MASTER

Conformance testing using formal specifications

Zwegers, F.E.W.

Award date:
2007

Link to publication
Conformance testing
using
formal specifications

F.E.W. Zwegers
Abstract

A desired property for any software is good reliability. The reliability is closely related to the number of faults in an implementation. So detecting more faults during the test phase of a software project, i.e. being more thorough, will help to increase the reliability.

In this report research is done on the use of a formal specification of a protocol for the conformance testing of an implementation of that protocol. The thesis is that this results in a more thorough and efficient test.

In this research the specification language mCRL2 and the test tool TorX are used. The mCRL2 language is developed by the Design and Analysis of Systems group from the Eindhoven University of Technology (TU/E). TorX is a test tool for testing reactive software based on the IOCO theory developed by the Formal Methods and Tools group of the University of Twente (UT).

Tests are done on two protocols that are used in a general X-ray system from Philips Medical Systems B.V.. They are first specified using mCRL2. Their specifications are then used to develop three conformance testers based on the test tool TorX.

With these conformance testers a hardware and three software implementations of the two protocols are tested for conformance with the formal specifications. These tests produce some interesting results: Two problem reports are sent to the manufacturer of the hardware implementation and further testing of the hardware implementation is ceased due to these problems.

Finally it a case is made that the researched approach has some benefits like an unambiguous, formal specification of a protocol for which certain properties can be proven, and a thorough approach for conformance testing.

However there are also some drawbacks like a high learning curve for the developers that want to use these formal methods and an awkward pass-fail condition for the tests.
Acknowledgements

Many people at the university, at Philips and at home helped me to complete this project. Some played an important or special role in this process without whom I wouldn’t have completed it in this way.

First I would like to thank both Jan Friso Groote and Mark den Hartog for their council, supervision and reviewing. You really helped me to find the right people with the right information; even when they were in Hamburg.

Second I would like to thank Raoul Bastiaens for his work in acquiring a flat panel detector with a working pixbox for my test setup. Although it has taken a few weeks to get everything right, the test results made the time worthwhile.

Third I want to thank Axel Belinfante for all his advice on TorX and answering all my questions.

And finally I want to mention the members of the Brics project team who have been my colleagues for the last 8 months. I really enjoyed my time at Philips and wish all of you all the best.
Contents

1 Introduction 1
   1.1 Background ......................................................... 1
   1.2 Problem definition ............................................... 1
   1.3 Approach .......................................................... 2
   1.4 Organizational setting .......................................... 3
   1.5 List of abbreviations ............................................ 3
   1.6 Structure of this report ........................................ 3

2 Conformance testing 5
   2.1 The IOCO test theory ............................................ 5
   2.2 TorX ............................................................... 9

3 mCRL2 13
   3.1 Brief history ...................................................... 13
   3.2 The process language ............................................ 14
   3.3 The data language ................................................ 15

4 Specification of the Mailbox protocol 17
   4.1 Description LTP ................................................... 18
   4.2 Description Mailbox .............................................. 21
   4.3 mCRL2 specification LTP ......................................... 23

5 Specification of the DVL protocol 29
   5.1 Description ......................................................... 29
   5.2 mCRL2 specification of DVLP ................................... 33

6 Conformance testers 39
CONTENTS

6.1 Development environment ............................................. 39
6.2 Test setup ....................................................................... 39
6.3 General approach ............................................................. 40
6.4 General design decisions .................................................... 41
6.5 Conformance tester for the Mailbox protocol ....................... 41
6.6 Conformance tester for the DVL protocol ......................... 43
6.7 Conformance tester for the PX5100 .................................. 45

7 Quality of a conformance tester ........................................ 47
  7.1 Theory on Error seeding ................................................ 47
  7.2 Test approach ................................................................ 48
  7.3 Seeded errors ............................................................... 48
  7.4 Test results ................................................................. 49

8 Conclusion ................................................................. 51
  8.1 Using a formal specification ............................................. 51
  8.2 Thorough testing .......................................................... 51
  8.3 Efficient testing ............................................................ 52
  8.4 Developments at PMS .................................................... 52
  8.5 Further research ........................................................... 52

Bibliography ......................................................................... 55

List of Figures ...................................................................... 57

List of Tables ....................................................................... 58
Introduction

A desired property for any software is good reliability. The reliability is closely related to the failure rate \cite{GYM02} and number of faults in an implementation. Good reliability is especially important in medical systems where an error could potentially inflict harm on a human being.

Philips Medical Systems B.V. (PMS) is a top player in the segment of medical imaging systems. Their products include cardio/vasculair X-ray systems and magnetic resonance systems. One of their products is a line of general X-ray detectors \cite{PMS}.

1.1 Background

Each X-ray detector has a basic system setup depicted in figure 1.1. At one end is the detector. This subsystem is responsible for doing the actual X-raying. It basically consists of a røntgen tube, a flat panel detector and a hardware controller. This hardware controller can control the amount of radiation that is emitted from the røntgen tube and collects an image from the flat panel detector.

This image is then transported to the Image Processing (IP) subsystem using a protocol designed by PMS called the Digital Video Link Protocol (DVLP) \cite{Fab04}. The image processing subsystem is a chain of operations that enhance the X-ray image so it is clearer and more data can be retrieved from it.

The enhanced image is then transported to the backend. This subsystem consists of several applications that physicians can use to view and store the images and add meta data to them. This system is also linked to the patients electronic file which allows the X-ray images to be consulted by other physicians too.

1.2 Problem definition

The detector subsystem is made by a third-party in cooperation with Philips\textsuperscript{1}.

The external communication of the detector, which is the responsibility of the hardware controller, is defined in two protocols created by Philips called Mailbox and DVLP (explained further in sections 4 and 5).

Each time a new version of a detector is created Philips has to validate its compliance with

\textsuperscript{1}There currently is only one third-party which creates the detector subsystem: Trixell. Trixell is a joint venture between Thales Electron Devices (51%), Siemens Medical Solutions (24,5%) and Philips Medical Systems (24,5%).
these protocols. Furthermore Philips has some libraries and tools that should also adhere to these protocols.

Currently a set of test cases is executed to test for conformance with these protocols. Unfortunately these test cases do not fully test the detector subsystem. This results in unexpected failures in the field.

The goal of this thesis is to find out if this conformance check can be done more thoroughly and efficiently using a formal specification.

1.3 Approach

At the start of the project a choice has been made that the test tool TorX [Tor06] and the specification language mCRL2 [MCR06] were going to be used for this project.

To be able to monitor the progress of the project a planning is made that has several phases of development. Each phase has one or more goals:

1. a. Understand and model the protocols using mCRL2
   b. Experiment with TorX and understand the theory behind it
2. Build a conformance tester for each protocol.
3. Conduct tests on some implementations
4. Write a final report
1.4 Organizational setting

This master thesis is done at Philips Medical Systems B.V. in Best for the department of Digital Image Handling of the Business Line Components. Because the interface specifications for both protocols are in the software archive of the BRICS project team, I was added to that team.

1.5 List of abbreviations

All non-standard abbreviations in this report that occur more than once are listed below for easy reference. Note that the abbreviation IP is defined twice. The context where this abbreviation is used is clear on which of the definitions holds.

- **ACP** Algebra of Communication Processes
- **DVLP** Digital Video Link Protocol
- **FIFO** First In First Out
- **IP** Internet Protocol
- **IP** Image Processing
- **IOCO** Input Output Conformance
- **IOTS** Input Output Transition System
- **IUT** Interface Under Test
- **LTP** Lightweight Transport Protocol
- **LTS** Labeled Transition System
- **mCRL2** Mili Common Representation Language 2
- **MSC** Message Sequence Chart
- **PMS** Philips Medical Systems
- **TCP** Transport Control Protocol
- **TU/e** Eindhoven University of Technology
- **UDP** User Datagram Protocol
- **UT** University of Twente

1.6 Structure of this report

This report is divided into four parts. Chapters 2 and 3 introduce resp. TorX and the theory behind TorX, and the mCRL2 specification language. Using mCRL2 a specification for each protocol is made. These are treated in chapters 4 and 5. The actual construction of the conformance testers and the test results are presented in chapter 6. And finally chapter 7 discusses how the quality of a conformance tester can be determined using error-seeding and presents the results of applying this technique to one of the conformance testers.
Testing an application for conformance with a specification or a reference application is easier said than done. Several questions come to mind when thinking about what conformance testing comprises:

- When does an application conform to a specification?
- How to test this conformance?
- What should a conformance tester test?
- What shouldn’t a conformance tester test?

This chapter provides answers to these questions. First the test theory on which the conformance testers are based is discussed in section 2.1. Then a test tool, called Torx [Tor06], that supports this theory and is used to construct the conformance testers, is presented in section 2.2.

### 2.1 The IOCO test theory

A formal approach to conformance testing is the IOCO test theory described in [Tre96]. IOCO is an abbreviation for Input Output Conformance. This theory can be used to test reactive software. Reactive software is software that accepts external input and produces observable output based on this input.

An example of a reactive software system is a coffee machine. When pushing the coffee button - the external input -, the machine provides a cup of dark fluid - the observable output.

Based on this notion of reactive software a first definition of conformance testing can be given:

**Definition 2.1**

An implementation is conformant with a certain specification when the implementation doesn’t exhibit any external behavior that isn’t specified in the specification.

This definition gives an indication on what conformance testing is, but still leaves a lot open for own interpretation; so let’s take a formal approach.

This formal approach is based on the assumption that both implementations and specifications can be modeled as a Labeled Transition System (LTS). Definition 2.2 defines an LTS. Definition 2.2 and 2.3 and the paragraph between them are almost fully quoted from [Tre96].
CHAPTER 2. CONFORMANCE TESTING

Definition 2.2
A Labeled Transition System is a 4-tuple $(S, L, T, s_0)$ where

- $S$ is a countable, non-empty set of states
- $L$ is a countable set of labels
- $T \subseteq S \times (L \cup \{\tau\}) \times S$ is the transition relation, where $\tau$ is the unobservable, internal action
- $s_0 \in S$ is the initial state

The labels in $L$ represent the observable actions of a system; the special label $\tau \notin L$ represents an unobservable, internal action. A transition $(s, \mu, s') \in T$ is denoted as $s \xrightarrow{\mu} s'$. A computation is a (finite) composition of transitions:

$$s_0 \xrightarrow{\mu_1} s_1 \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_{n-1}} s_{n-1} \xrightarrow{\mu_n} s_n$$

A trace captures the observable aspects of a computation; it is the sequence of observable actions of a computation. The set of all finite sequences of actions over $L$ is denoted by $L^*$, with $\epsilon$ denoting the empty sequence. If $\sigma_1, \sigma_2 \in L^*$, then $\sigma_1 \cdot \sigma_2$ is the concatenation of $\sigma_1$ and $\sigma_2$.

We denote the class of all labeled transition systems over $L$ by $\mathcal{LTS}(L)$. For technical reasons we restrict $\mathcal{LTS}(L)$ to labeled transition systems that are strongly convergent, i.e. ones that do not have infinite sequences of transitions with only internal actions. Some additional notation is introduced in definition 2.3

Definition 2.3
Let $M = (S, L, T, s_0)$ be a LTS with $s, s' \in S$, and let $\mu(i) \in L \cup \{\tau\}$, $a(i) \in L$, and $\sigma \in L^*$.

$$s \xrightarrow{\mu_1 \ldots \mu_n} s' = \text{def} \ \exists s_0, \ldots, s_n : s = s_0 \xrightarrow{\mu_1} s_1 \xrightarrow{\mu_2} \ldots \xrightarrow{\mu_n} s_n = s'$$

$$s \xrightarrow{a_1 \ldots a_n} s' = \text{def} \ \exists \sigma : s = s_0 \xrightarrow{a_1} s_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} s_n = s'$$

Furthermore we split $L$ in $L_I$ en $L_O$ for resp. labels for input and output transitions. An LTS that distinguishes input and output transitions is called an Input-Output Transition System (IOTS). $\mathcal{LOTS}(L_I, L_O)$ denotes the set of all (strongly convergent) input-output transition systems where $L_I$ is the set of labels for input transitions and $L_O$ is the set of labels for output transitions.
2.1. THE IOCO TEST THEORY

With definition 2.2 and the extra terminology we can now define traces(s) as the set of all traces that start in a state s, s after σ as the set of states reachable after a trace σ from state s, and out(S) as the set of output transitions for all states s ∈ S. To put it more formally:

**Definition 2.4**
Given an LTS \( M = \langle S, L_I \cup L_O, T, s_0 \rangle \). We define for a state s ∈ S and set of states \( S' \subseteq S \):

1. \( \text{traces}(s) = \text{def} \{ \sigma \in L^* \mid s \xrightarrow{\sigma} \} \)
2. \( s \text{ after } \sigma = \text{def} \{ s' \in S \mid s \xrightarrow{\sigma} s' \} \)
3. \( \text{out}(s) = \text{def} \{ \mu \in L_O \mid s \xrightarrow{\mu} \} \)
4. \( \text{out}(S') = \text{def} \bigcup \{ \text{out}(s') \mid s' \in S' \} \)

For better readability the definitions of \( \text{traces}(s) \) and \( s \text{ after } \sigma \) are extended to \( M \):

1. \( \text{traces}(M) = \text{def} \text{traces}(s_0) \)
2. \( M \text{ after } \sigma = \text{def} s_0 \text{ after } \sigma \)

Based on these definitions definition 2.1 can be rewritten as a relation called i/o-conformance: \( \text{ioconf} \)\(^1\).

**Definition 2.5**
Let \( M_I \in IOTS(L_I, L_O), M_S \in LTS(L_I \cup L_O) \), then

\[ M_I \text{ ioconf } M_S = \text{def} \forall \sigma \in \text{traces}(M_S) : \text{out}(M_I \text{ after } \sigma) \subseteq \text{out}(M_S \text{ after } \sigma) \]

The definition states that all possible output transitions of the implementation \( M_I \) after a trace in the specification \( M_S \) should at least be possible in the specification. The advantage is that every trace that exists in \( M_S \), must be possible in \( M_I \) if the state in which the trace ends has one or more output transitions.

There are also some drawbacks to the \( \text{ioconf} \) relation. One of them is depicted in figure 2.1. It illustrates a general problem of black-box testing. Because only output actions can be observed from an implementation, it is impossible to test for input actions that aren’t allowed without trying to do a transition with every label in \( L_I \) for every \( s \in S \) in \( M_S \).

Another drawback of the \( \text{ioconf} \) relation is shown using the following example: Figure 2.2 shows a specification and an implementation of a coffee machine that have the \( \text{ioconf} \) relation. The specification \( (C_S) \) states that if you push the button \( (\text{but}_I) \) once a coffee \( (\text{cof}_O) \) may be given. If no coffee is given pushing the button a second time will result in a cappuccino \( (\text{cap}_O) \). Pushing the button after either a coffee or cappuccino was provided will not result in a new output.

It is left to the reader to determine that the implementation \( (C_I) \) equals \( C_S \) given the \( \text{ioconf} \) relation.

\(^1\)This is definition 4.7 in [Tre96]
The drawback is that the ioconf relation only tests for the presence of output actions during a trace and not for their absence. For instance it should be testable that after having pushed the button once nothing happens and after pushing it a second time only cappuccino will be outputted.

The absence of output actions is called quiescence and is denoted by a special action: $\delta$. To include this action the definition of $\text{out}(s)$ needs an addition.

**Definition 2.6**

Let $M = \langle S, L_I \cup L_O, T, s_0 \rangle$ be an LTS. We define for each state $s \in S$:

1. $\delta(s) =_{def} \neg \exists \mu \in L_O s \xrightarrow{\mu}$
The set of traces that include $\delta$ for quiescence are called suspension traces and are written as $\text{Straces}(M)$ for $M \in \mathcal{L}_{TS}(L)^2$. The $\text{ioconf}$ relation is the $\text{ioconf}$ relation that is restricted to suspension traces:

**Definition 2.7**

Let $M_I \in \mathcal{IOTS}(L_I, L_O)$, $M_S \in \mathcal{L}_{TS}(L_I \cup L_O)$, then

$$M_I \text{ ioco } M_S = \text{def } \forall \sigma \in \text{Straces}(M_S) : \text{out}(M_I \text{ after } \sigma) \subseteq \text{out}(M_S \text{ after } \sigma)$$

The $\text{ioconf}$ relation checks for states with quiescence, but still cannot solve the problem depicted in figure 2.1.

The conformance testers, which are discussed in chapter 6, test for this relation given a specification and an implementation of a protocol.

### 2.2 TorX

The linux-based test tool TorX [Tor06] can test if an implementation is IOCO compliant with a (formal) specification, i.e. they have the $\text{ioconf}$ relation. It was developed by the Formal Methods and Tools research group of the University of Twente (UT). It has been chosen for this project, because it’s the only known conformance testing tool that accepts a mCRL2 specification as input.

The component based structure of TorX is depicted in figure 2.3. Each component has a specific task and a well-defined interface. This makes TorX better adaptable to support another formal language. In the next subsections a small introduction to each component of TorX is provided. On the TorX website [Tor06] the components and their interfaces are explained in detail.

\[2\] A formal definition is omitted here, because it is not needed to understand a suspension trace.
2.2.1 Specification and IUT

The specification component is the formal specification that will be used to check the conformance of the Interface Under Test i.e. the implementation.

The specification may be in any formal language for which an explorer component can be constructed. Some of the supported languages include LOTOS \cite{vEVD89}, FSP \cite{MK99} and mCRL2.

Currently the IUT can be any interface which allows communication by UDP/IP \cite{RFC80, RFC81a} or TCP/IP \cite{RFC81b}. However the adaptor component can easily be modified to support other types of connections too.

2.2.2 Explorer and primer

The explorer provides a (textual) interface to the LTS of the specification. Its purpose is to provide information on the transitions that can be done from the current state. It is unaware of input and output transitions and doesn’t maintain any history.

The primer divides the transitions from the explorer in input and output actions. The user of this component must provide a config file which describes which transitions are an input action and which are an output action. This config file can also list certain actions which trigger a failure or pass response. These responses can be used by the driver to determine if a test should end.

The primer also provides an interface to the driver component.

2.2.3 Adaptor

The adaptor translates actions received from the driver to actual calls that the IUT can understand, e.g. UDP packets, and vice-versa.

Furthermore some basic type checking of the received output actions can be done in the adaptor. At first sight this might sound questionable, because from a modeling point of view the specification should check for these inconsistencies. However there are several reasons to implement the type checking in the adaptor:

1. It makes the specification, i.e. the model, less complex and prevents a state space explosion due to enumeration of a type, e.g. an integer.
2. It prevents arguments that have no meaning in the specification from being modeled, but checks their validity nonetheless. For example a version number should always be 1.
3. The \texttt{ioco} relation will fail on any output action that is not in the specification. If such an action is already detected in the adaptor, this will not cause a different output for the conformance test.
2.2.4 Driver

This component implements the algorithm which checks the IOCO relation. It uses the information provided by the primer and the adaptor components to decide what action should be done next. When an input action is chosen, the adaptor is instructed to process this action. If an output action should be expected, the driver waits until either a message from the adaptor arrives or a timeout, i.e. quiescence, occurs.

After an action is done, the primer is instructed to execute that action in the specification. If the action is unavailable in the primer, the driver reports that the conformance test has failed.

Unlike the explorer, primer and adaptor, the driver configuration is not affected by the specification or the IUT.
The formal basis of the conformance testers is a specification of the protocols in the specification language mCRL2. This language can be used to specify and analyze the behavior of distributed systems. In this project mCRL2 has been used to specify the Mailbox and DVLP protocols. Chapter 4 and 5 describe these protocols and the resulting specification.

The standard approach for analyzing a system using mCRL2 is:

1. A specification of the system’s behavior is written in the mCRL2 language.
2. This specification is converted to a Linear Process Specification (LPS). This is an mCRL2 specification in a stricter format.
3. The LPS can be modified/simplified using various tools from the mCRL2 toolset.
4. A LTS can be generated from the modified LPS. This LTS can then be used to check for errors using model checking techniques or can be used in the TorX tool set.

This chapter gives a brief history of mCRL2 and describes the syntax and semantics of the language to the extend that someone unknown to the language can understand the specifications of the protocols. The content of this chapter is largely based on [GMR+07]. The full description of the mCRL2 language and the concepts behind it, can be found in [GvdP04, Fok00].

### 3.1 Brief history

The mCRL2 language is the successor of the $\mu$CRL specification language [Gro97]. The $\mu$CRL language was developed to specify communicating systems in the Algebra of Communicating Processes (ACP) using equational abstract data types. This use of data types within a process algebra specification turned out to be very valuable in the specification and analysis of real-life systems.

The mCRL2 language has a number of improvements over the $\mu$CRL language that increase the expressive power of the language and add to the user-friendliness of it. These improvements include standard data types, representing time, multi-actions and local communication.

* * * *

The mCRL2 language is divided in two separate languages: the process language and the data language. The first comprises the ACP algebra with some extensions. The second is used to describe the abstract data types which can be used to model data in the process language.
3.2 The process language

The most basic notion in the mCRL2 language is an action. The mCRL2 fragment below - taken from section 2.1 of [GMR+07] - shows the definition of three actions. The action error is parameterless. The action send has a data parameter of type $B$ (booleans). And the action receive has two data parameters one of type $B$ and one of type $N$ (natural numbers).

```
act error;
send : B;
receive : B × N;
```

Each process is expressed by a process expression. Process expressions, denoted by $p, q, \ldots$, are either an action or an operator combined with process expressions. The following lists contains most the special actions and operators. Most of this text is quoted from [GMR+07].

- **Deadlock** or inaction $\delta$, is used to indicate that the specification cannot do an action anymore.
- **Internal action** $\tau$ is a special action that denotes some (unknown) internal behavior.
- **Alternative composition**, written as $p + q$. This expression non-deterministically chooses to execute either $p$ or $q$.
- **Sequential composition**, written as $p \cdot q$. This expression first executes $p$ and upon termination of $p$ executes $q$.
- **Conditional operator**, written as $c \rightarrow p \diamond q$, where $c$ is a data expression of type $B$. This process expression behaves as an if-then-else construct: if $c$ is true, then $p$ is executed else $q$ is executed.
- **Process references**, written as $P(d), Q(d)$, etc., where $d$ is a parameter list, are used to refer to processes declared by process definitions of the form $P(x:D) = p$. This process definition declares that the behavior of the process reference $P(d)$ is given by $p[d/x]$, i.e. $p$ in which all free occurrences of variables $x$ are replaces by $d$.
- **Summation operator**, written as $\sum_{x \in D} p$, where $x$ is a variable of type $D$ and $p$ is a process expression in which this variable may occur. The corresponding behavior is a non-deterministic choice among this process $p[d/x]$ for all elements $d \in D$. For $D = \{d_0, d_1, \ldots, d_n, \ldots\}$ this can be expressed as $p[d_0/x] + p[d_1/x] + \ldots + p[d_n/x] + \ldots$.
- **Parallel composition** or merge $p||q$, which interleave and synchronizes the actions of $p$ with those of $q$.
- **Restriction operator** $\nabla_V(p)$ (also known as allow), where $V$ is a set consisting of a set of actions names specifying exactly which action from $p$ are allowed to occur.
- **Communication operator** $\Gamma_C(p)$ (also known as comm), where $C$ is a set of allowed communications of the form $a_0| \cdots |a_n \rightarrow c$, with $n \geq 1$ and $a_i$ and $c$ action names. For example communication may occur when parallel processes $p||q = a.p'||b.q'$ and $a|b \rightarrow c$. 
• **Multiaction operator**, written as $a \sqcup b$, where $a$ and $b$ are actions. A multiaction is used to represent that $a$ and $b$ are truly done in parallel, i.e. they can be seen as an atomic action. By definition $\tau$ represents the empty multiaction.

• **Time operator**, written as $p@t$, where $t$ is a data expression of type $\mathbb{R}^{\geq 0}$. The expression $p@t$ indicates that the first action of $p$ happens at time $t$. Because $t$ is only used for modeling, it has no dimension.

Using these operators the processes $C_S$ and $C_I$ in example ?? can written as:

```plaintext
act  but, cof, cap;
proc C_I   =  but\cdot C_{cof} + but\cdot CR_I;
       CR_I =  but\cdot (CR_I + cof\cdot C_{but} + cap\cdot C_{but});
       C_S   =  but\cdot C_{cof} + but\cdot CR_S;
       CR_S =  but\cdot (CR_S + cap\cdot C_{but});
       C_{cof} =  but\cdot C_{cof} + cof\cdot C_{but};
       C_{but} =  but\cdot C_{but};
```

There is always a process expression that is executed first. This is the initial process.

```plaintext
init  C_S;
```

### 3.3 The data language

The mCRL2 data language is a functional language to create abstract data types and mappings, i.e. functions, on these data types.

A data type in mCRL2 is called a **sort**. There are some standard sorts like $\mathbb{B}$, $\mathbb{N}$, $\mathbb{N}^+$, $\mathbb{Z}$ and $\mathbb{R}$ that are already defined. Some basic mappings on these standard sorts like $\text{div}$, $\text{mod}$, $+$, $-$, $\cdot$, $<$, $\leq$, etc. are also predefined. A complete list can be found in [Mat06].

A user can also define his own sort. This either a reference to another sort, e.g. a sort $\text{Space}$ which equals $\mathbb{N}$, or a **structured type**. Example 3.1 shows the definition of a sort $\text{Tree}$.

**Example 3.1**

```plaintext
sort   Tree =  struct  Leaf(\mathbb{N})
       |        Node(Tree, Tree);
map    TreeSum : Tree \rightarrow \mathbb{N};
var    t_1, t_2 : Tree;
       n : \mathbb{N};
eqn    TreeSum(\text{Leaf}(n)) = n;
       TreeSum(\text{Node}(t_1, t_2)) = TreeSum(t_1) + TreeSum(t_2);
```
The sort $\text{Tree}$ is a structured type describing a binary tree. Each $\text{Leaf}$ of the tree holds a natural number; each $\text{Node}$ has two sub-trees.

The mapping $\text{TreeSum}$ calculates the sum of all leaves in a tree. It is defined as a function from $\text{Tree}$ to $\mathbb{N}$. The equations on $\text{TreeSum}$ show that the sum is calculated using recursion on the structure of the tree. The free variables that are needed for these equations are defined using the keyword $\text{var}$.

The mCRL2 language also has some predefined sorts like lists, sets, bags and lambda functions. These can be very powerful in describing the behavior of a system. More on these data types can be found in [Mat06].
Mailbox is a connection-oriented protocol based on UDP/IP [RFC80, RFC81a] designed to exchange detector command messages between the detector and the backend (see chapter 1). Although in practice the only known use is this communication, it is specified in the interface specification [Can06] as a generic handshake protocol which omits the actual detector command set.

Mailbox consists of two protocols: the Lightweight Transfer Protocol (LTP) and the mailbox protocol. This name is ambiguous with the name of the entire protocol. In this rest of this chapter 'mailbox' will refer to the protocol that fills the data field of the LTP protocol, unless explicitly noted otherwise.

It is not documented why this division was made, since the mailbox protocol is only a data structure that packages a detector command. A reason could be that if LTP is used to transmit a different kind of data, there is a distinction between LTP packets containing a detector command within a mailbox shell and LTP packets containing some other type of message.

The protocol stack is depicted in figure 4.1. The mailbox protocol is build on top of the UDP protocol. This was chosen over TCP mainly because the performance of UDP is better, due to less overhead, and the control of time-outs on a per connection basis in TCP is not possible.

The LTP and mailbox protocols are situated in the transport layer. The content of the mailbox protocol, the detector command set, is defined in the application layer and will therefore be left out of the scope of this chapter.

Before a formal specification of LTP is presented (section 4.3), first a description is given in
### Chapter 4. Specification of the Mailbox Protocol

<table>
<thead>
<tr>
<th>Field</th>
<th>Range</th>
<th>Length in bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Magic number</td>
<td>0x6C747020 (fixed)</td>
<td>4</td>
<td>Identifier for ltp packet</td>
</tr>
<tr>
<td>Version</td>
<td>0...0xFFFF</td>
<td>2</td>
<td>Version of the protocol</td>
</tr>
<tr>
<td>Flag</td>
<td>0...0xFFFF</td>
<td>2</td>
<td>Type of packet</td>
</tr>
<tr>
<td>Sequence no.</td>
<td>0...0xFFFF</td>
<td>2</td>
<td>The sequence number for DataCmd and DataAck messages.</td>
</tr>
<tr>
<td>Datalen</td>
<td>0...0x0218</td>
<td>2</td>
<td>Length of the Data field in bytes</td>
</tr>
<tr>
<td>Data</td>
<td>≤536</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: LTP message format

Section 4.1. The mailbox protocol itself will only be described, because no non-trivial formal specification can be constructed for it. This will be shown in section 4.2. The descriptions are largely based on the interface specification in [Can06].

### 4.1 Description LTP

LTP has been designed with the following requirements:

1. Reliable (no data lost)
2. Good error indications
3. Transparent
4. In sequence (FIFO)
5. Connection-oriented. The connections are uni-directional. A bi-directional connection can be made by creating two separate uni-directional connections.

To keep things simple, functionality like windowing, piggy backing or sending multiple LTP messages in one UDP packet are not supported. This means that LTP is a straightforward, ethernet-based, handshake protocol.

#### 4.1.1 LTP message format

Table 4.1 shows the message format for LTP. The first five fields are the header. The sixth field, the data field, can be used to transport for example a mailbox message. Some header fields need further explanation.

**Flag**

The flag indicates the type of message. Table 4.2 shows the different types of messages and their abbreviations.
### 4.1. Description LTP

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONN_CMD</td>
<td>Request a connection</td>
</tr>
<tr>
<td>CONN_ACK</td>
<td>Acknowledge a connection</td>
</tr>
<tr>
<td>DATA_CMD</td>
<td>Send data</td>
</tr>
<tr>
<td>DATA_ACK</td>
<td>Acknowledge data reception</td>
</tr>
<tr>
<td>NODE_RST</td>
<td>Reset connection (node)</td>
</tr>
<tr>
<td>DISCONN_CMD</td>
<td>Request disconnection</td>
</tr>
<tr>
<td>DISCONN_ACK</td>
<td>Acknowledge disconnection</td>
</tr>
</tbody>
</table>

Table 4.2: LTP flags

**Sequence no.**

The sequence number is zero for all message types except DATA_CMD and DATA_ACK. For these types the sequence number is used to make sure that each message is relayed to the application only once and that they are relayed in the correct order.

#### 4.1.2 Basic message flow

The message sequence chart (MSC) in figure 4.2 depicts the basic message flow in a LTP connection.

The sender initiates the connection by transmitting a CONN_CMD message. The receiver confirms the connection with a CONN_ACK message. Then the sender sends $n$, $n \in \mathbb{N}$, DATA_CMDs which are each acknowledged DATA_ACK with corresponding sequence number. Subsequently the sender closes the connection by transmitting a DISCONN_CMD and the receiver acknowledges this by responding with a DISCONN_ACK.

This message sequence chart would always be applicable when the channel between sender and receiver is reliable: Meaning that each packet that is sent, is received by the receiver. It should be taken into account that a channel may not be reliable and thus loses messages.

#### 4.1.3 Channel loses a message

Figure 4.3 depicts a situation where a message is lost. In this case the DATA_CMD message is lost, but it could be any other message type. The solution to losing a message is also in the figure: a timeout occurs and the message is resent.

This might look like a good solution at first sight, but when the channel is also unavailable the second time the message is sent the same thing happens. This could result in an endless loop and thus a deadlock.

Therefore LTP has maximum retry count of three. This means that after sending a message four times, the connection is reset. This results in the transmission of a NODE_RST message - which doesn’t require any acknowledgement and thus it is irrelevant if the message is not received.
Figure 4.3 only shows what happens when a packet from the sender is lost. If a packet from the receiver is lost, no timeout can occur, since there is no feedback on the reception of the acknowledgement. This is however solved by the way timeouts are defined in the sender. If an acknowledgement is lost, the sender will timeout and retransmit the original message. The receiver just has to reacknowledge that message.

It could also be the case that there is nothing wrong with the channel, but one of the nodes, the sender or the receiver, is reset.

4.1.4 A node is reset

Figure 4.4 and figure 4.5 depict two cases in which a node - the sender or receiver - is reset.
4.2. DESCRIPTION MAILBOX

The sender is reset

When the sender is reset (see figure 4.4), it just starts with sending a CONN_CMD. This implies that the receiver must always accept a CONN_CMD and reset its connection too. What happens when the sender is reset and receives an acknowledgement? This is not covered in [Can06]. We assume that the received message can be ignored.

The receiver is reset

When the receiver is reset (see figure 4.5), it waits for a CONN_CMD from the sender. Every other packet triggers a NODE_RST message. After this the sender will reset itself and start with a CONN_CMD. This implies that the receiver must reply with a NODE_RST message to every unexpected packet.

* * * * *

There are also some other circumstances in which a message won’t arrive at its destination, e.g. when the receive-buffer in a node is full. These cases can however be simplified to one of the above cases.

4.2 Description Mailbox

Because the mailbox protocol is only a data structure for transporting commands for a detector, it suffices to just define the message format. Table 4.3 lists the fields in a mailbox.
message. The first three fields are the header.

Since there are no other protocol requirements besides the message format, unlike in LTP where there are acknowledgments etc., the mailbox protocol doesn’t require a formal model to check its validity. Inspection of the header will reveal any errors in this protocol.

**Example 4.1**
The figure below shows an example of the mailbox protocol in which a request for the temperature is sent to the detector and the temperature is returned as a response.
4.3 MCRL2 specification LTP

The formal specification of LTP is divided into two separate specifications using a common data structure. One specification is for sending (TX) and one is for receiving (RX) LTP.

In the descriptions for the sender and receiver the notion \textit{state} is used a few times. Here this notion is not a state as in a LTS, but it comprises a set of LTS states that share a same property, e.g. the sender being idle.

Because this protocol is on top of LTP, establishing the connection, sending and acknowledging the data can be left out of the MSC.

### Table 4.3: Mailbox message format

<table>
<thead>
<tr>
<th>Field</th>
<th>Range</th>
<th>Length in bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>0×4D424F5820 (fixed)</td>
<td>4</td>
<td>Identifier for mbox packet</td>
</tr>
<tr>
<td>Reserved</td>
<td>0</td>
<td>2</td>
<td>For future use, must be 0</td>
</tr>
<tr>
<td>Length</td>
<td>0...0×0200</td>
<td>2</td>
<td>Length of the customer data field, in bytes</td>
</tr>
<tr>
<td>Customer data</td>
<td>\leq 512</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4.3.1 General datatypes

The sort \texttt{LTPPacket} is a struct that represents one of the six LTP message types or the message \texttt{nothing}.

\begin{verbatim}
sort LTPPacket = struct ConnCmd ConnAck DataCmd(SeqNr,LTPData) DataAck(SeqNr) NodeRst DisconnCmd DisconnAck nothing;
sort SeqNr = struct prev same next other;
\end{verbatim}

The \texttt{DATA\_CMD} and \texttt{DATA\_ACK} messages have a sequence number, \texttt{SeqNr}, as argument. Although in the protocol description this argument is an integer, in the specification it has been simplified to a struct that has four possible values: prev, same, next and other. This was done, because introducing a 2-byte integer in the specification causes a state space explosion: Using an integer introduces 65535 possibilities for each \texttt{DATA\_CMD} and each \texttt{DATA\_ACK}. Using only 4 options decreases the number of states that are influenced by this sequence number with approximately 16000\%.

This abstraction is correct, because the rules in the protocol that use sequence numbers only depend on the previous (prev), same or next sequence number\footnote{The option \texttt{other} was added for completeness, but isn’t used in the specification and should trigger an error!}. The real sequence numbers are stored in the TorX adaptor.

The \texttt{DATA\_CMD} message also has an argument \texttt{LTPData}. The type and content of this packet depends on the test setup and is irrelevant for describing the formal LTP specification.

The \texttt{nothing} message is used to simulate that a timeout occurred. This could also have been specified as a separate mCRL2 action, but due to possible implementation in TorX this option was chosen. This will be discussed further in chapter 6.

4.3.2 LTP Sender

Actions

LTP TX only needs two actions: one for sending and one for receiving a \texttt{LTPPacket}. The actions \texttt{ltpTXSend} and \texttt{ltpTXReceive} are used for this, resp.

\begin{verbatim}
act ltpTXSend, ltpTXReceive : LTPPacket;
act ltpStop;
\end{verbatim}
After some tests were done with a preliminary specification another action, ltpStop, was added to detect a special kind of faulty protocol implementation: the non-responsive implementation. This specific type of implementation doesn’t respond to any message it receives. According to the protocol description such an implementation would be correct; a timeout occurs every time a message is sent. To be able to detect such a implementation the formal specification does a ltpStop action when three time-outs have been received.

Establishing a connection

```mcrl2
proc LTP_TX_IDLE =
    ltpTXSend(ConnCmd).LTP_TX_WAIT_FOR_CONN_ACK(0)
    + ltpTXReceive(NodeRst).LTP_TX_IDLE
    + \Sigma_{s:SeqNr} ltpTXReceive(DataAck(s)).LTP_TX_IDLE
    + ltpTXReceive(DisconnAck).LTP_TX_IDLE;

LTP_TX_WAIT_FOR_CONN_ACK(TxRetryCnt:Nat) =
    ( (TxRetryCnt < 3) \rightarrow
        ( ltpTXReceive(ConnAck).LTP_TX_CONNECTED
        + ( ltpTXReceive(nothing).
            ltpTXSend(ConnCmd).
            LTP_TX_WAIT_FOR_CONN_ACK(TxRetryCnt + 1)
        )
    )
    \circ ltpStop.delta)
    + ltpTXReceive(NodeRst).LTP_TX_IDLE;
```

The sender starts in an `idle` state. In this state only a `CONN_CMD` message can be send. When another message is received, it is ignored since the sender could have been reset.

After sending a `CONN_CMD` a reply is expected. There are three options:

1. A `CONN_ACK` message is received. This establishes the connection.
2. A timeout occurs. The `CONN_CMD` message is resent. After resending three times without receiving a `CONN_ACK`, a ltpStop action is done.
3. A `NODE_RST` message is received. This resets the connection and lets the sender return to the `idle` state.

Sending Data

```mcrl2
proc LTP_TX_CONNECTED =
    ltpTXSend(DisconnCmd).LTP_TX_WAIT_FOR_DISCONN_ACK(0)
    + LTP_TX_SEND_DATA
    + LTP_TX_IDLE;
```

When the connection is set up, the sender is a `connected` state. In this state there are again three options:
CHAPTER 4. SPECIFICATION OF THE MAILBOX PROTOCOL

1. Send a DATA_CMD message
2. Disconnect with a DISCONN_CMD message
3. Do a node reset by either receiving a NODE_RST message or sending a new CONN_CMD message.

```
proc LTP_TX_SEND_DATA =
   ltpTXSend(DataCmd(next,LTPData).
   LTP_TX_WAIT_FOR_DATA_ACK(0);

LTP_TX_WAIT_FOR_DATA_ACK(TxRetryCnt:Nat) =
   (TxRetryCnt < 3) →
   ( (ltpTXReceive(DataAck(same)).LTP_TX_CONNECTED)
   + ( ltpTXReceive(nothing).
      ltpTXSend(DataCmd(same,LTPData).
      LTP_TX_WAIT_FOR_DATA_ACK(TxRetryCnt + 1)
   )
   )
   ◦ ltpStop.delta
   + ltpTXReceive(NodeRst).LTP_TX_IDLE;
```

When the sender sends a DATA_CMD message the first time, a new sequence number is used and thus the SeqNr argument must be next. The second argument contains a piece of data. Note that using LTPData as a value here is a violation the mCRL2 syntax. It should be replaced with a meaningful value.

After a DATA_CMD message is sent, a reply is expected. Several replies are possible:

1. A DATA_ACK message is received with the same SeqNr as used for sending. This concludes the sending of the DATA_CMD message.
2. A timeout is received. The DATA_CMD message is resent with the same SeqNr. After resending three times without receiving a DATA_ACK, a ltpStop action is done.
3. A NODE_RST is received and the connection is reset.

Disconnecting

```
proc LTP_TX_WAIT_FOR_DISCONN_ACK(TxRetryCnt:Nat) =
   ( (TxRetryCnt < 3) →
   ( (ltpTXReceive(DisconnAck).LTP_TX_IDLE)
   + ( ltpTXReceive(nothing).
      ltpTXSend(DisconnCmd).
      LTP_TX_WAIT_FOR_DISCONN_ACK(TxRetryCnt + 1)
   )
   )
   ◦ ltpStop.delta
   + ltpTXReceive(NodeRst).LTP_TX_IDLE;
```
4.3. MCRL2 SPECIFICATION LTP

When a DISCONN_CMD has been sent, the sender waits for an acknowledgement. Preferably this is a DISCONN_ACK message, but it can also be a NODE_RST message. Again three timeouts may occur which are followed by a retransmission of DISCONN_CMD before action ltpStop is done.

4.3.3 LTP Receiver

Actions

The LTP RX actions are very similar to the LTP TX actions. Again there is one action for sending, ltpRXSend, and one for receiving, ltpRXReceive. The ltpStop action is not used, since the receiver doesn’t use any timeouts.

\[
\text{act } \text{ltpRXSend, ltpRXReceive : LTPPacket;}
\]

Establishing a connection

\[
\text{proc } \text{LTP\_RX\_IDLE} = \\
\text{ltpRXReceive(ConnCmd).LTP\_RX\_CONN\_ACK} \\
+ \sum_{s:SeqNr} \text{ltpRXReceive(DataCmd(s,LTPData)).LTP\_RX\_NODE\_RST} \\
+ \text{ltpRXReceive(DisconnCmd).LTP\_RX\_NODE\_RST;}
\]

\[
\text{LTP\_RX\_CONN\_ACK} = \\
\text{ltpRXSend(ConnAck).LTP\_RX\_CONNECTED} \\
+ \text{ltpRXReceive(ConnCmd).LTP\_RX\_IDLE;}
\]

\[
\text{LTP\_RX\_NODE\_RST} = \\
\text{ltpRXSend(NodeRst).LTP\_RX\_IDLE} \\
+ \sum_{g:LTPPacket} \text{ltpRXReceive(g).LTP\_RX\_NODE\_RST;}
\]

When the receiver is in its \textit{idle} state, it can receive the following messages:

1. A CONN_CMD message. This is replied with a CONN_ACK message.
2. A DATA_CMD or DISCONN_CMD message. These can be received when the receiver is reset and the sender hasn’t been notified. A NODE_RST message is sent in reply.

Receiving data

\[
\text{proc } \text{LTP\_RX\_CONNECTED} = \\
\text{ltpRXReceive(DisconnCmd).LTP\_RX\_DISCONN\_ACK} \\
+ \sum_{s:SeqNr} \text{ltpRXReceive(DataCmd(s,LTPData)).LTP\_RX\_RECEIVE\_DATA(s)} \\
+ \text{ltpRXReceive(ConnCmd).LTP\_RX\_CONN\_ACK;}
\]
CHAPTER 4. SPECIFICATION OF THE MAILBOX PROTOCOL

proc LTP_RX_RECEIVE_DATA(s:SeqNr) =
    (ltpRXSend(DataAck(same)).LTP_RX_CONNECTED) + LTP_RX_CONNECTED;

When the receiver is in its connected state, it waits for a DATA_CMD that can be acknowledged. When such a message is received, it replies with sending an acknowledgement for the same sequence number.

It can also be that one of the following messages is received:

1. A DISCONN_CMD message.
2. A CONN_CMD message. The sender was reset and tries to establish a new connection. The receiver runs the procedure to acknowledge this message.

Closing the connection

proc LTP_RX_DISCONN_ACK =
    ltpRXSend(DisconnAck).LTP_RX_IDLE + ltpRXReceive(DisconnCmd).LTP_RX_DISCONN_ACK
    + ltpRXReceive(ConnCmd).LTP_RX_IDLE;

To close the connection the receiver sends a DISCONN_ACK. It could also be the case that the receiver receives another DISCONN_CMD when in this state (e.g. when a timeout occurred). This message is ignored. Finally it could also be possible that a CONN_CMD is received when the sender was reset after sending a DISCONN_CMD.
The Digital Video Link protocol (DVLP) is a connection-less protocol based on UDP/IP that was developed to transmit a stream of grayscale, two dimensional, X-ray images.

The basic setup is shown in figure 1.1. When the X-ray detector is started, it generates a stream of images. These images are often quadrilateral with sides of 512, 1024 or 2056 pixels. Since an image is too large to transmit in one packet, it is divided into strips. This not only provides a way to send an image in small pieces, but also enables the image processing (IP) to start as soon as the first strip arrives. Images are usually sent with a speed of 30 fps (based on a $1024^2$ image). This means that the connection should have a transfer speed of at least 69 Mb/s.\(^1\)

To get a clear view on what DVLP is, first a description is presented in section 5.1 based on the DVLP interface specification in [Fab04]. This description has been used to create a formal specification in mCRL2 (section 5.2).

5.1 Description

To be able to transmit an image stream from a source to a destination, a network is required. Philips has chosen to use standardized network protocols – Ethernet II, IP and UDP – up to the transport layer (see figure 5.1). This choice is quite obvious since standardized network equipment is widely available and often cheaper than custom build equipment.

Figure 5.1 shows that DVLP is part of the transport layer. It has been build on top of the UDP protocol. UDP was chosen because it is relatively easy to implement in hardware and the protocol overhead is relatively small compared to the TCP protocol [RFC81a]. For the remainder of this section we will focus only on the DVLP part of the protocol stack.

5.1.1 Protocol Messages

Based on the interface specification in [Fab04] three message types exist in the DVL protocol:

1. Image Header Message
2. Image Strip Message
3. Error Correction Message

\(^1\)When transmitting 31 lines per strip, 34 strip packets and an image header have to be sent. Standard UDP packet size is 65536 bytes. This results in $35 \times 65536 \times 30 = 69 \times 10^6$ bytes/second.
In a basic DVLP run first an Image Header Message will be sent. This message is followed by around 34 Image Strip Messages and optionally an Error Correction Message after each 15th Image Strip Message. Then the next image will be transferred and a new Image Header Message is sent.

Both the Image Strip Message and the Error Correction Message are defined for a variant where the image data is encoded with 16 bits per pixel and a variant where it is encoded with 8 bits per pixel. Since the conformance testing will only be on the 16-bit variant, the 8-bit variant is omitted.

In the next subsections when a relation between Image Strip Messages and/or Error Correction Messages is discussed, it is left implicit that the messages belong to the same image. If this is not the case, it is mentioned explicitly.

5.1.2 Image Header Message

The transmission of an image starts with sending an Image Header Message. The attributes of the Image Header Message are given in table 5.1 (This is table 1 in [Fab04]).

Some fields need a bit more explanation:
5.1. DESCRIPTION

<table>
<thead>
<tr>
<th>Field</th>
<th>Range</th>
<th>Length in bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOH</td>
<td>0×FFFF (fixed)</td>
<td>2</td>
<td>Start of header identifier</td>
</tr>
<tr>
<td>run_nr</td>
<td>0...0×FFFFF</td>
<td>2</td>
<td>Indicates the run number</td>
</tr>
<tr>
<td>image_nr</td>
<td>0...0×FFFFF</td>
<td>2</td>
<td>Indicates the image sequence number, increasing</td>
</tr>
<tr>
<td>image_size_x</td>
<td>0...0×FFFFF</td>
<td>2</td>
<td>Image size (in pixels) in horizontal direction</td>
</tr>
<tr>
<td>image_size_y</td>
<td>0...0×FFFFF</td>
<td>2</td>
<td>Image size (in pixels) in vertical direction</td>
</tr>
<tr>
<td>strip_size</td>
<td>0...0×FFFFF</td>
<td>2</td>
<td>Size of an image strip (pixels in vertical direction)</td>
</tr>
<tr>
<td>X-ray taste</td>
<td>0...0×0003</td>
<td>2</td>
<td>X-ray taste selection</td>
</tr>
<tr>
<td>ffm_size</td>
<td>0...0×0400</td>
<td>2</td>
<td>Size (in bytes) of free format message</td>
</tr>
<tr>
<td>EOH</td>
<td>0×FFFF (fixed)</td>
<td>2</td>
<td>End of header identifier</td>
</tr>
<tr>
<td>ffm</td>
<td>0...0×FF</td>
<td>0...0×400</td>
<td>Free format message data (application dependent)</td>
</tr>
<tr>
<td>padding</td>
<td>0×00 (fixed)</td>
<td>0...0×5CA</td>
<td>Optional padding up to MTU = 1500</td>
</tr>
</tbody>
</table>

Table 5.1: Image Header Attributes

- **run_nr**: Each time a new video stream is started, the run number is increased to distinguish different streams.
- **image_nr**: For each run the image number starts at 0 and is increased by 1 each time an image was sent.
- **strip_size**: The number of image lines that will be present in each Image Strip Message. Note that $\lceil \frac{\text{image_size}_y}{\text{strip_size}} \rceil$ doesn’t necessarily yield the number of Image Strip Messages that will be sent, because an image line could occur more than once in the Image Strip Messages of a particular image. It will follow from the definition of Error Correction Messages that the strip_size also indicates the number of image lines in that message type.
- **X-ray taste**: Represents the type of X-ray image that was taken. This field has no relevance for the protocol.\(^2\)

5.1.3 Image Strip Message

As already mentioned in the introduction of this chapter, an X-ray image is divided into strips. A strip always contains a round number of image lines. A strip can overlap other strips. Although from a networking point of view sending an image line more than once is waste of bandwidth, from a processing point of view this can be convenient; for example when a convolution filter is applied to each strip as soon as it arrives an overlap is needed for the convolution kernel [Jon97]. Table 5.2 (This is table 2 in [Fab04]) shows the attributes for the Image Strip Message.

\(^2\)One may wonder why a field that has no relevance for the protocol is an explicit field in a message. If this field would have been in the ffm field, this would have allowed the protocol to be used more easily for transmitting other types of images.
### Table 5.2: Image Strip Attributes

<table>
<thead>
<tr>
<th>Field</th>
<th>Range</th>
<th>Length in bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOS</td>
<td>0×EEEE (fixed)</td>
<td>2</td>
<td>Start of header identifier</td>
</tr>
<tr>
<td>image_nr</td>
<td>0…0×FFFF</td>
<td>2</td>
<td>Indicates the image sequence number within the run</td>
</tr>
<tr>
<td>strip_nr</td>
<td>0…0×FFFF</td>
<td>2</td>
<td>Indicates the strip sequence number within the image</td>
</tr>
<tr>
<td>processing_strip_start</td>
<td>0…0×FFFF</td>
<td>2</td>
<td>Indicates from line number of processing strip</td>
</tr>
<tr>
<td>processing_strip_end</td>
<td>0…0×FFFF</td>
<td>2</td>
<td>Indicates to line number of processing strip</td>
</tr>
<tr>
<td>line_start</td>
<td>0…0×FFFF</td>
<td>2</td>
<td>Line start location of image strip</td>
</tr>
<tr>
<td>line_end</td>
<td>0…0×FFFF</td>
<td>2</td>
<td>Line end location (to line, not included) of image strip</td>
</tr>
<tr>
<td>reserved</td>
<td>0…0×00</td>
<td>16</td>
<td>Reserved for future use, shall be set to zero</td>
</tr>
<tr>
<td>EOS</td>
<td>0×EEEE (fixed)</td>
<td>NOB</td>
<td>End of header identifier</td>
</tr>
<tr>
<td>image_data</td>
<td>16-bit per pixel</td>
<td></td>
<td>Image data (pixels), size is $NOB = 2 \times (\text{line.end} - \text{line.start}) \times \text{image.size.x}$</td>
</tr>
</tbody>
</table>

In the message there are two pairs of line numbers. The pair $(\text{line.start}, \text{line.end})$ contains the start and end line numbers of the image data in the message. The pair $(\text{processing_strip_start}, \text{processing_strip_end})$ contains the line numbers of the part of the image that should be processed together. This could span the image data of multiple Image Strip Messages. Figure 5.2 depicts this. Given that the stripsize is fixed, this implies that the difference between a pair of line numbers must be a multiple of the strip size.\(^3\) Since the image height is not required to be a multiple of the stripsize, this observation doesn’t hold for the line numbers concerning the last Image Strip Message of an image.

#### 5.1.4 Error Correction Message

The final message type in DVLP is an Error Correction Message. The purpose of this message is to be able to retrieve the image data of one lost Image Strip Message from a series of images. This means that given a set of Image Strip Messages and the Error Correction Message on this set, one Image Strip Message from the set can be constructed from the Error Correction Message and all other messages in the set. Finally it should be noted that Error Correction Messages are optional for both DVLP transmitters and receivers.\(^4\)

The attributes of the Error Correction Message are almost identical to those in the Image Strip Message. The differences are listed in table 5.3.

---

\(^3\)This observation makes figure 2 in [Fab04] erroneous.

\(^4\)There currently is no detector that includes Error Correction Messages.
5.2 MCRL2 SPECIFICATION OF DVLP

The image data of the Error Correction Message contains the XOR sum (on a pixel by pixel basis) of the transmitted image lines since the last transmission of an Error Correction Message. (see figure 5.3 - This is figure 3 in [Fab04]) Since at least one Image Strip Message is needed to calculate an Error Correction Message, there should be at least one Image Strip Message before an Error Correction Message.

5.2 mCRL2 specification of DVLP

The formal specification of DVLP has been constructed in an incremental fashion. First only the recognition of the messages was specified. This was extended with the validation of the message sequence. And finally a counter for the number of Image Strip Messages was added. It is assumed that, although the UDP protocol is used, the Image Header Message will arrive before any of the Image Strip Messages or Error Correction Messages of that image arrive.
This assumption is correct when all messages are transmitted in the order in which they are put on the network stack and there is just one path from source to destination; this implies that there could for example be a switch between detector and receiver as long as there is only one route between detector and receiver.

The conformance tester will be created to verify a detector. Therefore the specification is created from the point of view of the receiver.

In the descriptions the notion state is used a few times. Here this notion is not a state as in a LTS, but it comprises a set of LTS states that share a same property, e.g. the receiver being idle.

5.2.1 Data types

To be able to recognize the three different message types, the data type DVLPPacket is used to represent them. All structs in DVLPPacket have an ImageNr argument. The data type for this argument was specified exactly the same as the SeqNr in the LTP protocol (see section 4.3). The dvlpHeader struct has an extra argument with the ImageSize. This ImageSize has one of three values: small, normal or large. They are respectively related to images of $512^2$, $1024^2$ or $2048^2$. This implies that a DVLP implementation which sends images with an image size that differs from one of these three formats cannot be validated with this specification. This limitation is not a problem for Philips, because these are the standard image sizes that are used in their equipment.
5.2. MCRL2 SPECIFICATION OF DVLP

\textbf{sort} DVLPPacket = \textbf{struct} dvlpHeader(ImageNr, ImageSize) \\
| dvlpStrip(ImageNr) \\
| dvlpError(ImageNr);

\textbf{sort} ImageNr = \textbf{struct} prev \\
| same \\
| next \\
| other;

\textbf{sort} ImageSize = \textbf{struct} normal \\
| small \\
| large;

\textbf{map} getStripCount : ImageSize \rightarrow \text{Int};
\textbf{eqn} getStripCount(small) = 9;
getStripCount(normal) = 42;
getStripCount(large) = 137;

\subsection*{5.2.2 Actions}

Because communication in DVLP is uni-directional, the only action on the receiver’s side is a receive action. Therefore we define:

\textbf{act} dvlpRX : DVLPPacket;

The action \textit{dvlpReceive} takes as argument a DVLPPacket.

\subsection*{5.2.3 State diagram}

The informal description was used to construct the state diagram in figure 5.4. This diagram visualizes what message sequences may exist in DVLP. The three different transition labels \textit{header}, \textit{strip} and \textit{error} correspond to the \textit{dvlpRX} action with as a parameter resp. \textit{dvlpHeader}, \textit{dvlpStrip} and \textit{dvlpError}.

\subsection*{5.2.4 Procedures}

Each state in the state diagram in figure 5.4 can be represented by a procedure in mCRL2. In this subsection we shall look at each of them in more detail.

\textbf{Init}

\textbf{proc} DVLP_RX,IDLE =
\sum_{sz: ImageSize} dvlpRX(dvlpHeader(next, sz)).DVLP_RX_INIT_RECEIVE(sz);
CHAPTER 5. SPECIFICATION OF THE DVL PROTOCOL

From the initial state the only possible action is to receive an Image Header Message. The ImageNr is the next image number\(^5\).

Received header

\[
\text{proc} \quad \text{DVLP}_\text{RX}_\text{INIT}_\text{RECEIVE}(\text{imageSize} : \text{ImageSize}) = \text{dvlpRX(dvlpStrip(same))).DVLP}_\text{RX}_\text{RECEIVE}(1, \text{imageSize});
\]

In this state only an Image Strip Message may be received containing data for the current image. Figure 5.4 indicates that also an Image Header Message could be received. This message could be received if an empty image, i.e. with dimension \(0 \times 0\), could be transmitted. Due to the fixed image sizes however, this is impossible and is therefore left out of the specification.

Received strip

\[
\text{proc} \quad \text{DVLP}_\text{RX}_\text{RECEIVE}(\text{recv}\_\text{count} : \text{Int}, \text{imageSize} : \text{ImageSize}) = \\
(\text{recv}\_\text{count} < \text{getStripCount}(\text{imageSize})) \to \\
\quad \text{dvlpRX(dvlpStrip(same))).DVLP}_\text{RX}_\text{RECEIVE}(\text{recv}\_\text{count} +1, \text{imageSize}) \\
\quad \diamond \text{DVLP}_\text{RX}_\text{IDLE} \\
\quad + \text{dvlpRX(dvlpError(same))).DVLP}_\text{RX}_\text{RECEIVED}\_\text{ERROR}(\text{recv}\_\text{count}, \text{imageSize});
\]

\(^5\) Where 0 is also seen as a next image number when no image has been transferred yet.
When an Image Strip Message has been received, there are several possible actions:

1. An Error Correction Message is received. This message is not an Image Strip Message and thus the recv_count isn’t increased. Since the Error Correction Message could also be the last message that is received, i.e. when the recv_count already equals getStripCount(ImageSize), this possibility doesn’t depend on the size of the recv_count.
2. Another Image Strip Message is received if the recv_count is less than the number of required Image Strip Messages. This increases the recv_count by 1.
3. An Image Header Message is received with the next image number after the required number of Image Strip messages are received.

Received error

```
proc DVLP RX RECEIVED_ERROR(recv_count : Int, imageSize : ImageSize) =
    dvlpRX(dvlpStrip(same)).DVLP RX RECEIVE(recv_count + 1, imageSize)
    + (recv_count ≥ getStripCount(imageSize))→DVLP RX IDLE;
```

This procedure is called after an Error Correction Message has been received. Since no two Error Correction Messages may be sent after each other, there is no action to receive another Error Correction Message. When the recv_count is large enough, only a new Image Header Message may be received. Otherwise the next Image Strip Message should be received.
Conformance testers

In the previous chapters all ingredients that are needed to construct conformance testers for the Mailbox and DVL protocols have been presented. This chapter will show for each conformance tester how it was made, which design issues occurred and what the test results are.

6.1 Development environment

The environment on which the conformance testers have been developed is called Bambi. It’s configuration is:

- Radisys Procelerant RMS218-7520SM [RAD] with
  - Processor: 2 × Intel Xeon 3.00 GHz with Intel E7520 (Lindenhurst) chipset
  - RAM: 1 Gb
  - Network: 2 × Intel 8254x Gigabit Ethernet controller
- Ubuntu Linux version 6.06 LTS Dapper Drake; kernel linux-686-smp 2.6.15.25

Linux is chosen over Microsoft Windows, the standard OS at Philips Medical Systems, as operating system, because TorX is built for the Linux platform only. This choice causes portability issues in the future. These issues are, however, beyond the scope of this project and are therefore left unresolved\(^1\).

6.2 Test setup

During the testing of the conformance testers, two test setups have been used. The first one is used for all tests that test a software implementation. It is depicted in figure 6.1. Bambi is used to run a conformance tester. The SDE runs the test implementation. Both machines are connected through a gigabit ethernet connection using 1000 base-T.

The second test setup is depicted in figure 6.2. It is used to test the hardware implementation: the Trixell PX5100 v.1.0 DVLP ELEC. This device is the hardware controller of an X-ray detector. It is the way for the user to communicate with the detector and to issue commands to let the detector conduct an X-ray exposure, retrieve an X-ray image, transmit this image - using DVLP -, etc. We will refer to this device as the pixbox. The control interface for the pixbox uses the Mailbox protocol and the set of detector commands in [PVRJ06] as the LTPData in the Mailbox protocol.

\(^1\)This was decided in consultation with my supervisor at Philips.
CHAPTER 6. CONFORMANCE TESTERS

The test setup consists of *bambi* running the conformance tester, a 1000 base-T to 1000 base-X (optical fiber) convertor and the pixbox.

Note that every test setup has a direct connection between the machine with the conformance tester and the machine with the test implementation.

### 6.3 General approach

The general approach that has been chosen in developing the conformance testers is a bootstrapping approach. This approach is chosen, because of the unfamiliarity with TorX and its underlying language - the Tool Command Language [TCL] -, and because a similar approach is chosen for designing the specifications.

Each iteration in the bootstrapping process looks like:

1. Add some feature to the specification
2. Update the conformance tester
3. Test using an implementation
4. Process the results
   - If the results show a fault in the specification, then repeat steps 1-3 and retest
   - If the results show a fault in the conformance tester, then repeat steps 2-3 and retest.
   - If the results show a fault in the implementation, report the fault, get it fixed and retest.
   - If the results show no fault, continue.
5. Backup the iteration
The most difficult step turned out to be to process the results. If a fault occurs, it is sometimes hard to determine what the origin of that fault is. In those cases going through the specification to determine what actions are done before the fault and relating this to the protocol specification [Fab04, Can06] has proved to be a good approach in locating the origin of the fault.

6.4 General design decisions

During the development some general design decisions are made that have influenced how the conformance tester have been constructed.

Type and range checking of header fields is done in TorX

As already explained in section 2.2.3 using the adaptor to do type and range checking of the header fields of the received messages, keeps the specification from getting cluttered with lots of checks that have no direct influence on how the protocol should function.

The price paid for a slightly more complex adaptor matches up to having a simple specification which contains only the parts of the interface specification that really describe how the protocol works.

Verification of the specification is not done

When creating a specification using mCRL2, it is common that the specification is formally verified using its requirements. In this project this has not been done due to two reasons.

The first is that the assignment is about finding out if it is possible to construct a conformance tester based on a specification and not about verifying that the specification has certain properties.

The second reason is that the verification of the specifications could not be fitted in the time schedule. For this project it is more beneficial for Philips to invest time in the development of (prototypes for) conformance testers than to verify their protocols.

6.5 Conformance tester for the Mailbox protocol

The development process for the Mailbox conformance tester consists of 4 iterations:

1. Create the conformance tester for the sender part of the Mailbox protocol.
2. Create the conformance tester for the receiver part of the Mailbox protocol.
3. Combine the conformance testers for sender and receiver.
4. Add testing for time-outs.

Iteration 1 and 2 are tested using a C++ implementation of the Mailbox protocol made by PMS Hamburg [vdH06]. This will be called the Hamburg implementation. This implementation comes with a small test program that can run as a client or a server. When it is in client-mode it actively tries to establish a connection. This works well for testing the iteration
2 conformance tester. In server-mode it will wait for a connection, establish a new connection and echo any DATA_CMD that is received on the incoming connection on the outgoing connection. This works well for testing iteration 1.

Iteration 3 and 4 are not only tested on the Hamburg implementation, but also on the pixbox.

6.5.1 Design issues

Some issues occurred during the development of the Mailbox conformance tester:

1. When trying to visualize a part of the state space while the sequence numbers were modeled as integers in the mCRL2 specification, the state space exploded. The solution was to model the sequence numbers using four states: prev, same, next and other. Because the DVL protocol also had the same problem with its image numbers, this design issue was also implemented there.

2. Imbedding time-outs in the conformance tester in iteration 4 gave some unexpected problems. There are a few ways to specify these time-outs:

   a. **Modeling time in mCRL2**: The mCRL2 language has constructs to model time (see chapter 3). TorX, however, cannot interpret these constructs. This means that only an action *nothing* could be specified in mCRL2 indicating a time-out, but not the actual timing.

   b. **Applying quiescence in TorX**: The \( ioco \) relation can check for quiescence in a certain state (see section 2.1). This means that that state may not have any output actions. If such a state exists, TorX can check for the quiescence by waiting a certain period of time and then generating a quiescence action \( \delta \). In this case however the time-outs occur when an expected output action doesn’t occur. Therefore a state in which a time-out can occur is not a quiescent state.

   c. **Generating a time-out output action in the TorX adaptor**: The TorX adaptor has a notion of channels on which input and output actions are resp. sent and received. Each action is bound to exactly one channel. The plan to solve the time-out issue was to set a timer in the adaptor when a certain input action is sent. This timer triggers an event after a certain period of time. This event can be used to check if the required output action was already received (and passed to the driver) or if a time-out output action should be generated. Although this sounds like a good plan, it doesn’t work, because the adaptor cannot put an action in one of its own channels. For this approach to succeed a large part of the adaptor needs rewriting. Due to lack of documentation and test cases on the internals of the adaptor, this is not a good option.

The issue was resolved by modifying the link to the IUT. This link consists of a program called *UDP*. This program provides a textual interface to send UDP packets to and receive them from a certain IP-address. It was modified to generate the reception of a time-out message some time after an input action was sent. Because both the receiver and sender part of the Mailbox conformance tester use a separate instance of this UDP program and because all messages that can be sent by the sender part of the Mailbox protocol require an acknowledgement, the modification could be done without introducing extra dependencies on the specification.
3. In section 4.3 the sort LTPPacket is declared. This sort has a struct for a DATA_CMD with two arguments. The LTPData argument is mentioned, but not specified further since it is not applicable to the rest of the protocol. When testing, however, a value should be provided for this argument.
In the tests for each iteration the value LINE(offline) has been used. This command is part of the set of detector command in [PVRJ06]. The response to this command in the pixbox, is the command itself. The Hamburg implementation also echoes each DATA_CMD it receives. So this command was chosen, because the specification doesn’t require a change of LTPData when another implementation is tested.

4. In section 3.2.5 of the interface specification [Can06] a (UML) state diagram is given which shows some states and transitions labeled with a input and output action. Is states for example that is possible to go from the Wait for CONN_ACK state to the Connected state by receiving a CONN_ACK and by sending a DATA_CMD. However it does not explain what should happen when after receiving a CONN_ACK a NODE_RST command is received. This omission is not clarified in the document and thus provides the implementor with a choice. Because this choice is not obvious, inconsistencies between implementations can easily occur at such a point. This issue was resolved by discussing with the author of the interface specification and my supervisor on what the intended response should be and documenting this.

6.5.2 Test results
The tests on both implementations yielded two inconsistencies:

- In the Hamburg implementation a deadlock situation is found. The deadlock occurs when the conformance tester issues a DISCONN_CMD on the incoming connection and the Hamburg implementation is waiting for an acknowledgement on the outgoing connection. The implementation then sends a DISCONN_ACK and removes both the incoming and outgoing connections. This causes an infinite wait for an acknowledgement by the thread handling the outgoing connection and thus a deadlock has occurred.
- The pixbox implementation has a curiosity which is valid in the interface specification, but shows that a part of the Mailbox interface isn’t implemented. When a DIS-CONN_CMD is sent to the pixbox, this message isn’t acknowledged. After three times, the conformance tester will draw the conclusion that a time-out has occurred and move on. However, the outgoing connection of the pixbox still keeps the connection with the conformance tester. This is also not wrong given the interface specification [Can06], but illustrates that the interface specification isn’t fool-proof.

6.6 Conformance tester for the DVL protocol
The DVLP conformance tester is created in 3 iterations:
1. Recognize the messages and the image numbers.
2. Check that the messages arrive in the correct order
3. Count the number of strip messages based on the standard sizes.
Two implementations have been used to test this conformance tester. The first one is called the *Nijntjesimulatator* [SWN06]. It’s a tool that is used internally in PMS to create a stream of DVLP messages. Unfortunately it can only send header and strip messages. The other implementation is called *DVLPSend* [Zwe06]. This tool also generates a stream of DVLP messages and can send error correction messages.

One could wonder why the pixbox has not been used to test the DVLP conformance tester. It was not used, because the stream of DVLP messages can only be started if a connection is established using the Mailbox protocol and a command is sent to retrieve and send a X-ray image. Therefore the third conformance tester has been created which combines the Mailbox and DVL protocols.

During the construction of this conformance tester, no faults were found in both implementations. Nevertheless some design issues occurred that have taken some time to overcome.

### 6.6.1 Design issues

1. The image number in each message is also modeled using the structs prev, same, next and other.

2. The strip number in an Image Strip or Error Correction message is left out of the model. This seems strange at first, because the interface specification [Fab04] states that when an image is split up in $X$ strips, all strip numbers in the range $0 \ldots X - 1$ must be used. However due to the assumption of message order made in section 5.2 and the general design decision that type and range checking is done in the TorX adaptor, the requirement in the interface specification can be checked in the adaptor easily.

3. The normal transmit speed of DVLP messages is at least 69 Mb/s (see introduction of chapter 5). This speed doesn’t contain any information on the dispersal of the messages in time. One would expect that the transmission looks like the normal series in figure 6.3; the messages are divided equally over time. However, the actual transmission looks like the burst series; the messages are sent as a burst. This results in much higher maximum transmit speed than expected.

![Figure 6.3: Bursts in DVLP messages](image)

---

2This tool has been constructed by myself in the first month at PMS. This was a good start to get familiar with the DVLP protocol and some standard procedures at PMS.
Due to this burst behavior of all test implementations a problem occurred when testing the conformance tester: messages were lost. The problem was that the UDP buffer of linux was too small and discarded the messages that couldn’t be stored. This problem was solved by increasing the receive buffer to 8 Gb using the following commands:

```
sysctl -w net.core.rmem_max=8388608
sysctl -w net.core.rmem_default=8388608
```

Actually this solution doesn’t solve the problem; it just postpones it. Because TorX isn’t optimized for multi-processor systems, it isn’t fast enough to empty the UDP buffer before a new burst of messages is received. So the buffer is gradually filled and will eventually loose messages. For the tests that are conducted for this project, this solution proved sufficient.

6.7 Conformance tester for the PX5100

Based on the Mailbox and DVLP conformance testers a conformance tester for the pixbox has been created. This conformance tester is able to test if the detector commands from [PVRJ06] that are necessary to retrieve a DVLP image are correct and if the received DVLP image is correct. The iterations for the development of this conformance tester are:

1. Test the startup sequence for the pixbox
2. Add commands for triggering a frame request, i.e. requesting a DVLP image
3. Add functionality from the DVLP conformance tester
4. Optionally add more detector commands such as mode switching

Iteration 1 is based on [PVRJ06] section 5.1. This section describes the startup sequence of the detector. This startup sequence consists of a number of checks that are each terminated by an action which reports the outcome of this check. For example when the initial bootup went right, an action called PASS(BOOT) is sent. These checks are done in a predefined order, so the resulting actions should also be sent in this predefined order.

During the test of this iteration it was found that the pixbox implementation doesn’t adhere to this order. This resulted in a failed test. Because this fault is in the startup sequence of the pixbox, there is no other part of the conformance test that could be tester. Therefore the incorrect sequence was also allowed in the mCRL2 specification so development could continue.

Iteration 2 was developed without notable problems.

At iteration 3 a new inconsistency occurred, which is probably due to the pixbox’s auto-cycling option. When this option is enabled, the pixbox automatically keeps requesting a DVLP image and transmitting it. The Trixell specification states that this auto-cycling should be enabled after the startup sequence completes.

---

3The actual contents of this section is not part of this report, because it’s company confidential.
This is inconsistent with the implementation that was tested. That implementation starts to transmit DVLP images when the startup isn’t completed yet, i.e. the final status action has not been acknowledged. Because it cannot be detected if the auto-cycling option is enabled when this happens, only a strong assumption can be done that this is due to the auto-cycling mode.

The detection of this fault is a good example of the benefit of using a formal specification for conformance testing. The fault was found, because the specification didn’t have an output action for a DVLP message until after the startup sequence completed. Embedding such cases in a conformance tester based on fixed test cases, is not obvious.

After this problem occurred, both errors have been reported to the manufacturer. Because these problems were found and couldn’t be fixed by Trixell very fast, and because project time was getting scarce, development of this conformance tester was ceased.

When Trixell has created an upgrade for the pixbox implementation, retesting it is not a lot of work. Once the test setup is ready, just starting the conformance tester is all.

It is also possible to extend the conformance tester to include other detector commands and eventually create a conformance tester for the whole pixbox (iteration 4). Using this method it is possible to create an exhaustive conformance tester for the entire pixbox that tests the ioco relation.
Quality of a conformance tester

The purpose of a conformance tester is to check if a certain test relation holds between an implementation and a specification. Ideally a conformance tester finds all inconsistencies in an implementation that are possible to find given the test relation. Determining how well an implementation adheres to this ideal situation can be seen as a measure for the quality of a conformance tester.

The quality can be determined by relating the number of inconsistencies that are found after a certain time with the number of inconsistencies that are undetected. A conformance tester that on average has less undetected inconsistencies than another conformance tester after the same periods of time on the same implementations, has a better quality than that other conformance tester.

A problem in determining the quality using this measure is that the number of inconsistencies in a certain implementation has to be known. For any non-trivial specification this is impossible within a reasonable time. To overcome this problem test methods exist to estimate the number of undetected inconsistencies given the number of inconsistencies that are already found. One of these methods is error seeding [KA85, MS89, OH96].

This test method has been used to determine the quality of the Mailbox conformance tester. Section 7.1 treats the theory on error seeding. The test approach is discussed in section 7.2. The input and output of the approach are given in sections 7.3 and 7.4.

7.1 Theory on Error seeding

The IEEE standard definition of an error is a mistake made by a developer [IEE83]. An error may lead to one or more faults. The error seeding test method consists of deliberately introducing faults into an implementation and trying to discover them using a tester. The number of seeded errors and indigenous errors, i.e. non-seeded errors, that are discovered, are used to estimate the number of indigenous errors remaining [KA85, OH96].

If a good estimate can be done, the estimation of the quality is also better. To improve the estimate it sounds logical that enlarging the number of seeded errors will help [KA85]. Several approaches have been tried in this direction: automatic error seeding, line level seeding, "intelligent" seeders [MS89]. Most of them however don’t improve the estimate very much, because most seeded errors are waste. A seeded error is waste when there is another seeded error which tests the same part of the specification in the same way.

A technique that has shown good results in improving the estimate is selective mutation [OH96]. To understand why this technique is better than enlarging the number of seeded
errors, the definition of semantic size is required.

**Definition 7.1**

A program $P$ can be seen as a mapping from an input domain $D$ to an output domain $R$: $D \xrightarrow{P} R$. The specification $S$ on which $P$ is based also has such a mapping: $D \xrightarrow{S} R$. Faults can now be defined as $D_f \subseteq D$ for which $D_f \xrightarrow{P} R \neq D_f \xrightarrow{S} R$. The semantic size of a fault is defined as the relative size of the subdomain of $D$ for which the output mapping is incorrect [OH96].

The selective mutation technique generates mutants, i.e. implementations with a seeded error, that have distinct incorrect subsets of the input and output domain. To put it simpler, no fault is introduced more than once. Furthermore all faults that are introduced are semantically small faults. These are often harder to find and thus lead to better error seeding.

### 7.2 Test approach

To be able to determine the quality of the Mailbox conformance tester error seeding is done using the selective mutation technique.

The basic idea is to take a reference implementation of the Mailbox protocol, the Hamburg implementation [vdH06], and seed errors in this implementation. This reference implementation is first checked by the Mailbox conformance tester until no indigenous errors were found in 5000 steps, so any fault that is found in the test will be due to a seeded error.

Next, 6 mutants are created containing one seeded error each. Also one mutant is created which contains a seeded error, but this error doesn’t introduce any faults, i.e. the inconsistencies are allowed by the specification. Furthermore all mutants have a relatively small semantic size and are distinct.

Each mutant is tested using 3 different random settings of the conformance tester. If a fault is detected within 1000 steps, the test succeeds, else the test fails.

After the first series of tests, the faults caused by the failed mutants are improved in the conformance tester. When this improvement is complete, a second series of tests is run using the same settings.

### 7.3 Seeded errors

The following errors are seeded:

1. On reception of any message, a NODE\_RESET is sent. This should be allowed by the protocol.
2. A sequence number is included in the CONN\_ACK message although it should be zero.
3. On reception of a DATA\_CMD message, the command (LTPData) is added to the DATA\_ACK message.
4. On reception of a DATA_CMD message for sequence number \(X\), a DATA_ACK message is sent for sequence number \(X + 1\).

5. When sending a DATA_CMD, the data length field is one off.

6. When the Hamburg implementation sets up a connection by sending a CONN_CMD message, it doesn’t wait for a CONN_ACK but just assumes that a connection is established.

7. DATA_CMD messages are sent over the incoming connection instead of the outgoing connection.

### 7.4 Test results

In each test run a mutant is tested three times. The seeds that were used to initialize the TorX pseudo-random generator are: 42, 182 and 79. The tables that show the result of a test run not only list the number of steps it took to reach the result for each of these seeds, but also show the average number of steps and the test result. If the test result is PASS, the conformance tester was successful in finding the seeded error. For the correct implementation and mutant 1 PASS means that the conformance tester didn’t find an error. The test result FAILED means that the conformance tester failed to find the seeded error.

#### 7.4.1 Test run 1

<table>
<thead>
<tr>
<th></th>
<th>1 (42)</th>
<th>2 (182)</th>
<th>3 (79)</th>
<th>Avg.</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct</td>
<td>5000</td>
<td>5000</td>
<td>5000</td>
<td>5000</td>
<td>PASS</td>
</tr>
<tr>
<td>Mutant 1</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>PASS</td>
</tr>
<tr>
<td>Mutant 2</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>FAILED</td>
</tr>
<tr>
<td>Mutant 3</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>FAILED</td>
</tr>
<tr>
<td>Mutant 4</td>
<td>25</td>
<td>20</td>
<td>15</td>
<td>20</td>
<td>PASS</td>
</tr>
<tr>
<td>Mutant 5</td>
<td>3</td>
<td>13</td>
<td>6</td>
<td>7</td>
<td>PASS</td>
</tr>
<tr>
<td>Mutant 6</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>FAILED</td>
</tr>
<tr>
<td>Mutant 7</td>
<td>531</td>
<td>369</td>
<td>289</td>
<td>396</td>
<td>PASS</td>
</tr>
</tbody>
</table>

The results for test run 1 show that 5 implementations produced the correct result and 3 implementations failed to do this. Based on the failed mutants the conformance tester has been fixed. The modifications that are done are:

- Check all header fields for correct range (mutant 2)
- Check if the data field contains the same number of bytes as listed in the data length field. (mutant 3)
- Only allow reception of another CONN_CMD after receiving a CONN_CMD (mutant 6)
7.4.2 Test run 2

<table>
<thead>
<tr>
<th></th>
<th>1 (42)</th>
<th>2 (182)</th>
<th>3 (79)</th>
<th>Avg.</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>PASS</td>
</tr>
<tr>
<td>Mutant 2</td>
<td>3</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>PASS</td>
</tr>
<tr>
<td>Mutant 3</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>1000</td>
<td>FAILED</td>
</tr>
<tr>
<td>Mutant 6</td>
<td>63</td>
<td>2</td>
<td>2</td>
<td>22</td>
<td>PASS</td>
</tr>
</tbody>
</table>

After test run 2 the conformance tester still failed to detect one mutant. Investigation showed that there was still a bug in the conformance tester; if the data length was zero, no more checks on the data field were done. This bug was fixed and also mutant 3 passed the tests.

* * *

Although the error-seeding did improve the quality of the Mailbox conformance tester, no statement can be made on the quality yet. At least one other conformance tester for the same protocol should be tested using the test approach of section 7.2, before a comparison can be made. Also the mutations that have been created for this conformance tester don’t cover the entire input and output domains of the protocol. For a good comparison between several conformance tester, this coverage should also be improved.

What is interesting when seeing these test results, is that the number of steps it takes to find an error is very small compared to the size of the state space of the specification. It could be interesting to do further research on whether this is due to the seeded errors or that it might be a benefit for using formal specifications for conformance testing.
Conclusion

This report has described how a conformance tester can be created using TorX and mCRL2, what a possible approach for development is and what problems occurred during development. The conformance testers for the Mailbox protocol and the pixbox even came up with some interesting test results which resulted in a problem report to the manufacturer of the pixbox.

However the goal of this project was to find out if conformance testing for the Mailbox and DVLP protocols can be done more thoroughly and efficiently using a formal specification (see section 1.2). The answer to this problem isn’t a simple yes or no. Let’s dissect the problem and look at some of the pieces in isolation.

8.1 Using a formal specification

There are some real benefits when a protocol is formally specified. The biggest one is that there is no ambiguity in a formal specification. An implementor doesn’t have to use his own interpretation when something is not clear. This probably accounts for most errors in an implementation.

Furthermore a formal specification allows the formal verification of protocol requirements. For example with mCRL2 it can be proven that the DVLP specification adheres to the requirement that no sequence number is used twice within one image. An implementation which conforms to a specification for which certain requirements are proven, also fulfills these requirements.

Unfortunately there are also some disadvantages in using a formal specification. The most important one is that it is quite hard for someone who is unfamiliar to formal methods to understand what is going on in a formal specification. So the learning curve is high.

8.2 Thorough testing

TorX uses an LTS as input for its test algorithm. This algorithm checks the $\text{ioco}$ relation for each trace in the LTS. This is a very thorough approach, because a lot of test cases are verified that would not have been considered using a test method based on a fixed test suite.

The biggest drawback of using an LTS is that the set of possible traces often is infinite. A self-loop in a state already makes the number of traces infinite. The result of this drawback is that the only way to let the conformance test succeed when the number of traces is infinite, is to end the test after some number of steps and report that no faults were found until
now. A statement that no fault exists in the conformance between the specification and the implementation now only holds for the given number of steps.

### 8.3 Efficient testing

Assume that the number of possible test cases for a certain specification is finite. Assume that there we can construct a test suite which comprises every possible test case. To determine if the test method that uses a test suite is more efficient than the test method using a formal specification, a possible measure is the amount of time required to create the test cases and prepare the test setup. For the test method using a test suite this means the time necessary to design and implement this test suite. For the test method using the formal specification this means creating the formal specification and developing the adaptor in the conformance tester.

If a developer is equally proficient in creating a test suite as he is in creating a specification, then for a specification with a reasonable amount of test cases the test method using a formal specification will probably be more efficient. This however is only an educated guess.

* * *

Based on the arguments in the above sections it may be concluded that doing conformance tests using formal specifications has a lot of benefits over a conventional test method that uses a test suite. It’s no holy grail, but it can be another step in improving the quality of software.

Companies like Philips are interested in these kind of techniques, but in general don’t want to spend lots of time and resources to implement them in their projects and educate the developers.

### 8.4 Developments at PMS

Partly due to this project a team at PMS has been created which will look more closely into the use formal methods to describe their protocols and tools which can be developed to work with these models. The team not only uses the outcome of this project as input, but is also talking with a third party called Verum [Ver].

### 8.5 Further research

One of the biggest issues that was encountered during the project was that of the timing requirements. It is currently not possible to use TorX to check these kinds of requirements. A recent article [Ede07] reports that a doctoral student of the UT extended the IOCO theory\(^1\)

---

\(^1\)The contact with Verum is due to a successful pilot project performed by Verum and PMS in 2006. This pilot consisted of the development and verification of control software for a scanner using Verum’s Analytical Software Design (ASD) method [Roo06].
and TorX to support timing [Bri07]. The benefit of this improvement for conformance testing could be looked into.

In section 6.4 it is stated that the verification of the specifications is not done. To improve the quality of the conformance testers and to find open ends in the interface specification, this can also be a possible next step.
Bibliography


[Mat06] A. Mathijssen, *Data types for mCRL2*, Eindhoven University of Technology, October 2006, Private communication.


## List of Figures

1.1 Basic X-ray system setup ............................................. 2
2.1 Problem with i/o-conformance ........................................ 8
2.2 Coffee machine example .............................................. 8
2.3 Components of TorX .................................................... 9
4.1 Mailbox protocol stack ................................................. 17
4.2 Basic message flow in LTP ............................................. 20
4.3 Message is lost in channel ............................................ 21
4.4 Sender is reset ......................................................... 22
4.5 Receiver is reset ....................................................... 23
5.1 DVL protocol stack ..................................................... 30
5.2 Relation between processing strips and image strips .............. 33
5.3 Calculation of the Error Correction Message ...................... 34
5.4 Initial state diagram for DVLP messaging ......................... 36
6.1 Test setup for SW implementations .................................. 40
6.2 Test setup for HW implementations .................................. 40
6.3 Bursts in DVLP messages ............................................. 44
List of Tables

4.1 LTP message format .................................................. 18
4.2 LTP flags ................................................................ 19
4.3 Mailbox message format ............................................. 23
5.1 Image Header Attributes ............................................. 31
5.2 Image Strip Attributes ............................................... 32
5.3 Error Correction Attributes ....................................... 33