

The goal with this feature is to be able to record an interesting session of engine execution. This could be interesting if a user of the Populus editor have an especially important chain of events that he or she wants to take a closer look at. It could also be a convenient way of making presentation demos, and work as a substitute for recorded videos in these situations.

A basic version of the recording feature was successfully implemented during the project work. The limitations with the implemented solution mainly come down to the fact that there is no way of saving complete states of the engine. This means that it is impossible to go backwards in a recording session. Above this, it also means that the engine needs to be manually put into the same state as it had at the beginning of a recorded session, to achieve the right behavior. The reason for not implementing the ability to save complete engine-states was that it was considered to demand too much work to be performed during the timescale of this project.

8.6 Further work

The focus of the project has been centered on getting the base functionality working. This means that not much time have been spent on making the graphical presentation to the user more appealing. Because of this, the usability of the debug-functions could be greatly increased by spending time on this area.

To make the recording-function really useful the ability to save engine-states would be have to be implemented. Even though this would require a big work effort, it would almost be required to make this feature live up to its full potential.

References

[1] Java api 6 interface serializable. `http://java.sun.com/javase/6/docs/api/java/io/Serializable.html`.

[2] Mecel AB. Hmi development. `http://www.mecel.se/products/mecel-populus`.

[3] Jianlin Shi Martin Trngren Karl-Erik rzen, Jad El-Khoury. *Real-Time Control Systems*. Department of Automatic Control Lund Institute of Technology, Department of Machine Design The Royal Institute of Technology, 2008.

[4] Jan Skansholm. *Java Direkt med Swing*. 5th edition, 2005.