CHALMERS

Dynamic Protocol Analysis Tool for Trains

Master of Science Thesis in the Programme Computer Science - Networks and Distributed Systems

Rickard Gustafsson
Jimmie Högklint

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, August 2011

Dynamic Protocol Analysis Tool for Trains

Rickard Gustafsson
Jimmie Högklint

Examiner: Arne Linde

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 August 2011

Abstract

Bombardier is in the process of developing a new solution for controlling trains which act autonomously. The central unit, Train Control Center(TCC), controls the vehicles via TCP/IP. When developing such as system a great amount of time has to be reserved for testing the product. When testing the vehicle software the TCC is unfit to perform and repeat desired test cases. A TCC simulator, TCCSim, was developed instead. The simulator will also be helpful in the beginning of the development phase were nothing to test against exists. TCCSim is able to dynamically define messages which can be composed and sent to the vehicle as well as inject faults into the messages. The vehicle response is logged and can be analysed. Test cases are easily reperformed with the ability to script the behaviour of TCCSim.

Sammanfattning

Bombardier är i utvecklingsfasen av en ny lösning för att kontrollera tåg som agerar autonomt. Den centrala enheten, Train Control Center(TCC), kontrollerar fordonen via TCP/IP. När ett sådant system utvecklas måste mycket tid läggas på att testa systemet. När tester gällande mjukvaran i fordonen utförs lämpar sig inte TCC att utföra dessa och återskapa testfallen. Istället utvecklades en TCC-simulator, TCCSim. Simulatorn är även praktisk i början av utvecklingsfasen då det inte existerar någonting annat att testa mot. TCCSim är kapabel att dynamiskt definiera meddelanden som kan bli sammansatta och skickade till fordornet. I meddelandena kan även fel injiceras. Fordornets svar loggas och kan bli analyserat. Testfall är lätta att återskapa med möjligheten att skripta TCCSims beteende.

Acknowledgements

We would like to thank our supervisor and examiner at Chalmers University of Technology, Arne Linde, and our supervisors at Bombardier Transportations, Lars Antbäck and Jan Kiessling, for all the help and feedback provided throughout the project. We would also like to thank Bo Hermansson at Bombardier Transportation for providing an Com-to-IP converter, making it possible for us to test our product with the old test system.

Rickard Gustafsson and Jimmie Högklint, Gothenburg 11/08/01

Contents

List of Abbreviations

BCC	Block Check Code
CRC	Cyclic Redundancy Check
DLL	Dynamic-Link Library
GUI	Graphical User Interface
IP	Internet Protocol
LINQ	Language INtegrated Query
PInvoke	Platform Invocation Services
STDOUT	Standard output, text presented from console application
TCC	Train Control Center
TCCSim	Train Control Center Simulator
TCP	Transmission Control Protocol

1

Introduction

Bombardier is a cutting edge company[1] in the field of transportation with trains and airplanes. One area is autonomous train control. In the process of developing their product for simultaneously controlling trains a significant amount of time has to be put on testing the products. This is to make sure the product executes in a safe manner because of the high safety standards regarding trains in general. It is unpractical to perform all these test cases with the actual product thus a simulator can be of help to ease the process.

The central system, called Train Control Center(TCC), is controlling the vehicles and the communication is today via radio link, using serial communication. For testing and development purposes there exists a simulator software. The serial communication is being exchanged with TCP/IP, which means that the central system simulator has to be updated as well. The existing simulator is incomplete and lacks application handiness which means that developing a new simulator is the best option at this point.

1.1 Description of Task

The task for this project is to produce the TCC simulator, TCCSim. Apart from the high level goals that can be found in Bombardier's project description, the process of developing the simulator includes full specification and implementation.

The purpose of the simulation software is to assist the development and verification of

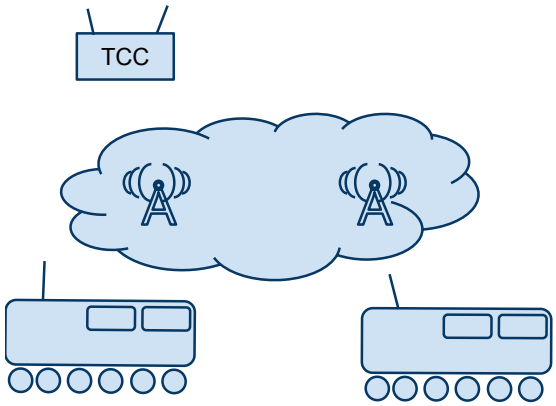

Figure 1.1: TCC and trains communicating

the vehicle software and the communication protocol used between the TCC and the vehicles.

1.2 Project Goals

The main interface should provide the main functionality such as sending messages, see message exchanges between TCCSim and the vehicle and closer analysing of received and sent messages. This will be presented in a structured and reviewable manner.

There should be an interface which makes it easy to handle and create new messages. It should provide the possibility to save these messages as a file which makes them easy to reuse and transfer between computers. It should be possible to create and send new messages without saving them. The interface has to be intuitive and allow fast and easy input of data for different message types.

This project uses TCP/IP to communicate with a vehicle. Since this might change in the future, the communication module should be easily exchangeable.

To truly test the robustness of the protocol and the vehicle software, faults has to be injected into the messages sent from TCCSim. There should be circumstantial possibilities to put these areas under test by injecting faults.

By implementing scripting ability into the application even more extensible test cases can be performed. The goal is to let a user of TCCSim easily write scripts which can be loaded and executed. The abilities of the script should be the same as the rest of the application.

2

Tools and Specification

2.1 XML - eXtensible Markup Language

XML[2] is convenient when storing a moderate amount of data. It handles versatile and complex data structures with ease and it is easy for both humans and computers to understand and manipulate the content. The XML files are also easily transfered between computers.

2.2 C#

The environment the application is to operate in is the Microsoft Windows platform. Since Microsoft supplies a very good development environment for Windows in Visual Studio this is the natural choice for quick and easy development. Some language supported by Visual Studio has to be chosen. C# is an obvious candidate since it was developed with XML in mind[3]. The integration of XML in C#'s standard libraries makes it effortless and transparent working with XML-formated data. The fact that C# also has automatic memory management[4] and is type secure[5] helps the development process. C# is designed by Microsoft based on development of object oriented applications for Microsoft Windows using the .NET framework, this gives a language that is uncluttered and with strong integration to that API.

2.3 MVC/Events

To get a good structure and maintainability of the code the design pattern Model-View-Controller(MVC) is used. It divides the different parts of the program in a reusable and a reviewable manner. Since C# is used, delegates and events can be used to pass information around the application in a MVC type fashion. Each event has an object attached which contains information regarding the type of event. This object inherits the existing class EventArgs. Since C# has co- and contra-variant type system the event handlers takes an object of type EventArgs as input but is then able to convert to a more narrow type. This way one event handler can handle many different type of events by checking the actual type of the incoming object.

2.4 IronPython

To enable scripting abilities to the application a second programming language has to be chosen and embedded into the C# code. IronPython[6] is a good fit for this. It is a Python implementation in the .NET platform which makes IronPython very easy to integrate into the application. All implemented Python features, the .NET libraries as well as TCCSim's functionality will be at the user's disposal writing the script.

2.5 LINQ

LINQ, Language INtegrated Query, was released with version 3.5 of the .NET Framework. With LINQ the programmer gets a SQL-query like interface to both in memory data and relational databases. The in memory data that can be accessed is arrays, lists, XML and data sets[7]. LINQ gives an uniform method to access unordered data that is transparent, not regarding which kind of data that is to be accessed. This is a very powerful tool that makes programs easy to maintain because only small changes has to be done to change the whole data source of the program.

2.6 Application

In the development of this new autonomous system both the stationary and trainborne subsystems have to be developed at the same time since the transportation medium is being exchanged. Both parts are dependent on each other to verify the functionality of

the system. Because of this, some other equipment is required for both parts during the development phase. The needs differs a bit from the different parts. The target for this project is to develop the solution chosen for the trainborne system.

The trainborne system needs something that sends messages so it can be tested. The more comprehensive this system is that sends the messages the better the trainborne system can be tested. A simulator that can fully emulate the behavior of the real stationary system were chosen. A simulator that can act accordingly to instructions given from the development team. The simulators main functionality should be to create message and sending them as well as logging all sent and received messages.

The previous simulator used for this task had a cluttered and not very intuitive user interface. The possibility to analyse transfered messages were also limited and not user friendly.

2.7 Application Protocol

The application protocol which is used when the TCC is communicating with a vehicle is a polling protocol. TCC starts the communication by polling the vehicle with a default message. If the vehicle responds it means that poll was successful. The polling will continue with a fixed interval until the communication is shut down. The vehicle is essentially a finite state machine which is controlled by the polling messages.

In the process of controlling the vehicles there is no demand of realtime operations. The protocol specifies that a tolerance measured in seconds is more than enough. Antbäck[1] claims there are even areas which are known to cause radio shadow. This means that the vehicle should follow already given directions until the network connection is restored.

Every message, inbound or outbound, starts with a safety header, as shown in Figure 2.1. All in all it contains seven fields for added safety and error detection. It starts with a value called <STX> which is a message start byte. The next field is a vehicle ID which gives what vehicle it was either received from or sent to. The subsequent field contains the payload length of the message.

The next two fields are time stamp values. The first time stamp, T_SENDER, is the time when the message was sent. Its value should be greater for each message received. It should, however, also support roll-over meaning when the field passes the value 65 535, which is the highest number the two byte field can contain, it should start over at

[1]Lars Antbäck at Bombardier Transportation. Interviewed the 6th of April, 2011

zero. The second time stamp field, T_REF, is a reference time stamp. When a message is received the reference time stamp should contain the last time stamp that was sent by the receiving party. This is to protect against data obsolescence since the different subsystems are not time synchronized.

Following the reference time stamp is the payload of the message. The payload has a message type which defines what information can be expected. The information is divided into fields and blocks. The fields are obligatory for that message type while the blocks most often is optional and may appear an arbitrary number of times. The blocks have a block type and also contain fields.

At the end of the message a 32 bit Cyclic Redundancy Check(CRC) is appended. It is used to detect errors during transmission. Another 8 bit CRC calculation called Block Check Code(BCC) is appended to the very end of the message as an extra check for transmission errors.

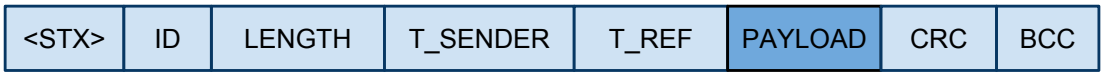

Figure 2.1: Safety layer fields in light blue

3

TCCSim - Train Control Center Simulator

TCCSim consists of a few different parts. This chapter will first give an overview of the parts and then more in depth explanations.

3.1 System Overview

A simplified overview of the complete system can be seen in Figure 3.1. TCCSim is visualized using green color. The definition of the application protocol, saved messages, and scripts are stored in the file system which is visualized using the color yellow. Visualized with color red is the vehicle which TCCSim uses TCP/IP to communicate with.

TCCSim is developed in a MVC fashion. The controller is, however, not displayed in the figure since it merely is a way of distributing tasks from user interactions. The user can interact with the model via three different interfaces. Most of the controls regarding the communication can be found in the main view. This is also where information such as connection status and received messages is displayed. The message creation view is more tightly coupled with the message specification module. This is where messages are created or modified and can also be sent to the vehicle. The script interface is more or less the main view and the message creation view combined. Even though a script cannot create a message from scratch, they can be loaded and modified as well as sent. Received messages are also available for a script to analyse.

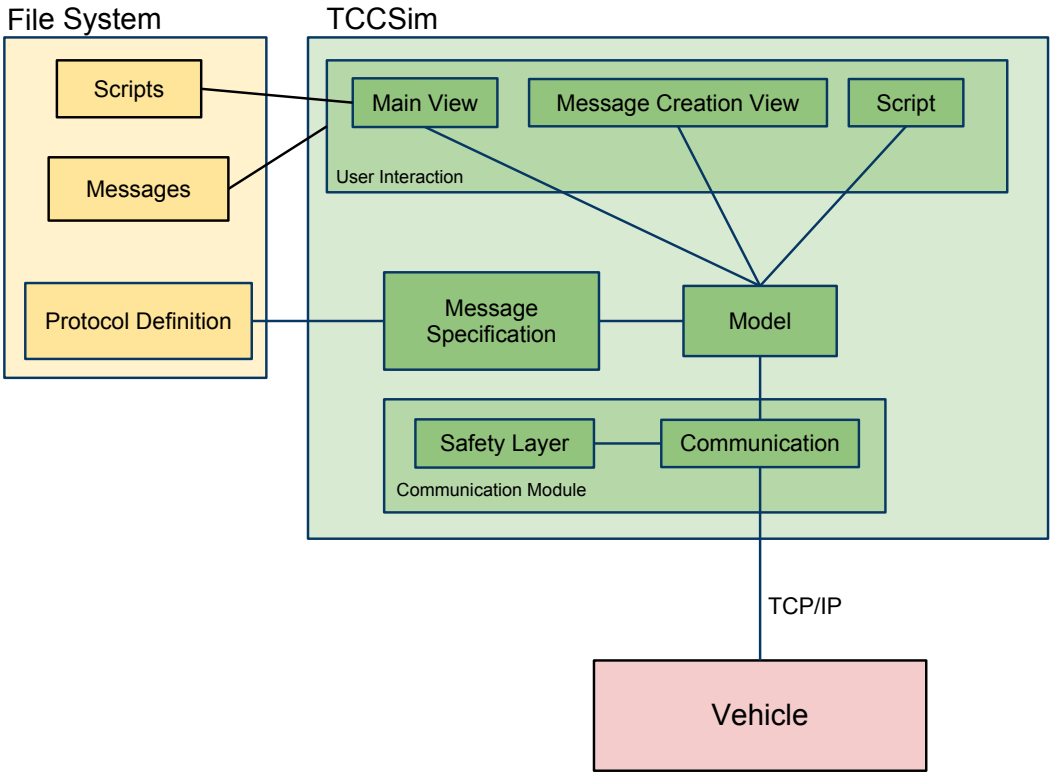

Figure 3.1: System Overview

The message specification module understands the external protocol definitions and is able to translate them for the application to use. It can also translate messages to a more readable format using the definitions. The definitions are stored using XML files located on the local file system.

The communication module is built as a Dynamic-Link Library(DLL) since it is likely that TCP/IP will not be used in other future projects and should therefor be easy to exchange. As seen in Figure 3.1 the safety layer is not actually used as a layer like in a general protocol stack. This is because of TCCSim's need for the safety header information even after the message has been delivered to the application and the fact that TCCSim does not use the safety header to improve safety in any way.

3.2 TCCSim's Main View

The main user interface in TCCSim is called the main view and will be described in this section.

3.2.1 Transfer Log

All messages that TCCSim sends or receives are stored for observation. They are displayed in the main view of the application in the order they are transfered. Since TCCSim always sends a message with a fixed interval, most often every three seconds, the number of messages displayed makes it hard to see the outline of the conversation. Therefor a feature for filtering what type of messages should and should not be displayed was added. Of course a lot of information can be neglected this way so a status table was also added. The status table displays momentary information of the most common type of message received from a vehicle. This status table in combination with the message filter makes a great match since only the most important information is displayed if desired.

If the transfer log is needed at a later time it can be saved to a file and loaded into TCCSim again. The format of the saved log is very simple. One message per row. Each row starts with a time stamp, followed by the message in a byte format and then four values telling if the message was sent by TCCSim, if it contains an id error, if it contains a sender time stamp error and reference time stamp error.

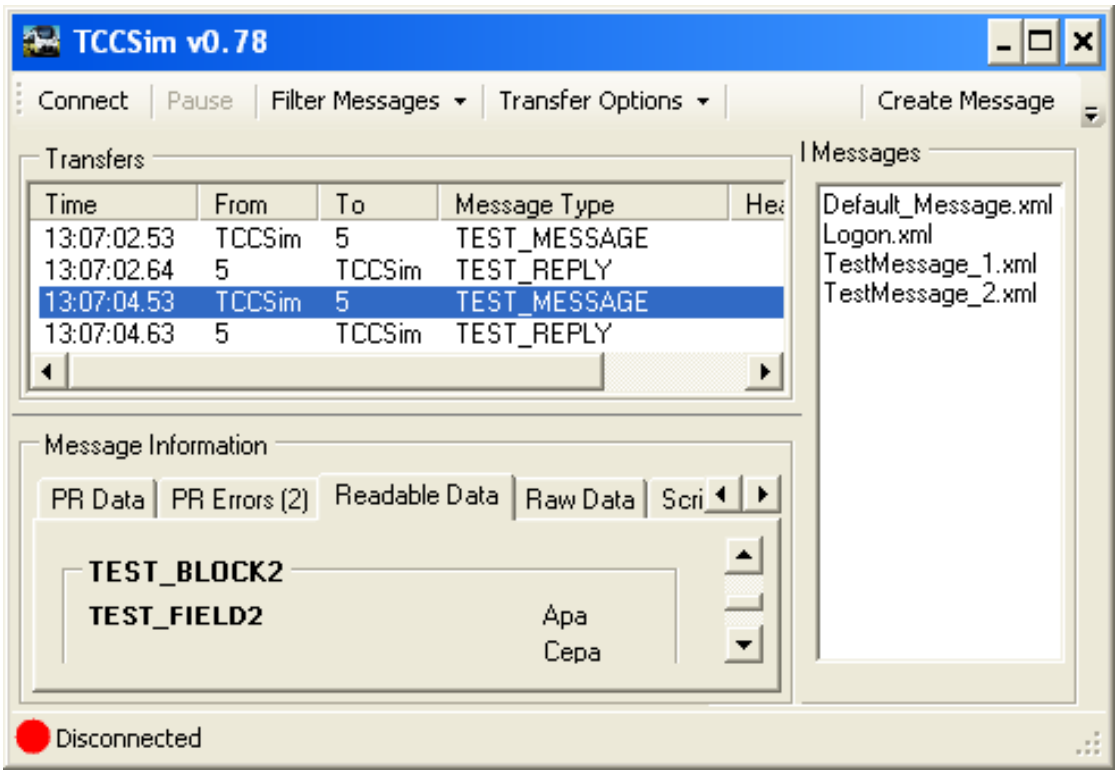

Figure 3.2: Main view with transfer log at top, message information at bottom, and saved messages to the right.

3.2.2 Display information

Apart from displaying status information in a table, each message in the transfer log can be selected and thereby display the message's full information. The message's fields are displayed with name and value in a table followed by the message's blocks with the fields grouped together. The raw data is also displayed in hexadecimal form which is useful if TCCSim cannot parse the selected message i.e. not display the field names and values.

3.3 Dynamic Messages

One of TCCSim's greatest advantages is the ability to alter the protocol specification without the need to rebuild the application. This was not a feature TCCSim was intended to handle in the initial goals. It was just a proposed desire from the employer during the initial design cycles. The possibility to achieve this was investigated and later realised.

This feature enables the user to make changes to the protocol that just needs a restart of the application to take effect. This allows TCCSim to be able to handle more domains than it was originally intended to. How the dynamic message definitions is handled are described in this chapter.

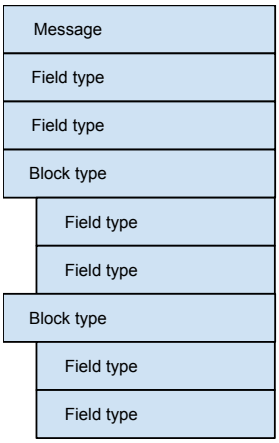

Figure 3.3: Message structure

3.3.1 Message payload structure

A message is build from a range of field types and maybe block types, as seen in Figure 3.3. A field type is an entry of a single value which can be a signed or unsigned integer or a text string, with restrictions of size and values defined. The smallest size a field type can have is one byte. A field type is a mandatory entry in a message. A block type is a collection of field types, the difference is that the collection of field types in a block type are optional in contrast to field types which are mandatory.

In TCCSim, messages are totally configurable by the possibility to combine field and block types however the user wants. With the ability to control the content of messages down to byte level by just specifying the structure this renders a very dynamical environment. Giving the user a powerful testing environment.

3.3.2 Creating/Defining Messages

To achieve easy modifiable messages all message types, block types, and field types are stored in XML files. One file holding all field types, one for all block types, and one for all message types describing which field and block types each message type contains. To parse these files the .NET functionality LINQ is used which gives a queryable interface to various data structures.

Generating message creation view

With the information in these files the form for filling the values for the different field types in the messages is created. By looking at what different fields and blocks a message consists of, various Controls for input are generated for the desired message type. The Controls are setup according to what kind of data type they are suppose to represent. Minimum and maximum values are also considered during this phase to help the user by not allowing values out of bounds.

When a field type of signed or unsigned integer is found, first it is checked if that field type has a description for each value. If so, then a list of descriptions connected to each value is created and added to a ComboBox. Otherwise a NumericUpDown Control is created with maximum and minimum values for the field type as bounds. In some cases it is desirable for the value to be presented in base 16. For this case a custom Control class has been created, HexNumericUpdown. This is a modified NumericUpDown Control that displays values with the prefix 0x to make it clear for the user that values are presented in base 16.

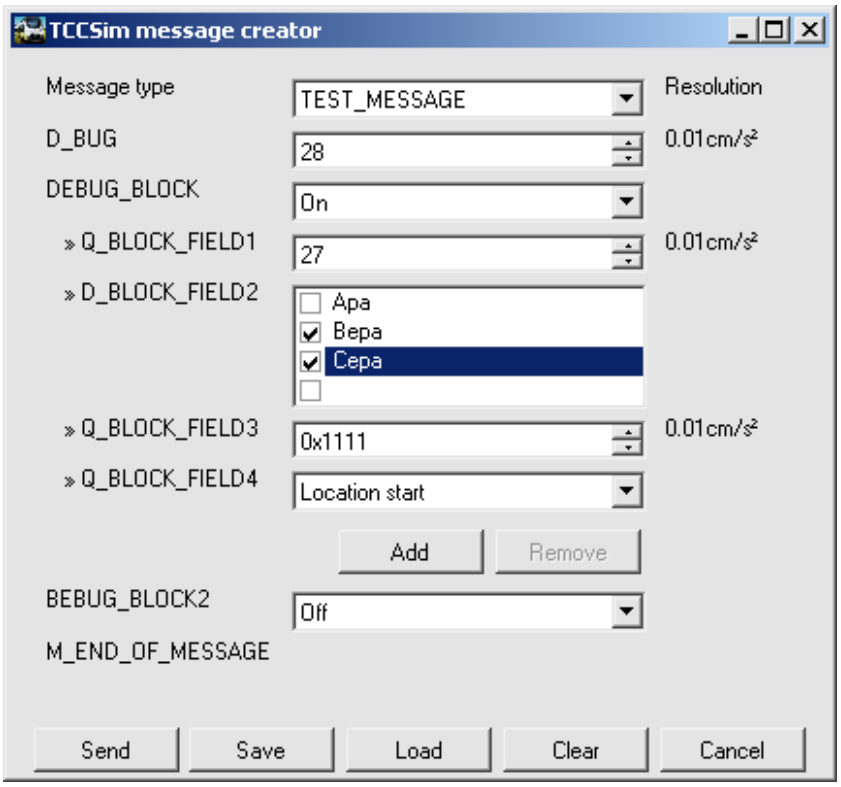

Figure 3.4: Message creation interface

In some fields an unsigned integer is used where each bit has its own description. In this case a list for each bit and its description is generated. This list is presented with a CheckedListBox so that the user easily can see which instructions that are being included in the message.

Besides different integers, text strings can also occur. In this case a TextBox with the max input length set to the length of the field type.

For each block type in the message type a control for enabling and disabling the block is created. This is just a ComboBox with the description On and Off. Here a ComboBox is used since it allows for easy manipulation by the user from the keyboard, which have been on of the major design goals for the message creation view. When Controls are created they are added to a TableLayoutPanel which helps positioning the Controls dynamically so no own calculations of positioning have to be performed. This is extra helpful when working with blocks since this is just a collection of fields that is to be inserted in the message. Some blocks may also be inserted multiple times. In this case buttons for adding and removing a block entry is added after each block entry so entries can be added and removed in which ever order wanted. The ability to insert fields breaks the tab index of the form, to solve this all Controls interesting for tab

index is stored in an ArrayList so tab index can easily be updated. When a Control is inserted in the TableLayoutPanel it is also inserted in corresponding position in the ArrayList and the tab index of all Controls after it in the ArrayList is updated. The reason that the collection of Controls in the TableLayoutPanel cannot be used for this is that there Controls are ordered in the way they were added to the TableLayoutPanel. This ArrayList is also used when the values for the message is to be fetched for either storing or sending.

Storing and sending messages

Regardless if a message is to be stored or sent, all values are read from the Controls to a XML structure in an XElement. The structure created is very simple, just a tag for each field with its name and value as an attribute. The start of a new block entry is marked with a tag that tells that here is a start of a block entry and the block id and name as attribute.

The data is collected from the Controls by iterating the ArrayList used to fix the tab indexing since all Controls are ordered in the way they should be collected. How each value should be collected is determined by the type of the Control. In the case of NumericUpDown Controls, the entered value is just read. With ComboBox Controls, only the selected index is fetched and stored in the tag. The only case where something special has to be done is with the CheckedListBox Controls. In this case each bit selected gets its own tag in the parent tag describing the name of the field. This to keep it more descriptive.

When the XElement of all Control values have been generated it can easily be saved to file with the method XElement.Save(filename). This gives a file with a structure and representation that is easy for a human to read and manipulate if needed. The same XElement can also be passed to the method Message.XMLtoByte which packs the data in a byte string used to send out to the trainborne system.

The message data is started with the message type id and after that all fields are entered in order. Block type id and fields are added if existing. When doing this, all integers and strings are converted to byte strings with the help of the class BitConverter. Since all data in the outgoing byte string should be in network byte order, the integer byte strings has to be reversed before added to outgoing byte string. If the value being added to the outgoing byte string is a bit mask it has to be converted from the XML bit mask representation to an unsigned integer. This is done by iterating all selected bits and adding them as shown in Listing 3.1.

```
uint value = 0;
foreach (bit in bits)
  value += (1 << bit);
```

Listing 3.1: Calculate bitmask value

When a signed or unsigned integer is to be converted to a byte string and added to outgoing byte string the integer value is handled as if it was a 64 bit long integer to ensure that it is parsed to a long enough byte string, then just the interesting bytes are picked and added to outgoing byte string.

Another challenge that emerged was the presence of 3 byte signed integers. Since C# does not have a built in data type to handle this a regular 4 byte integer had to be used with some calculations for the signing as shown in Listing 3.2.

```
int24 = parse32bitvalue(value_string)
// Take interesting bytes.
int24 &= 0xffffff
// Test for negative.
if ((int24 & 0x800000) != 0)
   // Set negative
   int24 |= 0xff000000
fi
```

Listing 3.2: Calculating 3 byte signed integer

XML structure

```
<MessageDescriptions>
  <Message type="Message name" value="1">
    <StationaryMessage>True</StationaryMessage>
    <Filtered>False</Filtered>
    <FieldType Name="Field1" />
    <FieldType Name="Field2" />
    <BlockType Name="Block1" Numeric="1">
      <Max>1</Max>
    </BlockType>
    <FieldType Name="End" />
  </Message>
</MessageDescriptions>
```

Listing 3.3: Message XML structure

```
<BlockDescriptions>
  <Block type="Block1" value="1">
    <FieldType Name="Field3"/>
    <FieldType Name="Field4"/>
    <FieldType Name="Field5" />
  </Block>
</BlockDescriptions>
```

Listing 3.4: Block XML structure

```
<FieldDescriptions>
  <FieldDescription type="Field1">
    <Detail>Field1 description</Detail>
    <Length>2</Length>
    <Min>1</Min>
    <Max>65535</Max>
    <Resolution></Resolution>
    <Format>BITMASK</Format>
    <Special>
      <Bits>
        <Bit value="0">Bit1 description</Bit>
        <Bit value="1">Bit2 description</Bit>
      </Bits>
    </Special>
    <Default></Default>
  </FieldDescription>
</FieldDescriptions>
```

Listing 3.5: Field XML structure

3.3.3 View Messages

TCCSim can translate a byte string of a message into a human readable format. This is done by fetching a message description from XML, either by knowing which message type or by reading the message type from the byte string. The message description is stored in an XElement. The case where the message type is know is when messages sent from TCCSim is parsed. This is used for faulty message definitions in the case of fault injection in the message payload.

The fetched message description is used to know which field types that should be found in the message byte string and which block types there is a possibility to find.

The parsing begins with reading all the message safety header fields. Each value is stored in a FieldType object. After all header values is fetched the message description is used to fetch all field types in the message. Each field type is stored in a FieldType object.

If the message type can have any block types the payload is checked for this. Each block type with its fields are represented by a BlockType object. Each field type in the block is stored as a FieldType object in the ArrayList of the BlockType. Which field types that a block type consist of is looked up by reading the block id from the message byte string. That value is then used to look up the block name in the message description which in turn is used to fetch the block description from the XML file describing all block types. The message is searched for block types until an end of message marker is found.

3.4 Communication Module

The communication module is responsible for all communication with the vehicle. The module is fed, by the main part of the application, with messages to send and it replies with messages that is receives from the vehicle. The outgoing messages also passes through a safety layer which is included in the module. It also informs if any error occurs.

Since TCCSim is not dependant on what type of communication is being used the communication module should be a stand-alone part of the application. It is therefor compiled as an application extension and included from a DLL file for the rest of the application to use.

This project uses TCP/IP as mean for external communication and this section of the report describes how the module was designed. This includes choices made regarding the application protocol, ways of making the module stand-alone and how the safety layer is used.

3.4.1 Regarding the protocol

The application protocol that is specified for this system is a polling protocol meaning that TCCSim will poll the vehicle with a fixed interval and wait for a response in between these polls. The module therefor has to execute for as long as it is active. Knowing that this is a user interactive application, having the process executing indefinitely will make the GUI totally unresponsive. Instead the communication will be executed in a different thread and report progress to the main thread.

Since the other modules of the application is passing around information using C#'s events, it would be a good thing if the communication thread and the main thread were joint in the same manner. This would allow the communication between the threads

to be completely asynchronous. Events themselves have however no recollection of in what thread an event should or should not be executed. Firing an event from one thread means that the same thread will execute the event handler registered to that event. C#, though, have the ability to marshal events between threads and this becomes very easy with the existing class BackgroundWorker. It is built to pass information between a user interface thread and a thread doing lengthy executions in the background. Information passed from the communication thread includes the actual messages being delivered, errors regarding faulty connection data, errors regarding the connection or the lack of being able to establish a connection.

The old system, which is being replaced, is using an application protocol developed for serial communication. The new system's protocol will be very similar to the old protocol since serial communication might be used again in the future. This means that this module will have to handle the incoming data as if it was transfered using serial communication. This includes searching through the incoming stream of bytes for the message start value "<STX>" and reckon with receiving fragmented messages. This approach also eases testing since old testing equipment can be used to test TCCSim when new hardware is not available.

3.4.2 Stand-alone Module

Since the system simulated by TCCSim might change in the future this communication module has to be as little attached to the rest of the application as possible. Only the application protocol is certain to remain while all lower level protocols might be exchanged. Therefor the module is built as a separate DLL file, which means that it is like any other external program, unable to depend on functions in the application. The module contains three main classes. The communication class, an abstract class which the communication class inherits and the safety layer class. The abstract and safety class should not need to be changed when changing the communication architecture unless, of course, there are changes regarding the safety layer. The abstract class is created to make the structure and the development of a new communication class easier and does not have anything to do with the actual communication architecture. Alongside these main classes there are five classes used to pass information when events are fired. It is hard to know in advance, but these five classes inheriting C#'s EventArgs class should be enough even in future development. They are used to deliver messages, pass information on the connection status and to report unknown errors.

Making the module stand-alone means also making the user interface independent of what type of communication architecture being used. This includes input of connection data. In this project that means to input a destination IP address and TCP port. Two fields to input this data might be enough and it might not when TCP/IP is being

exchanged for something else. This is why the two field was merged to a single field. In this case it means giving the data in the form "<IP address>:<TCP port>". The communication class will have to parse the data in the appropriate way for the communication architecture it is developed for. When delivered messages are being displayed by the main view the connection data is unessential. Only which train, given by an ID from the safety header, is necessary to show. The safety header, as described in Section 2.7, is completely separated from the underlying communication architecture.

3.4.3 Safety Layer

Every message, inbound or outbound, starts with a safety header. It starts with a value called <STX> which is a first check that the message is usable after receipt. Any errors encountered are noted but the message is still delivered since it is valuable information to the user.

The time stamp containing the time of the send out should have an increased value for each sent message but it should, however, also support roll-over. TCCSim checks this by calculating the difference between the previous value received and the current value, with roll-over awareness. If the difference exceeds the maximum value of the polling interval and the timeout value times two, it is not seen as a roll-over but an error. Shown in Listing 3.6.

```
if(Current_Timestamp < Prev_Timestamp && (Current_Timestamp -
    Prev_Timestamp + 65535) > (Max(Polling_Interval, Timeout_Value) * 2))
  Timestamp_Error = true;
fi
```

Listing 3.6: Check if a time stamp error has occurred

The CRC calculations specified for this system appears not to be standard calculations. Therefor existing C code from the live system was brought into TCCSim and called from C# code with Platform Invocation Services[8] (PInvoke). PInvoke allows managed code to call unmanaged code in the .NET platform. The C code uses a table lookup algorithm[9] to increase speed when calculating the CRC. The same technique is used when calculating the BCC but this algorithm is translated from the provided C code to C# code.

3.4.4 Work Flow

When the thread executing in the communication module is started, it first verifies that connection data and timer values are in order. If there is anything wrong, the user is informed and the thread is self aborted.

If everything is in order the execution enters the main loop and stays until told to abort. The loop contains all necessary parts for the communication to never cease. At the start of every round the status of the TCP connection is checked to be connected. If there is no connection, establishing one is priority one and it will be priority one until the connection is established or the thread is aborted. Either way, the user is informed via events on what the status is.

Next step is to send a message to the vehicle. Since there always has to be a message sent with a fixed interval a default message is given the module at initialization. This default message is imported from an XML file and can therefor be changed after compile time. If the user has not provided a different message to send the default message will always be sent. The safety header is then calculated and added to the chosen message as described in Section 2.7. Since the protocol is developed with serial communication in mind, five bytes are appended to the front of the message and one to the back with value $E7_{16}$.

A response from the vehicle is then to be expected. When the communication socket is initialized a timeout value is set. If no response is received within this time something has gone wrong, the socket is closed and the main loop is started from the beginning. When a response is received, first the value <STX> is sought. When what seems to be <STX> is found, the safety header i.e. the next seven bytes is brought into memory. The total length of the message is checked and also brought into memory. If the length exceeds the maximum value of how long a message is allowed to be it was not the correct <STX> that was found. The same conclusion is taken if not enough bytes from the socket was available before timeout was experienced or if the BCC calculation of the received message fails. In this case, <STX> is sought once again. This time from the position after the previous <STX> was found. This continues until the correct <STX> is found or until no more data from the socket is available. If <STX> is never found the connection times out and a new connection has to be established.

The communication module should have as little as possible to do with the contents of the received messages. But the time stamp fields has to be checked by the module since time stamp error has to be looked for. The results of this and the message is then delivered to the main thread.

In addition to this there are also thread safe flags which are controlled by the main

thread. One flag that enables pause. When it is set there is no messages sent or received and the current state of the connection is unchanged as far as the module knows. Other flags are used to inject errors in the safety header. More about this in Section 3.5.

3.5 Fault Injection

By enabling the ability to inject faults into outbound messages more thorough tests can be performed. It allows for the system undergoing the tests to be presented with data which are not according to specifications. This section handles the different methods of injecting these faults.

3.5.1 Safety Header

A number of different scenarios are possible when injecting errors into the safety header. The number is although limited to the degree that it will be enough to declare a flag for each type of error injected. The value of the flags is given to the safety layer class each time the safety header for an outbound message is calculated. When a flag is set it will only affect the next outbound message and then it will be reset.

The interesting fields for injecting errors in the safety header is ID, T_SENDER, T_REF, CRC and BCC(see Figure 2.1). The ID field is used to double check what recipient the message was intended to (apart from the communication endpoint). When giving a faulty ID, the new value is picked at random but still parted from the correct value.

For each sent message the time stamp value of T_SENDER should be increasing (with the exception of roll-over). Two options for injecting fault to this fields is consequently possible. If message M is sent with time stamp T_0 the subsequent sent message M' should have a time stamp $T_1 > T_0$. The first possibility for injecting a fault is to let $T_1 = T_0$. It would mean the previous message M is being resent. The other possibility is $T_1 < T_0$ which means that the receiver of M' has received an old message. Both of these injections are controlled with one flag each. Of course only one of the faults can be staged on each message.

The time stamp reference field T_REF should contain the last time stamp value that was received. When receiving message M with time stamp T, the response M' should therefor contain the time stamp reference value $R = T$. The two different possibilities for injecting faults to this field is consequently $R < T$ and $R > T$. Each of the cases is controlled by a flag.

The transmission error detection fields CRC and BCC can be changed to not correspond with the actual calculated values. Each of the fields are controlled with a flag. When a flag is set a random number is added to the calculated value to simulate a transmission error. The added value is chosen to not make the calculated value to be chosen again in the event of roll-over.

3.5.2 Erroneous messages

By the design choice to put all the message descriptive data in external configuration files, there is an opportunity to easily create messages that behave outside of the protocol. Errors that can be induced are field data out of bounds, block types that should not exist in a given message type. Block types can be set up to exists too many times in a message. Extra fields can be added to a message or block, this will disrupt the parsing of the message from that point on.

To create a faulty block, field, or message type a new type has to be added to the corresponding XML-file with the same id as the correct type that is desired to spoof. It cannot have the same name as the correct type since block, field and message types are identified by their names by TCCSim.

3.6 Script

With the ability for a user of TCCSim to write scripts and execute them opens the possibilities to optimize the usage of the application. It would be more efficient and more exact to let a script handle all events when communicating with a vehicle. This section describes how this was solved while Section 4.5 describes why this path was taken.

3.6.1 Overall Structure

A Python script has to be written in a third party text editor. The complete script is then loaded into TCCSim via the application's main view where it is interpreted by IronPython (described in Section 2.4). The script can be loaded in which ever state TCCSim happens to be (connected/disconnected). Preferably the script is suited to start in that state, which is up to the user to ensure. All state changes the script performs is recorded and will be adopted by the GUI when the script returns. All of

the GUI's functionality to interact with the ongoing conversation is disabled while the script executes.

The GUI cannot be completely disabled or frozen if the script should be of any use to the user. For this to be at all possible the script has to execute in its own thread. Once again, as with the communication module, the BackgroundWorker from C#'s libraries was used. When a script is loaded, the BackgroundWorker will start to run the script in the background. It will use the report progress feature to interact with the main thread. This way, inbound and outbound messages will be displayed in the main view as usual and the user will be able to analyse the content of the messages by clicking them. The script returns when it is finished or the user can cancel the script at any time.

3.6.2 Script Functionality

If a written script should be of any use it has to reach key functions in TCCSim's arsenal. These functions are gathered in a class called Script, which is also the class that calls the script. When an instance of Script calls a given script it attaches itself to the call. When the script runs it will be able to call public functions of that instance. It is important to limit what a script should and should not be able to reach inside TCCSim since the script runs in a separate thread. A writer of a script cannot determine what functions can be called in a thread safe manner which means these public functions will have to be thread safe by nature. As mentioned in the previous section the script thread uses BackgroudWorker's progress reports to interact with the main thread.

The functions that a script has at its disposal includes connect, disconnect, pause/unpause, send messages, read messages, injecting faults into outbound messages, and the ability to load messages and manipulate them. Apart from these TCCSim bound functionalities, the implemented functionality of IronPython is also available. This does not include all Python features since IronPython at the time of writing is a work in progress. It does, however, mean any available C# library can be imported and used. This adds up to a very powerful scripting implementation.

3.6.3 Print

It is convenient to output text when running a script. Since TCCSim is not a console application another method than STDOUT would have to be used for printing. A new tab to the main view's lower part was added. The tab contains a TextBox with the looks and functionality of a output window. When printing from a script the output will be added to this TextBox.

IronPython provides the opportunity to redirect STDOUT of a running script to a StreamWriter. A new class called ScriptStream, which inherits StreamWriter, was created and given as output stream. A ScriptStream is given a BackgroundWorker, which is used by the script to communicate between threads. When writing to an instance of ScriptStream it will use the provided BackgroudWorker to send an event to the main thread and by that output text to the output TextBox. This technique makes it possible to use IronPython's default printing function to output text which appears in TCCSim's main view.

4

Design Choices

This chapters motivates the choice of method.

4.1 User Interface

The new user interface have been designed with the idea of make it more intuitive, reviewable, and fitting for the task. This have been achieved by looking at the old simulator's interface and learning what was good and not good. An open dialogue with the employer were held throughout the project to further make the interface fit their needs.

4.1.1 Message creation

The interface for creating messages was iteratively developed by a dialogue with the employer. Demands were that a more reviewable interface than the previous one (seen in Figure 4.1) should be created.

Another desire was that the interface should be browsable by just keyboard input. Considering this, and that the content of each message type is unknown, an interface where each value is entered in a top down approach were presented. The idea was liked by the employer and inferred in TCCSim and can be seen in Figure 3.4.

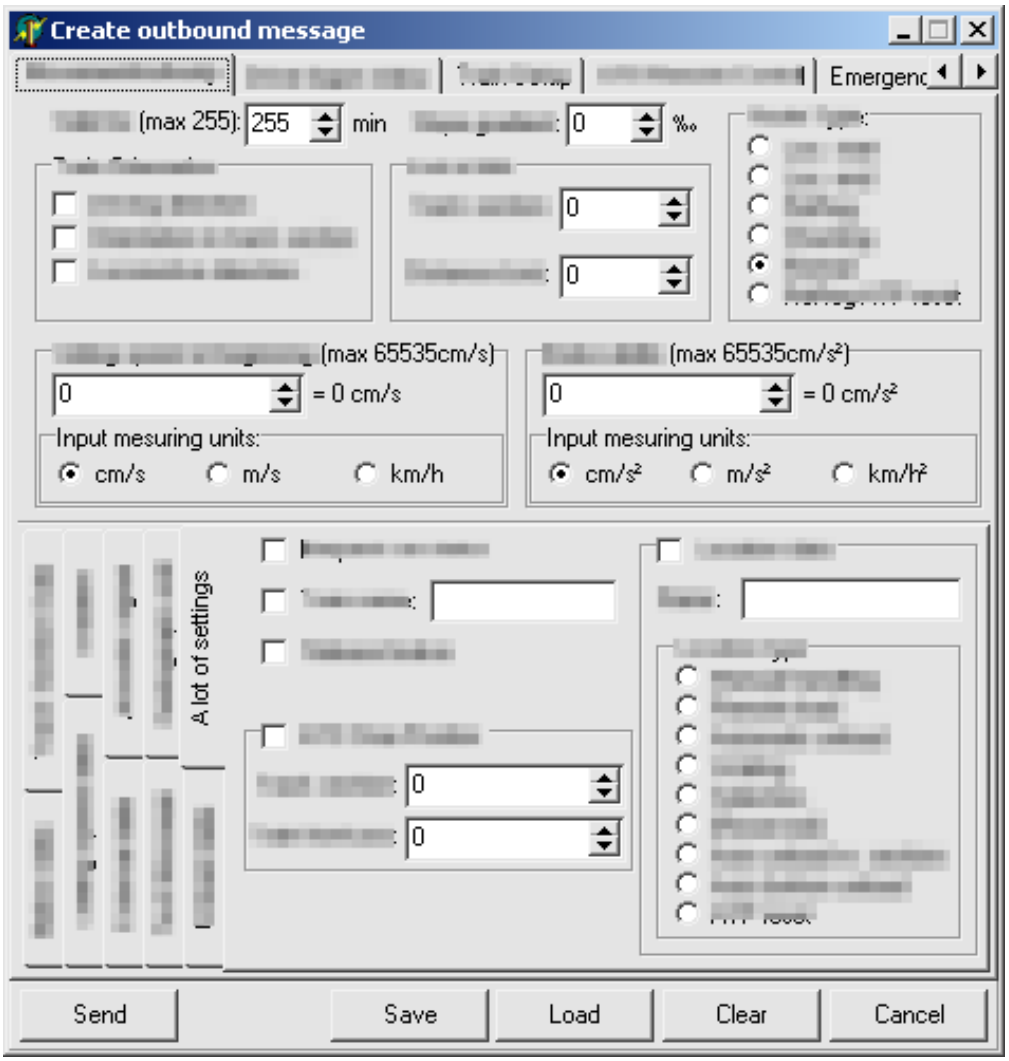

Figure 4.1: Old message editor

4.2 Dynamic Messages

4.2.1 Identification by name

The protocol identifies different message and block types by their numeric ids. In TCC-Sim they are instead identified by their names when possible, which is the case for all outgoing messages. For incoming messages only the numeric id is known. The reason for using names is to allow multiple message and block types with the same id in the protocol definition. This is used to be able to create faulty messages to be used when testing. For example, to test what happens when a message is longer than expected or

contains values out of bounds for what is expected.

4.2.2 Data types

FieldType

A FieldType is a data type for storing information about field types when they are parsed from a byte string of a message. In a FieldType there is a string for storing the name of the field stored in the FieldType and an ArrayList for the field data. The reason for having an ArrayList here is that the data in a field can vary between an integer, a string or a bitmask. Where integers and strings only need one object to represent its data in human readable form, bitmasks needs one object for each set bit.

BlockType

The BlockType data type consist of the same data objects as the FieldType. A string for the block name and an ArrayList for storing data. In the ArrayList objects of FieldType is stored for the field types that the block is built from. The reason for an additional data type where only the name differs from FieldType is to make it much easier for the programmer to differ a block from a field.

4.3 Communication Module

The main part of TCCSim and the communication module is executed in separate threads which means the threads have to communicate with each other in some way. The main part of the application uses events to pass information around. Since events do not have any awareness regarding threads, that approach could not be used without modification.

Since the communication thread is being fed, with messages to send, by the main thread, one option could be to use the same technique in reverse. The communication thread would, in this case, insert status updates and deliver messages by inserting the them into a list. The problem is that the main thread would have to take part of the inserted information in some way. It would be very inefficient and would make the user interface somewhat unresponsive to make the main thread poll the list at a given rate and therefor this idea was abandoned.

C# has the ability to marshal event between threads. For example, control objects that are used in Windows Forms remembers what thread they were created in. When trying to make changes to the object a check can be made to see if the thread trying to make the change is the same thread that created to object. If not, the execution can be marshalled to the correct thread and the change can be made. This method would, however, be very prone to errors since all changes to objects in the form would have to be preceded with this check. Also, all object which do not possess this thread checking ability could be altered by more than one thread at once, assuming no semaphores would be used.

Instead the existing C# class BackgroundWorker was used. It was created to run lengthy executions in the background with the ability to send status updates to a user interface. When sending a status update from the background thread it automatically marshals the event to the thread which created the BackgroundWorker object. This was the most convenient solution since no consideration to thread safety had to be taken when updates were sent. The main thread would also handle these events asynchronous which makes the solution efficient.

4.4 Fault Injection

Since the vehicle essentially is a finite state machine, testing methods aimed at these types of systems is applicable. Sidhu et al. [10] describes four types of methods for testing protocols defined using Mealy state machines. These methods are all possible to use for testing with TCCSim since any type of message can be sent at any given time. Because of this property, also incorrect sequences of messages can be staged, thus testing the stability of the protocol.

4.4.1 Erroneous messages

By the design choice to put all the message descriptive data in external configuration files there is easy to create messages that behave outside of the protocol. To create a message that have some data out of order, data that should not exist in given message type or similar, a message type with desired message id is added to the message descriptions. Since the faulty message type have the same id as the correct one, the program have to know the name of the new message type associated to such a byte string to be able to parse it.

Many of the vulnerabilities with software is robustness problems[11]. These types of problems most often comes to light when invalid input data is given the application. Most often robustness tests expects the application to crash or an exception being thrown.

With TCCSim's ability to specify any kind of structure for a message the robustness of the protocol and the vehicle software can be tested.

Other options that were evaluated were the possibility to randomly alter some data in the payload. This was discarded due to that it would render uncontrollable errors. Errors of this kind would have no value in the application of TCCSim. This since when TCCSim is used in test cases it is desirable to know what caused an error, not just that an error occurred. Otherwise random errors is an effective tool for testing robustness of applications according to Duran et al. [12]

4.5 Script

In the evaluation process of the possibility to introduce scripting ability in TCCSim one option evaluated was to implement a scripting language just for TCCSim. This was rather quickly abandoned since even a very simple language would have been too extensive for this project. It was not justifiable go through the process with inventing a syntax and for that implement lexing, parsing, type checking and code generation for such a small part of the project. [13]

The other option evaluated was to utilise an already implemented language with TCCSim. When looking for alternatives Python, via IronPython, and Lua, via LuaInterface, was found to be the alternatives that were the most developed and straight forward options. Both alternatives were easy to integrate in our tests. The reason for the choice of IronPython was that the personal at Bombardier favoured Python over Lua.

There is many other options for integration of interpreters exists, but none as well supported and documented than the evaluated alternatives. The option of dynamically execute C# code from TCCSim was not evaluated more than it was rejected directly since a simpler syntax were sought.

5

Testing

TCCSim has no specified behaviour and testing the application is therefor not straight forward. Most of the testing will be concerning the functionality of the application and thus making sure it will not crash. First a more in depth white box testing will be performed, followed by a more black box oriented approach.

5.1 White Box Testing

With this testing method the internal structures of the program is considered. Hench the name white box, thus the tester have full disclosure of the internal structures of the program. Sometimes also known as clear box or glass box testing. Units or modules of the code are tested with white box testing.

5.1.1 Parameterized Unit Tests

By using Pex[14], a specific kind of white box testing called Parameterized Unit Test(PUT) can be performed. PUTs are axiomatic specifications and aims at full coverage when doing test runs[15]. Pex analysis the code of the application and generates test cases relevant to what has been observed. Many of TCCSim's classes and functions do however have rather complex input data such as XML structures and data sockets. Pex does not handle this well and will therefor only be available to use at some degree.

5.2 Black Box Testing

No true black box testing can be performed at this stage since the developers of TCCSim is also the testers. The underlying structure of the code is known when testing. The application will, however, be tested by simply using it in possible scenarios that a user will and might encounter. At the time of testing no live hardware was available nor was the protocol specification definitive. Instead an old test system was used as vehicle for TCCSim to communicate with. This was possible since TCCSim is able to easily switch between different protocol specifications as described in Section 3.3. Since the old system communicates via serial port an IP-to-serial-port translator was also used in between.

5.3 Test Results

5.3.1 White Box

Most of the white box test cases that were carried out showed that input variables and return values has to be checked for null value and that array lengths are sufficient. A more unexpected problem was discovered in the communication module. If a vehicle sends trash data, which looks like a message, in front of the actual message, the TCP connection might timeout waiting for more data on the socket. This was fixed with select() which do not timeout the TCP connection even if no data was recorded. Instead the actual message is found and delivered.

5.3.2 Pex

The test cases that were generated by Pex revealed that either the input parameters must be checked for null value or that arrays must be checked to be of sufficient length.

5.3.3 Black Box

The black box tests shows that TCCSim has no problem communicating with the old test system. The test that was mainly performed is a startup sequence which requires a number of messages to be exchanged in a correct order. The messages were created and saved to file using the message creation interface. The sequence of messages were sent

both manually and by script. During the tests other TCCSim features such as injecting faults into the safety header and filtering the transfer log were used.

6

Results

Evaluating the results of this project will not be totally objective. This may not be needed since the scope of this project is not about comparing different testing environments rather develop an environment that fulfill the needs of the employer.

The main interface of TCCSim provides the basic functionality such as sending messages and analyse transfered messages. The outline of a conversation with a vehicle is easy to apprehend because of the message filtering feature. The content of the messages is clearly presented in both readable and raw data output. Full transfer logs can be saved for later viewing.

Creating messages to be sent is performed with the message creation interface. It allows fast input of data with the ability to tab through the input fields. Only the necessary fields are displayed to not clutter the interface. After creating a message it can either be sent or saved to file. The XML format which the files are stored in makes it easy change the content by hand or transfer the files between computers.

Since the TCP/IP communication should be easily exchangeable it was built as a separate module and included as a DLL. The GUI was also adapted to this with a single text field to input connection data. During the project a second version of TCCSim was developed called TCCsimRS232. External serial port communication code, from the previous simulator(mentioned in Section 2.6), was brought into TCCSim and replaced the TCP/IP code. Only the communication module DLL had to be rebuilt for this to work which proves the point that the means of communication is easily exchangeable in TCCSim.

Since the protocol definition of TCCSim is stored externally in XML files, injecting faults is very easy. Any kind of message may be defined which means the possibilities of creating erroneous messages are endless. This in combination with injecting faults into the safety header adds up to a powerful solution.

The ability to script the behaviour of TCCSim was implemented using IronPython. The functionality of the scripts are more or less the same as the application in general. The only drawback is the inability to create a new message from scratch in a simple way. This is still a powerful tool when doing extensible test cases.

Since TCCSim has the ability to define the protocol externally and the fact that a serial port communication module was developed in parallel, the application is completely usable with the old system. Due to this and the fact that TCCSim is more comprehensive and easier to work with it was inferred as a replacement of the old simulator, which gave some real world testing.

7

Future Development

7.1 Remote

The possibility to alter TCCSim for remote management have been researched. The test showed that it would be easy to replace the graphical user interfaces with a console application listening for connections and handling requests. Only modifications needed for this is to make a few changes in the controller to use the server view instead of the graphical view. The server is just registered to the eventhandlers and can choose to handle which events that is necessary.

The reason to not further develop a remote version of TCCSim is that there is no application for this in the current test environment.

7.2 More Than One Vehicle

To add support for handling more than one vehicle in TCCSim have not been evaluated. But one idea to implement a solution for that would be to develop the remote manageability of TCCSim and then use some program to handle several remotes at the same time.

8

Conclusion

Having a good and modular design pattern renders the application to be easy to extend with new functionality, and also reuse functionality. In TCCSim, an application were the program performs both data and user interaction, the powers of MVC appears clearly. The MVC design pattern made it easy to extend the program with scripting ability by simply attaching a scripting language to the application. The strength of MVC were also seen when remote management was evaluated. By removing the graphical user interface and hooking up a server object to the events for user interaction a whole new application was created.

Having continuously input from the employer is helpful to ensure that the final product performs and behaves like expected. This also ensures that unnecessary features of the product is not implemented.

By making the protocol dynamically reconfigurable the scope of use for the application is much larger by being able to alter it for different projects and environments. This was exemplified in this case by the fact that TCCSim were adapted to the old environment and replacing the simulation software previously used. Having the protocol easily reconfigurable also increases the number of possible test cases with the possibility to easily infer controlled errors in the testing.

9

Discussion

During the development there was never any possibility to test towards the system the simulator was meant to be used with. Fortunately by inferring the old protocol specification in TCCSim and some means to redirect IP communication to RS232 communication, testing could be performed towards the old system. This was very useful in the development process since it revealed problems not thought of and that would not have emerged during the tests towards a simple replication service. Due to the adaptability of the protocol specification handled by TCCSim this testing should be sufficient for when the new system is specified. Event though it would have been nice to been able to test it towards the intended system and by that confirm this since some modifications might have to be done to adjust the differences between TCP/IP and RS232 communication.

If TCCSim should be able to handle more than one vehicle at the time it would have been easier to implement this if that feature were considered more during the initial development phase. But considering how different communication methods uses different connection methods, this would have rendered the communication module to be much more extensive and complicated. For example with a radio protocol all communication is broadcasted to all vehicles at the same time, but with TCP a connection to each vehicle is established were just messages interesting for that vehicle is delivered.

References

[1] Bombardier awards (2011).
URL http://www.bombardier.com/en/transportation/about-transportation/awards?docID=0901260d800d8189

[2] Extensible markup language (xml) (2011-07-08).
URL http://www.w3.org/XML/

[3] C. Nagel, B. Evjen, J. Glynn, K. Watson, M. Skinner, Professional C# 2008, Wiley Publishing, Inc., Indianapolis, 2008.

[4] Garbage collection ((2011-03-29)).
URL http://msdn.microsoft.com/en-us/library/ms228629(VS.80).aspx

[5] Type safe (2011-03-29).
URL http://msdn.microsoft.com/en-us/library/ms173104.aspx#Y100

[6] Ironpython (2011-03-29).
URL http://ironpython.net/

[7] E. Meijer, B. Beckman, G. Bierman, Linq: reconciling object, relations and xml in the .net framework, in: Proceedings of the 2006 ACM SIGMOD international conference on Management of data, SIGMOD '06, ACM, New York, NY, USA, 2006, pp. 706–706.
URL http://doi.acm.org/10.1145/1142473.1142552

[8] Platform invoke tutorial (2011-05-18).
URL http://msdn.microsoft.com/en-us/library/aa288468(v=VS.71).aspx

[9] T. Ramabadran, S. Gaitonde, A tutorial on crc computations, Micro, IEEE vol. 8 (nr. 4) (1988) p. 62.

[10] D. Sidhu, T.-K. Leung, Formal methods for protocol testing: A detailed study, Software Engineering, IEEE Transactions on 15 (Issue:4) (1989) p. 413.

[11] R. Kaksonen, A functional method for assessing protocol implementation security (2001).

[12] J. W. Duran, S. C. Ntafos, An Evaluation of Random Testing, Software Engineering, IEEE Transactions on SE-10 (Issue:4) (1984) p. 438–444.
URL `http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=5010257`

[13] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques, & Tools with Gradiance (pkg), 2/E, Prentice Hall, 2007.

[14] Pex (2011-07-05).
URL `http://research.microsoft.com/en-us/projects/pex/`

[15] N. Tillmann, W. Schulte, Parameterized unit tests, One Microsoft Way.

A

UML

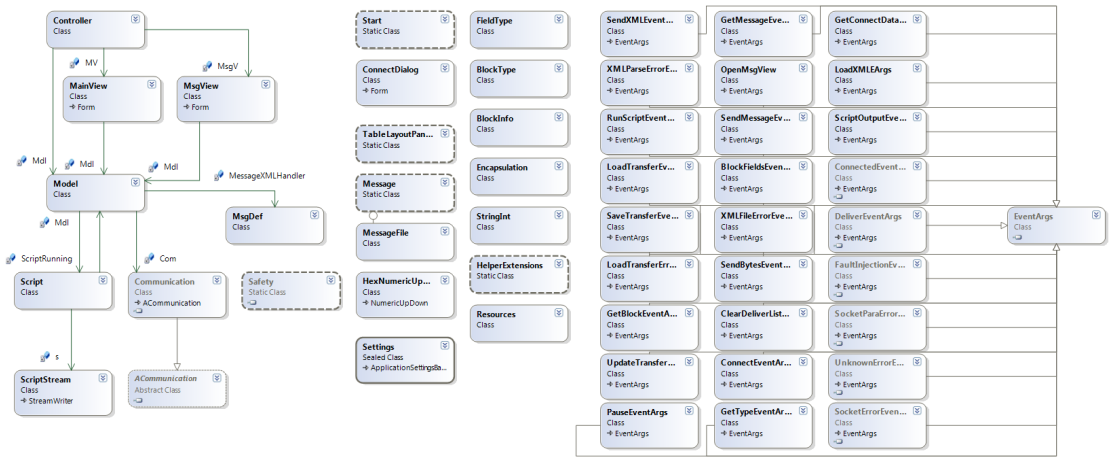

Figure A.1: TCCSim UML