Master of Science Thesis
University of Twente
Formal Methods and Tools
Océ-Technologies

Automatic Verification and Analysis of Test Results

of Océ Printers

Richard Rietema
May 2009

Committee:
Dr. M.I.A. Stoelinga (UT/FMT)
Dr. ir. A. Rensink (UT/FMT)
Ing. J. Reinders (Océ-Technologies)

Preface

With this thesis three years of Computer Science and a year of final project end. Both have been periods of hard work, great fun, good talks and learning.

Many people have contributed to these periods and to my thesis in various ways. I would like to thank all of them.

First of all, I want to thank my main supervisor Mariëlle Stoelinga. She guided me during the first five months and during the last two months. In between these periods she took maternity leave. Mariëlle gave lots of interesting suggestions and provided many useful remarks.

I also want to thank my supervisor Arend Rensink. He provided many practical ideas and useful remarks on drafts of this thesis. This thesis would not look the same without his remarks.

Furthermore, I would like to thank my Océ supervisor Jac Reinders. For the valuable moments of his time, for his remarks, and the technical documents and logfiles he provided. I would like to thank all colleagues for their questions and remarks, Lou, Joost, Peter, Ton, Reinier and Sander.

Also thanks to my colleague students at Océ, Alvaro, Esmée, Eugen, Klemens and Ralph. They made the period of final project pleasant.

I would like to thank all other people who made this period to an enjoyable time. The people from 'De Brug': family Veldhuizen, family Stephanus, and Bea. Thanks for the many meals and the good talks. And the people from Nieuwegein, for the meals and for encouraging me to write this thesis.

I would like to thank my landlady in Venlo, who used to turn the heat up before 6 am, to warm up my room.

Finally I want to thank my parents, family and friends, for their love and support, even when I only saw them once a month or less.

Nieuwegein, May 2009
Richard Rietema

Abstract

This thesis describes the automatic verification and analysis of a printer of Océ by means of test results in the form of logfiles.

Océ is a company that develops high performance, state-of-the-art printers that produce up to 250 pages per minute. To test the complex software within these printers, all printer processes write their actions into a global logfile. When executing tests, an automatic analysis and verification of logfiles is useful, especially when these tests are automated and performed on printers under development. Currently, these logfiles are often inspected manually, which is a cumbersome and time-consuming task.

For automatic verification and analysis the logfile is transformed to a log model, which only contains log statements of functionality for tests relevant. The same (relevant) functionality is modeled in a specification. This specification consist of a composition of reference, synchronization and test-specific models representing protocols, relations between protocols, and behaviour adjusted to tests. It is defined in the formal language LOTOS. The relation between the log model and the specification is a trace membership relation, which means that a log model, if it represents correct printer behaviour, is a member of the traces formed by the specification.

This relation is implemented in a tool chain, consisting of a preprocessor, an editor, two compilers, and a Testlog verifier. The preprocessor transforms the logfile into a log model. LOTOS compilers compile the LOTOS specification, which is made in the LOTOS editor, into C code, and the Testlog verifier checks whether a relation between the log model and the compiled specification exists. Depending on the printer behaviour a verdict, true or false, is returned. In the latter case, a sequence of transitions is given which leads to the first unexpected transition in the log model.

With this tool chain, three protocols with connecting relations were analyzed with logfiles from 15.000 to 700.000 lines. All known errors where identified correctly and in almost all cases the verification times where short (< 16 minutes). In some cases, with many protocol instances in parallel, they exceeded one hour.

Contents

ABSTRACT 5

CONTENTS 7

FIGURES 9

TABLES 11

TERMS AND ABBREVIATIONS 13

1 INTRODUCTION 15

1.1 INTRODUCTION TO THE PROBLEM 15
1.1.1 Problem 16
1.1.2 Motivation 16
1.1.3 Solution 16
1.1.4 Results 18
1.2 BACKGROUND 18
1.2.1 Structure 19
1.2.2 Protocols and logging 20
1.2.3 Test 22
1.2.4 Verification 23
1.3 OUTLINE 24

2 FORMAL PRELIMINARIES 25

2.1 LABELED TRANSITION SYSTEMS 25
2.1.1 Composition of LTSs 26
2.1.2 Relation between LTSs 28
2.2 LOTOS 30
2.2.1 Basic LOTOS 30
2.2.2 Full LOTOS 32
2.2.3 LOTOS example 35

3 METHOD 37

3.1 Log model 37
3.2 Specification 38
3.2.1 Reference model 39
3.2.2 Synchronization model 40
3.2.3 Test-specific model 42
3.3 Correctness relation 43
3.4 Verification 44
3.4.1 On-the-fly composition of specification. 45
3.4.2 Separate verification of specification models 45
3.4.3 Composition of specification with log model 45

4 IMPLEMENTATION 47

4.1 Specification language 47
4.2 The tool chain 48
4.2.1 The LOTOS editor 49
4.2.2 The preprocessor 50
4.2.3 The Testlog verifier 51
4.3 Alternatives 52
4.3.1 Language 52
4.3.2 Tool 52

5 FEASIBILITY STUDY 55

5.1 Protocols 55
5.1.1 Status protocol 55
5.1.2 Print protocol 56
5.1.3 Data protocol 57
5.2 Relations between protocols 58
5.2.1 Status Print synchronization model 59
5.3 Test specific situations 59
5.3.1 Number of sheets test-specific model 59
5.3.2 Time test-specific model 60
5.4 Composition 61

6 EVALUATION, CONCLUSIONS AND RECOMMENDATIONS 63

6.1 Evaluation 63
6.1.1 Formalization of the logfile 63
6.1.2 Creation of the specification 64
6.1.3 Comparison of the formalized logfile with the specification 65
6.2 Experience 66
6.2.1 Creation of specification 66
6.2.2 Detection of failures 67
6.2.3 Acceptance in Océ 67
6.3 Conclusions 67
6.4 Recommendations 68

REFERENCES 69

Figures

Figure 1: Formal specification of a printer.. 17
Figure 2: Formal log model of a printer 17
Figure 3: Printer in environment 19
Figure 4: Structure of a printer 20
Figure 5: A part of a logfile... 21
Figure 6: Formal verification; relation between model and specification 23
Figure 7: LTS of Status protocol.. 26
Figure 8: LTS of Print protocol 27
Figure 9: LTS of parallel composition of LTSs Status and Print 27
Figure 10: Trace through LTS of Figure 9... 28
Figure 11: Trace inclusion 29
Figure 12: LOTOS specification of Status protocol 32
Figure 13: LOTOS specification Print protocol... 32
Figure 14: Specification with LOTOS data type instances 33
Figure 15: Arbitrary LOTOS process 34
Figure 16: Alternative specification with value passing and constraint........... 34
Figure 17: LOTOS specification 35
Figure 18: Log model in sequence format 38
Figure 19: Reference model of Status protocol... 39
Figure 20: Reference model of Print protocol 40
Figure 21: Synchronization model for Status and Print protocol................... 41
Figure 22: Specification, containing reference and synchronization models ... 42
Figure 23: Test-specific model test print of one sheet................................. 43
Figure 24: Test-specific model test specific time 43
Figure 25: AVATR tool chain 49
Figure 26: Preprocessor, Perl script... 50

Tables

Table 1: Results verification and analysis of Status model 56
Table 2: Results verification and analysis Print model 57
Table 3: Results verification and analysis Data model 58
Table 4: Results verification and analysis Status Print synchronization model 59
Table 5: Results verification and analysis Number of sheets test model 60
Table 6: Results verification and analysis Time test-specific model 60
Table 7: Results verification and analysis composition specification model 61

Terms and abbreviations

Logfile	Text file in which internal printer actions are written, **20**
LOTOS	Language Of Temporal Ordering Specification, **30**
LTS	Labeled Transition System, **25**
Message	A communication package of a protocol, **20**
Model	A formal representation of a description of an implementation, **16**
Protocol	A method of communicating information between two or more entities, **20**
Requirements	A description of what a particular product or service should be or do, **17**
Specifications	The formalized requirements, **16**
Trace	A sequence of transitions in an LTS, **28**
Trace inclusion	All traces of an LTS are also traces of another LTS, **29**
Trace membership	A trace is an element of the set of traces of an LTS, **29**
Transition	A state change in an LTS, **25**
Validation	The check whether the formalized implementation is correct with respect to the implementation and the specification with respect to the requirements, **23**
Verification	The check whether the formalized implementation is correct with respect to the formalized requirements, **23**

The difference between theory and practice is a lot bigger in practice than in theory

(Peter van de Linden)

1 Introduction

In the last decades, printers of Océ have become more and more complex [1], making it more and more important to perform testing. Printer tests also have become more complex in order to keep up with the complexity of printers. Printer testing produces log information, originally used by engineers for debugging. Since these logfiles are becoming larger and larger, analysis of these files is cumbersome; it is a task for specialists, which is often time consuming.

At the same time, formal (mathematical) techniques are more and more used in industry, to evaluate correctness of systems [3, 4].

Due to these trends the question arises to perform logfile verification automatically and to analyze erroneous logfiles automatically by presenting failures.

This master thesis is the result of research to formally support automatic verification and analysis of logfiles produced by a printer of Océ.

Organization of this chapter
First, Section 1.1 introduces the project by giving a description of the problem and the motivation for the project together with a sketch of the solution and the results obtained. Subsequently, Section 1.2 describes the background of the problem, consisting of an overview of the structure of the printer, the communication mechanism used, the test, and the verification of the printer. This chapter ends with an outline of this thesis in Section 1.3.

1.1 Introduction to the problem

The project, described in this thesis, is summarized as: *the development of a method and a tool to formally support and simplify, the log verification and analysis process of a printer of Océ*. The project is called: Automatic Verification and Analysis of Test Results, shortly AVATR.

The printer used in this project contains several processes, which communicate via different protocols. Each process writes executed protocol actions to a global logfile. In this logfile, which grows easily to 25 Megabyte, all logged actions are ordered by the time they are executed and mixed up with actions of other protocols. A logfile shows in detail what actions are performed in the printer, and is currently used for:

- verification, which is an automatically performed count of particular log statements;
- analysis, which is a manual search for a missing log statement, a wrong order or a wrong timing of log statements.

1.1.1 Problem

In this verification, an obligatory or forbidden order of log statements is not taken into account, nor can this verification prove the correctness of a logfile, since it is only a count of log statements. Hence this verification is incomplete.

Furthermore, the analysis in case of incorrect printer behaviour is performed manually. This can take a while, depending on the experience of the involved engineer, which makes analysis labor-intensive.

1.1.2 Motivation

The drive behind the formal support and simplification of this verification and analysis is twofold.

- Maximization of the value of verification verdicts.
- Minimization of labor-intensive analysis.

Since the logfile contains detailed information about certain processes in the printer, this can be used to maximize the amount of properties in the verification. This detailed information can also be used to analyze which printer process or protocol fails, in case of an error.

1.1.3 Solution

In order to maximize the value of the verification verdicts and to minimize the labor intensive analysis, a method has been developed and implemented in a tool chain. This method requires a notion whether a logfile is correct or not. This notion is given by a formal specification, which is a formalization of certain aspects of the requirements of the printer. It is created in the formal specification language LOTOS, which is described in chapter 2.

Specification
Intuitively, this formal specification is a collection of all correct sequences of actions of the printer (see figure 1).

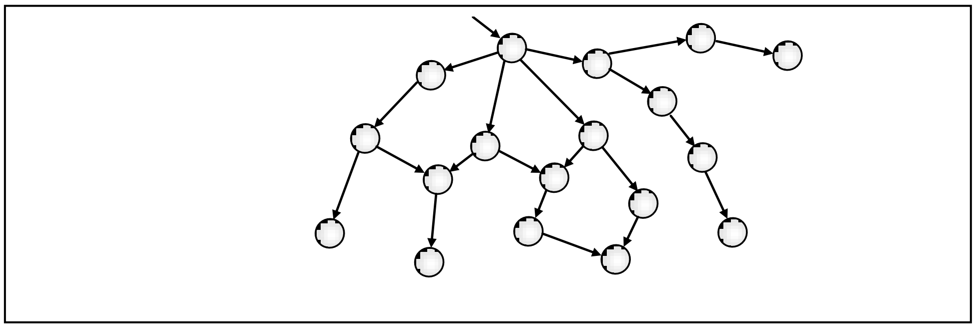


Figure 1: Formal specification of a printer

The formal specification exists of the composition of:

- one or more small reference models;
- zero or more synchronization models;
- zero or more test-specific models.

A reference model is an unambiguous description of a coherent subset of the requirements of the printer, typical a protocol. For each verification and analysis, this reference model can be different or it can be a combination of more models. Between reference models, synchronization constraints can exist. For example, an obligatory action order between actions of different reference models. These constraints are modeled in special synchronization models and are taken into account with the verification. Test-specific models contain test-dependent information, e.g., the number of printed sheets. With this information, the specification is matched to a specific test scenario.

Verification and analysis

The logfile, which is (intuitively) one sequence of actions of the printer, has been filtered and formalized to keep only actions of protocols which are modeled in the specification. The resulting (formalized) logfile, a log model (see Figure 2), is an element of the collection of action sequences of the specification, if the printer behaves correctly.

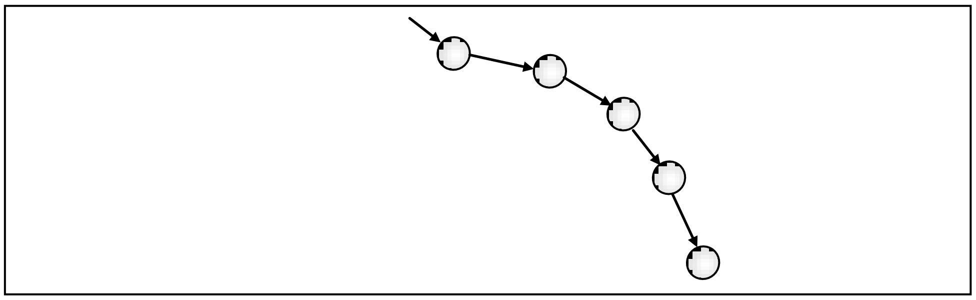


Figure 2: Formal log model of a printer

Formally supported verification is the check whether this is true or not. Formally supported analysis determines the first action in the logfile that causes a mismatch with the best fitting element of the collection of specification sequences.

Implementation

This method, to verify and analyze a printer by means of a logfile and a formal specification, has been implemented in a tool chain, which consists of five parts:

- a preprocessor, which filters and formalizes the logfile;
- a LOTOS editor, in which a specification can be created;
- a LOTOS functional compiler, which translates the functional part of the specification;
- a LOTOS abstract data type compiler, which translates the abstract data type instances of the specification;
- a Testlog verifier, which verifies and analyzes the formalized logfile with the created specification.

The preprocessor is implemented in the scripting language Perl [18]. The LOTOS editor is a text editor called VIM [2]. For the LOTOS compilers and the Testlog verifier the tools Caesar, Caesar.adt and Exhibitor, from the CADP (Construction and Analysis of Distributed Processes) toolbox are used [3, 4]. These tools are based on the formal description language LOTOS [5, 6, 7, 8].

1.1.4 Results

The main results of this thesis are summarized as follows.

- A method has been developed to transform a logfile into a log model. This transformation filters unneeded details out and formalizes used printer actions;
- A method has been developed to create a specification, consisting of reference, synchronization, and test-specific models, which describe (relevant) parts of printer requirements;
- A method has been developed to verify and analyze the log model with the specification efficiently.

These results have been implemented in the AVATR tool chain, which has lead to a proof of concept.

1.2 Background

The printer studied in this thesis is an Océ high performance printer, capable of printing 250 pages per minute. Since this printer is not released yet, no specific details are given. However, the method developed is applicable to many printers.

A typical environment of the printer studied, consists of one hundred personal computers (pc's) in an office. When a user gives a print command from a pc, for example a document from Microsoft Word, this is sent to the printer over the local area network (LAN). The printer processes this print job, and shows the printer status on the user interface (e.g., initializing, warming, printing, standby).

For research and development the printer sends internally performed protocol actions over the LAN to a log server, which stores this information in a global logfile, on a network drive (see figure 3).

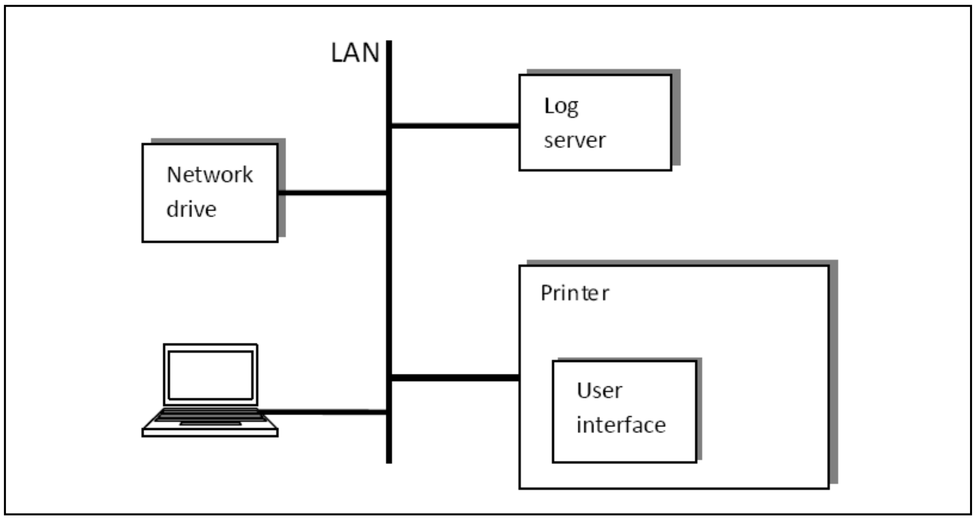


Figure 3: Printer in environment

1.2.1 Structure

The printer consists of a well-defined structure, which globally exists of a Controller and an Engine (see figure 4). The Controller is, besides image processing, responsible for the control of the Engine and the information transfer between the user and the Engine. In more detail, it:

- communicates with a user to provide information about the configuration (e.g., paper size, print quality, status);
- assigns print jobs from the user to the Engine;
- controls the Engine status to correctly execute send print jobs.

The configuration and error information provided by the Controller to the user is provided by the Engine to the Controller.

The Engine is responsible for the actual printing, it is the part where the high-level control messages are translated into low-level system signals. To do so, the Engine consists of several processes:

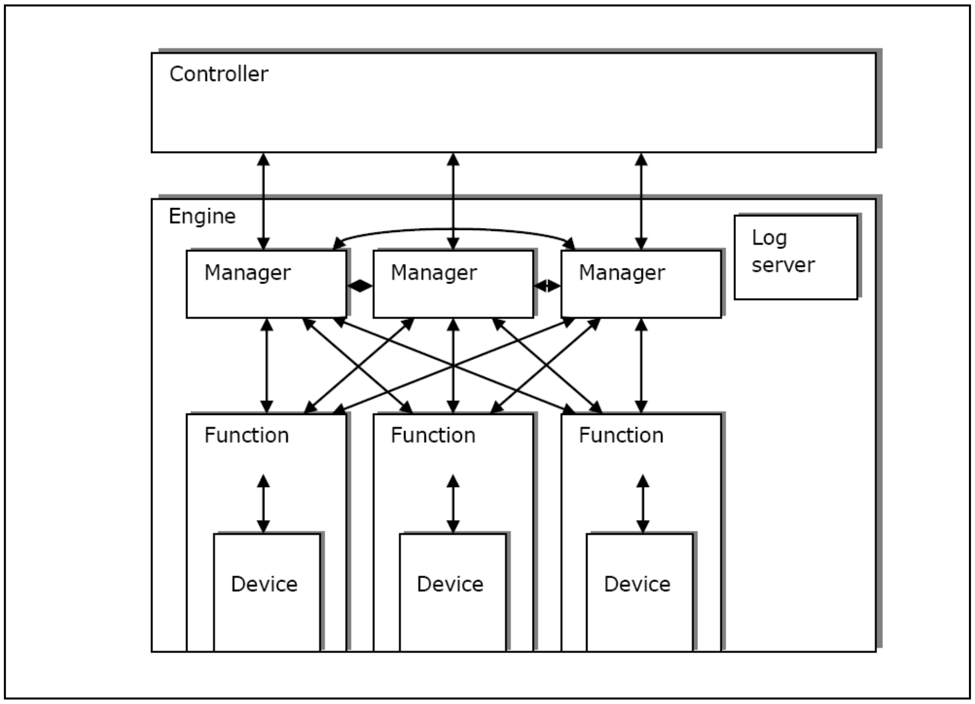


Figure 4: Structure of a printer

- Managers form an interface from the Engine to the Controller, to which a Controller can connect. Each Manager represents a functional aspect of the printer (e.g., error handling, page printing, status control). The required behaviour of a Manager can be established by the use of Functions and other Managers.
- Functions are a top-level decomposition to help managers dealing with dependencies and structures of hardware components. A Function controls a coherent set of devices, for instance the paper input. It 'converts' Engine physical parts to Engine functional parts.
- Devices consist of a software part and a hardware part. The software part of a Device is called a Device driver. It controls a coherent set of sensors and actuators, which can be on functional, temporal, or physical level. The hardware part of a device represents an actual part of the Engine hardware, and belongs to the hardware of the printer.

1.2.2 Protocols and logging

The processes of the Engine (managers, functions and devices) and the Controller communicate to each other by means of different protocols, depicted by black arrows in Figure 4.

A protocol is an agreed-upon method of communicating information between two or more entities, using an underlying service or medium [9]. There are two relevant ingredients in the used protocols:

- the messages, and their intended meaning;
- the order in which messages should be exchanged.

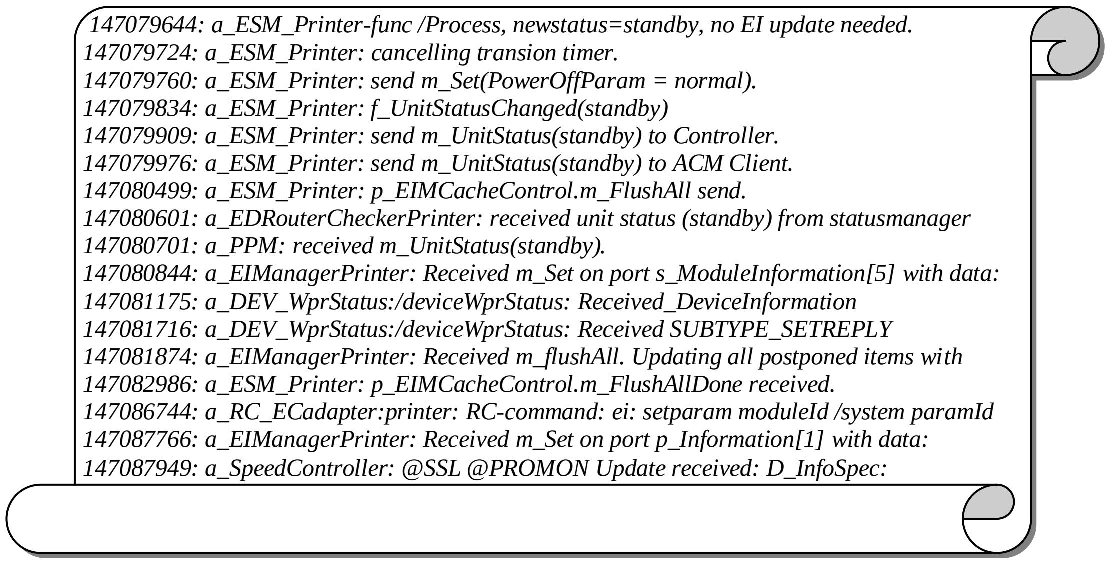
Messages contain a label and often an identification number, which distinguishes the protocol message from messages with an equal label. Identification numbers are also used to match the message with a message response.

Each protocol has its own distinct set of messages, there are no message sets that have messages with an equal label and identification number. In each protocol and between different protocols the combination label and identification number is always unique. Hence printer protocols form a deterministic system.

Protocol messages are not only sent from one process to another, but also to a log server in the Engine (see figure 4). This log server appends the current time to the message and sends the result represented as text string, via the Controller, to the log server outside the printer. In the printer a small buffer stores the latest log messages, which can be used to detect failures by customers. The level of messaging can be adjusted to relief printer processors.

Log messages which are sent to the log server form a large logfile, in which the message, the time the message is sent and the involved processes are written (see figure 5). Because this logfile is a sequential representation of parallel communications, and because some messages have to travel a longer way to the Engine log server than others, it can happen that protocol messages are stored in a different order and that timestamps are assigned different from their actual time.

When timestamps are not assigned following the actual time (i.e., an action which is performed earlier than another one, gets a lager timestamp) the logfile is not a correct representation of the order of actions taken in the printer. When these incorrect ‘timed’ logfile actions do not have a relation to each other this is not a problem. However, when they do have a relation, verification of these messages will fail. This case is very rare in practice because the extra travel time of log messages is small corresponding to the time between the log messages.



```
147079644: a_ESM_Printer-func /Process, newstatus=standby, no EI update needed.
147079724: a_ESM_Printer: cancelling transion timer.
147079760: a_ESM_Printer: send m_Set(PowerOffParam = normal).
147079834: a_ESM_Printer: f_UnitStatusChanged(standby)
147079909: a_ESM_Printer: send m_UnitStatus(standby) to Controller.
147079976: a_ESM_Printer: send m_UnitStatus(standby) to ACM Client.
147080499: a_ESM_Printer: p_EIMCacheControl.m_FlushAll send.
147080601: a_EDRouterCheckerPrinter: received unit status (standby) from statusmanager
147080701: a_PPM: received m_UnitStatus(standby).
147080844: a_EIManagerPrinter: Received m_Set on port s_ModuleInformation[5] with data:
147081175: a_DEV_WprStatus:/deviceWprStatus: Received_DeviceInformation
147081716: a_DEV_WprStatus:/deviceWprStatus: Received SUBTYPE_SETREPLY
147081874: a_EIManagerPrinter: Received m_flushAll. Updating all postponed items with
147082986: a_ESM_Printer: p_EIMCacheControl.m_FlushAllDone received.
147086744: a_RC_ECadapter:printer: RC-command: ei: setparam moduleId /system paramId
147087766: a_EIManagerPrinter: Received m_Set on port p_Information[1] with data:
147087949: a_SpeedController: @SSL @PROMON Update received: D_InfoSpec:
```

Figure 5: A part of a logfile

Note that the logfile also can contain incorrectly ordered messages as result of software failures (i.e., incorrect communicating software), hardware bugs or failures due to incorrect external parameters (e.g., usage of wrong paper or too high environment temperatures).

1.2.3 Test

To be sure different printer processes interact correctly with each other, a printer is extensively tested. Tests are performed in three stages, the stages of the V-model [10] according to which the printer is developed.

- A model of the printer is built, which simulates the required system.
- An embedded prototype is built, with code generated from the model.
- A final product (i.e., a printer) is built by gradually replacing the experimental hardware of the prototype by real hardware, until the printer is build in its final form as it will be used and mass produced.

Each of the above printer appearances (model, prototype and final product) follows a V-development cycle itself, including design, build and test activities. This implies that the complete functionality can be tested for the models as well as for the prototype and the final product. However, certain detailed properties cannot be tested on the model and must be tested on the prototype or the final product, for instance, the impact of environmental conditions.

The tests performed on the prototype and the final product have to deal with the whole embedded system, instead of only software in the model. These tests produce logfiles besides printed paper.

The evaluation of these tests makes use of these produced logfiles. The evaluation is partly manual and partly automatic. For instance, the check whether bitmaps are printed correctly or not is manual; the check whether the right amount of sheets is printed or not is automatic.

Tests can result in a correct or an incorrect verdict. An incorrect verdict exhibits incorrect printer behavior, which can be divided into:

- test-independent incorrect behaviour;
- test-specific incorrect behaviour.

Test-independent incorrect behavior is in general wrong behavior (e.g., print a sheet without first warming the Engine). This printer behaviour is for all tests unacceptable. It can be recognized by an incorrect order of protocol messages or missing protocol messages, without knowing the performed test.

Test-specific incorrect behavior is wrong depending on a certain test (e.g., five pages printed instead of three). To recognize an error of this kind, additional test information is needed, e.g., a logfile which shows one printed sheet can be correct with respect to the used protocols, but not with respect to the specific test.

1.2.4 Verification

By performing a test, processes in the printer produce log statements. These log statements are an unambiguous representation of the protocol messages send in the printer during the test, and can be used to verify the behaviour of the printer.

Definition
Verification is the mathematical proof of a formal relation between the formal representations of the implementation and the requirements [9] (see figure 6).

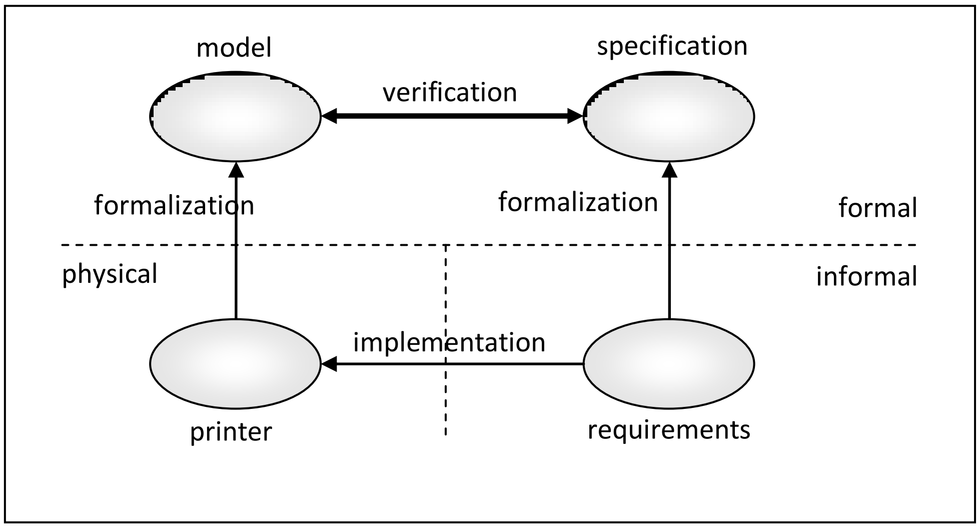


Figure 6: Formal verification; relation between model and specification

The requirements of the printer are formalized, which means: translated from a description in an informal language to a description in a formal language. A formal language is defined by a formal syntax, and has associated semantics, which give precise meaning to expressions in the syntax. The printer itself is also formalized, which means that a formal model is created from the implementation. This model is derived from the logfiles of the printer.

The formal relation between the (formal) specification and the (formal) model defines the behaviour that is allowed in the model by the specification. This relation can be that the model must be included by the specification, or that the model and the specification are equal.

The specification is created manually from the requirements of the printer. The validation of this specification, the check whether the formalization is correct, is a manual task. The model of the printer is automatically created from the logfile and should correspond to the behaviour of the printer.

1.3 Outline

The remaining part of this thesis is divided into four chapters:

- Chapter 2 (Formal preliminaries) describes labeled transition systems, the composition of these systems and the relation between them. Furthermore, it describes the formal description language LOTOS. This language is used to describe printer requirements.
- Chapter 3 (Method) describes the developed method. It describes the preprocessing of the logfile, the creation of a specification, consisting of reference, synchronization and test-specific models and the composition of these models. Furthermore it describes the verification and analysis of the log model with the specification and several alternatives for it.
- Chapter 4 (Implementation) describes the developed tool chain, which consists of a preprocessor, a LOTOS editor, two LOTOS compilers and a Testlog verifier. Also alternatives for this implementation are described.
- Chapter 5 (Results) shows the practical use of the method and the tool chain. It describes four cases, which describe a subset of the functionality of the printer, and gives results obtained.

This thesis ends with an evaluation about the AVATR project, conclusions about current results and recommendations for improvement of profit of the AVATR method and tool chain.

Seek simplicity and distrust it

(Alfred North Whitehead)

2 Formal preliminaries

Formal methods is a term used for mathematically-based techniques, used for specification, development and verification of software and hardware systems. They provide formal, unambiguous, models and precisely defined and proved methods that provide a means to verify, validate and test these models. These methods are increasingly used in industry to evaluate correctness [11, 12].

This chapter presents the required preliminaries to understand the method this thesis describes.

Organization of this chapter
Section 2.1 gives information about labeled transition systems, it includes the definition and the composition of these systems and the relations these systems have. Section 2.2 describes the specification language LOTOS, the implementation of the formal definitions in this description language. Each section in this chapter is illustrated with an example.

2.1 Labeled transition systems

Definition
A labeled transition system (LTS) [13] is a 4-tuple ⟨S, L, T, s0⟩ where

- S is a finite, non-empty set of states;
- L is a finite set of labels;
- $T \subseteq S \times (L \cup \{\tau\}) \times S$ is the transition relation;
- s0 $\in$ S is the initial state

A transition labeled μ from state *s* to state *s*′, i.e., (*s*, μ, *s*′) $\in T$ is written as: $s \xrightarrow{\mu} s'$. This is interpreted as: "when the system is in state *s* it may perform action μ and go to state *s*′". The labels in *L* represent the observable actions of a system; they model the systems' interactions with its environment. Internal actions are denoted by the special label $\tau \notin L$; τ is assumed to be unobservable for the systems environment. A series of transitions in which at least one with label σ and zero or more internal actions, from state *s* to state *s′*, is written as: $s \overset{\sigma}{\Longrightarrow} s'$. This is interpreted as: "when the system is in state *s* it may perform zero or more internal actions, one action *σ,* zero or more internal actions and go to state *s′*".

An LTS can be represented by a graph, where nodes represent states and labeled edges represent transitions.

Example
Figure 7 gives an example of an LTS represented by a graph. This figure gives a simplified representation of the Status protocol of a printer.

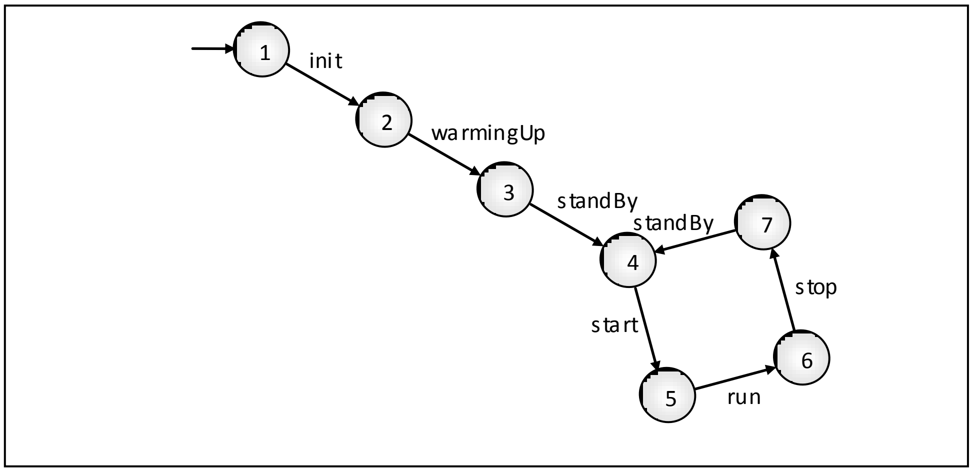


Figure 7: LTS of Status protocol

Figure 7 represents states by numbers and labels by character strings. The LTS has a set of states, *S*: {*0*, *1*, *2*, *3*, *4*, *5*, *6*}, a set of labels, *L*: {*init*, *warmingUp*, *standBy*, *start*, *run*, *stop*}, a set of transitions, *T*: {(*0, init, 1*), (*1, warmingUp, 2*), (*2, standBy, 3*), (*3, start, 4*), (*4, run, 5*), (*5, stop, 6*) , (6*, standBy, 3*)}, and an initial state, s_0: *0*.

2.1.1 Composition of LTSs

A composition of different LTSs, LTS_1 and LTS_2, is written as: LTS_1 |[*G*]| LTS_2, where *G* is a set of labels (see Section 2.2.1 below). This is interpreted as: "LTS_1 and LTS_2 synchronize on all transitions with a label of the set *G*, all transitions with other labels are interleaved" (i.e., in the composition each order of these transitions is possible). It means full interleaving if the label set *G* is empty (*G* = ∅), full synchronization, if the label set *G* equals the union of the label sets of the synchronizing LTSs and if this label sets are equal ($G = L(LTS_1) \cap L(LTS_2)$ *and* $L(LTS_1) = L(LTS_2)$), and partial interleaving otherwise. Synchronization can occur if both systems are able to perform a transition with a similar label. This implies that LTSs can also both block the occurrence of synchronization. Interleaving occurs when one of the LTSs can perform a transition that is not a synchronizing transition.

Example
A composition LTS is created from the LTS of Figure 7 with the one in Figure 8. The latter representing a simplified version of the Print protocol of a printer (see figure 8). The transitions *start* and *stop* in the LTS in this figure are not part of the protocol, they are appended for this example.

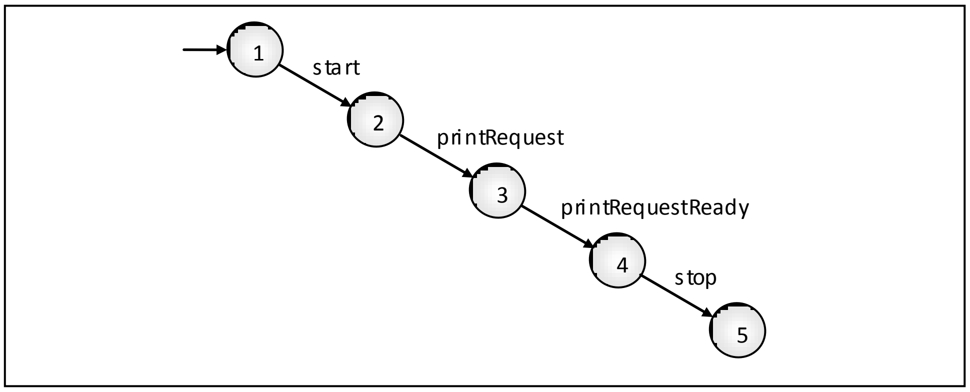


Figure 8: LTS of Print protocol

The resulting LTS, created using the composition operator |[*G*]|: Status |[*start*, *stop*]| Print, contains 11 states and 13 transitions (see figure 9).

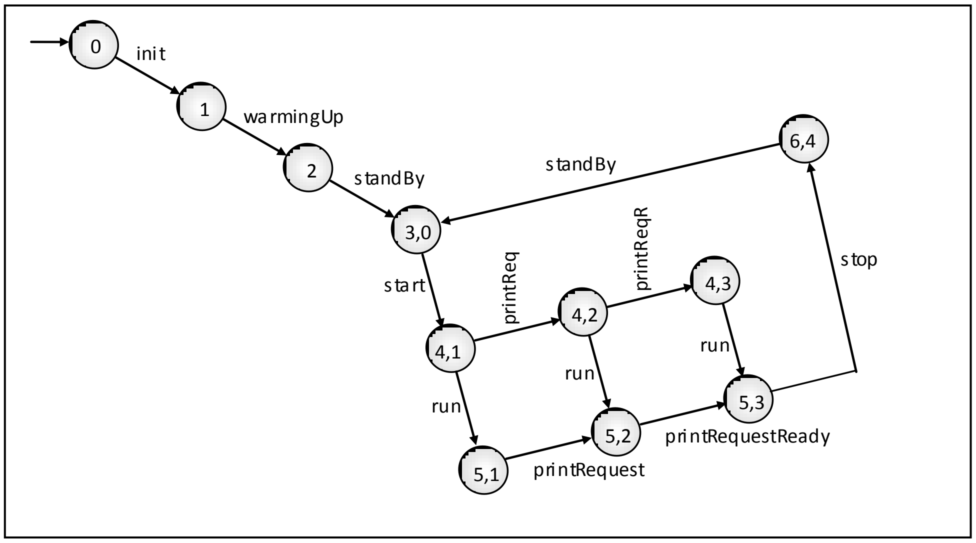


Figure 9: LTS of parallel composition of LTSs Status and Print

In this composition only transitions *start* and *stop* are synchronizing, all other transitions are interleaved. However, the LTS of the print protocol (figure 8) does not have transitions before *start* and after *stop*, so the only actually interleaving transitions in Figure 9 are *run*, *printRequest* and *printRequestReady*.

2.1.2 Relation between LTSs

LTSs are used to model the behaviour of systems, such as distributed systems and protocols. These systems, formalizations of processes, can be compared using formal relations, which are known from literature [14, 15]. In this thesis only the trace inclusion relation is important.

The trace inclusion relation compares different LTSs by means of their traces. A trace is defined as a sequence of actions which exist in a specific LTS.

Definition
A formal notation of a trace is:

$$\text{traces}\ (s) = \{ \sigma \in L^* \mid s \overset{\sigma}{\Longrightarrow} \}$$

where *s* is an arbitrary LTS, *L* the label set of *s* and L^* the set of sequences in *L.*

Example
Figure 9 contains infinitely many traces, one of them is depicted below (see figure 10).

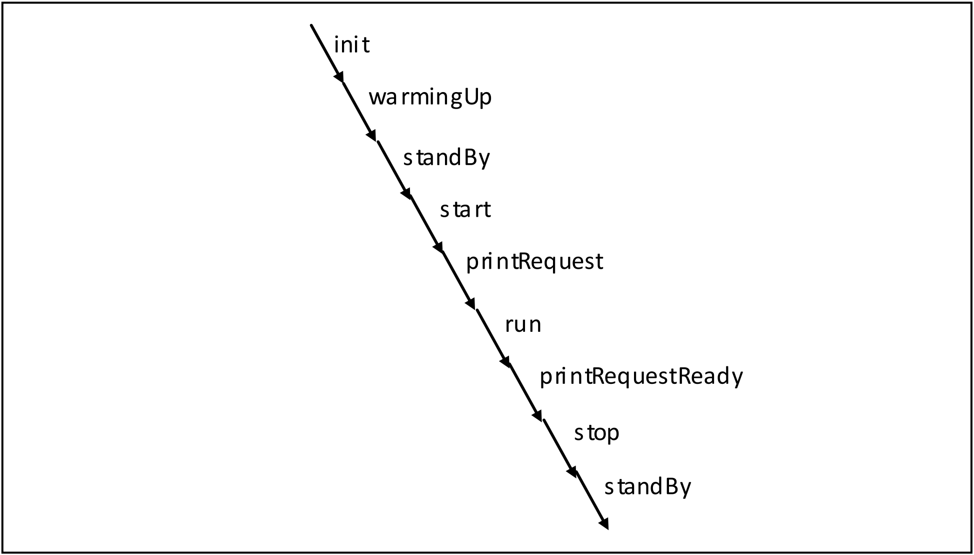


Figure 10: Trace through LTS of Figure 9

The trace in Figure 10 starts with the first transition of the LTS of Figure 9, and forms a trace through. The cycle in the LTS of Figure 9 causes an infinite number of traces.

A trace inclusion relation is intuitively defined as follows: if LTS *P* is *trace included* in LTS *Q*, the traces formed by the transitions of *P*, are also traces in *Q*. The other way around is not required, neither forbidden.

Definition
A formal notation of the trace inclusion relation [13] is:

$$P \leq_{tr} Q =_{def} \text{traces}(P) \subseteq \text{traces}(Q)$$

where LTS *P* is trace included by LTS *Q.*

Example
Figure 11 shows two LTSs, LTS P and LTS Q.

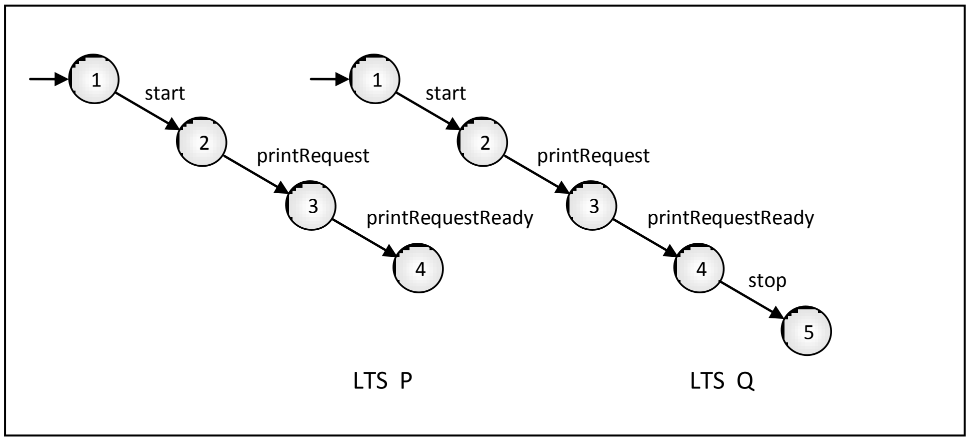


Figure 11: LTS P and LTS Q

LTS P is trace included in LTS Q since all traces of LTS P are traces of LTS Q. The traces of LTS P are traces(P): {init; init, printRequest; init, printRequest, printRequestReady} and the traces of LTS Q are traces(Q): {init; init, printRequest; init, printRequest, printRequestReady; init, printRequest, printRequestReady, stop}.

LTS Q is not trace included by LTS P since the trace: init, printRequest, printRequestReady, stop, is a trace in Q but not in P.

The relation used in this thesis is the trace membership relation. This is a simplification of the trace inclusion relation, since it describes the 'inclusion' of one trace into an LTS. This relation is defined as: the trace σ is an element of the set of traces of LTS Q

Definition
The notation for the trace membership relation is:

$$\sigma \in \text{traces}(Q)$$

Where σ is a trace and Q an LTS.

The Trace inclusion relation and the trace membership relation are similar when LTS P consists of only one trace (like σ is one trace).

If a composition of two LTSs, made by synchronizing on shared labels, contains a trace, then each of the individual LTSs contains a trace. The other way around also holds: if two LTSs both contain a trace, the composition of these two LTSs also contains a trace, when it is synchronized on shared labels.

In a formal notation:

$$\sigma \in traces(P \mid\mid_L Q) \Leftrightarrow (\sigma | L_p \in traces(P)) \cap (\sigma | L_q \in traces(Q))$$

Where the label set of the composition is the union of the label sets of both LTSs:

$$L = L_p \cup L_q$$

Where L is the label set of the composition, L_p is the label set of LTS P and L_q is the label set of LTS Q. The notation $\sigma|L$ means: σ for which holds L, in this case it means that σ is a trace only containing labels from the label set L.

2.2 LOTOS

The representation of a printer specification in an LTS, like Figure 7 or Figure 8, is not directly suitable to describe the correct behaviour of the requirements of a printer. There are two reasons for that:

- a composed model of a printer can easily have billions of states, drawing them is cumbersome;
- transitions can have corresponding data values, expressions and constraints, which cannot easily be modeled in an ordinary LTS.

To overcome these issues, another way of representing a transition system is needed. In this thesis the specification language LOTOS (Language of Temporal Ordering Specification), a process algebraic language, is used [5].

LOTOS has been developed for the formal description of the Open system Interconnection (OSI) architecture within the International Organization for Standardization (ISO), although it is applicable to distributed, concurrent systems in general.

In LOTOS a system is seen as a set of processes which interact and exchange data with each other and with their environment. The language consists of complementary formalisms for data and control. The control part, basic LOTOS, a CCS/CSP- based language, is a subset of the language where process synchronization is achieved, but without data exchange [6]. The data structures of LOTOS are derived from the specification language for abstract data types ACT ONE [7, 16]. Only data types (called sorts in LOTOS) and value expressions of ACT ONE are used and described in this thesis.

2.2.1 Basic LOTOS

In basic LOTOS, behavior is described by behaviour expressions [13]. The syntax for a behavior expression *B*, is the following:

$$B \ =_{def} \ a\,;B \mid \boldsymbol{i}\,;B \mid \Sigma\mathcal{B} \mid B|[G]|B \mid P \mid stop$$

These constructs have the following meaning:

- The action prefix expression *a; B*, with $a \in L$, the set of labels of the system, describes the action *a* (comparable with a transition in an LTS) and then behaves as *B*. The semantics for this axiom, usually formally defined by means of axioms and inference rules, is:

$$\vdash \; a\,;B \xrightarrow{a} B$$

  This axiom is to be read as: an expression of the form *a ; B* can always make a transition $\xrightarrow{a}$ to a state from where it behaves as *B.*

- The expression **i** ; *B* is analogous to *a* ; *B*, the difference being that **i** denotes an internal action τ in the transition system:

$$\vdash \; \mathbf{i}\,;B \xrightarrow{\tau} B$$

- The choice expression $\Sigma\,\mathcal{B}$, where $\mathcal{B}$ is a countable set of behaviour expressions, denotes a choice of behaviour. It behaves as any of the processes in the set $\mathcal{B}$. It is formally defined by the inference rule:

$$B \xrightarrow{\mu} B',\; B \in \mathcal{B},\; \mu \in L \cup \{\tau\} \qquad \vdash \quad \Sigma\,\mathcal{B} \xrightarrow{\mu} B'$$

  This inference rule is to be read as follows: suppose that we know that *B* can make a transition to *B*′; moreover we have that $B \in \mathcal{B}$ and μ is any observable or internal action, then we can conclude that $\Sigma\,\mathcal{B}$ can make the same transition to *B*′.

  B_1 [] B_2 is used as an abbreviation of $\Sigma\,\{B_1, B_2\}$, i.e., B_1 [] B_2 behaves as either B_1 or B_2. The expression *stop* is an abbreviation for $\Sigma\,\emptyset$, i.e., it is the behaviour which cannot perform any action, so it is the deadlocked process.

- The parallel expression B_1 |[*G*]| B_2, where $G \subseteq L$, denotes the parallel execution of B_1 and B_2. In this parallel execution all actions in *G* must synchronize, while all actions not in *G* (including τ) can occur independently in both processes, i.e., interleaved. || is used as an abbreviation for |[*L*]|, i.e., synchronization on all actions except τ, and ||| as an abbreviation for |[$\emptyset$]|, i.e., full interleaving and no synchronization. The interference rules are as follows:

$$B_1 \xrightarrow{\mu} B'_1, \qquad \mu \in (L \cup \{\tau\})\backslash G \quad \vdash \quad B_1 \,|[\,G\,]|\, B_2 \xrightarrow{\mu} B'_1 \,|[\,G\,]|\, B_2$$

$$B_2 \xrightarrow{\mu} B'_2, \qquad \mu \in (L \cup \{\tau\})\backslash G \quad \vdash \quad B_1 \,|[\,G\,]|\, B_2 \xrightarrow{\mu} B_1 \,|[\,G\,]|\, B'_2$$

$$B_1 \xrightarrow{a} B'_1, B_2 \xrightarrow{a} B'_2, \qquad a \in G \quad \vdash \quad B_1 \,|[\,G\,]|\, B_2 \xrightarrow{a} B'_1 \,|[\,G\,]|\, B'_2$$

- A process definition, *P*, links a process name to a behaviour expression:

$$P := B_p$$

The name P can be used in behaviour expressions to stand for the behaviour expressed by its corresponding behaviour expression. Formally:

$$B_p \xrightarrow{\mu} B', \quad P := B_p, \quad \mu \in L \cup \{\tau\} \quad \vdash \quad P \xrightarrow{\mu} B'$$

- The expression *stop* denotes a valid end expression.

As usual parentheses are used to disambiguate expressions. If no parentheses are used ';' binds stronger than '[]', which binds stronger than '|[*G*]|'. The parallel operators read from left to right; they are not associative for different synchronization sets.

Example
To illustrate this syntax, the examples from Figure 7 and Figure 8 are written in LOTOS (see figure 12 and figure 13).

```
S0 :=  init;
       warmingUp;
       standby;
       S3
S3 :=  start;
       run;
       stopped;
       standby;
       S3
```

Figure 12: LOTOS specification of Status protocol

```
P0 :=  start;
       printRequest;
       printRequestReady;
       stopped;
       stop
```

Figure 13: LOTOS specification Print protocol

2.2.2 Full LOTOS

Full LOTOS, or LOTOS, has the advantage over (basic) LOTOS that it has the ability to model with data types.

In full LOTOS, the semantics of parallel composition is unchanged with respect to basic LOTOS. Interprocess communication may still occur when two processes composed in parallel are offering the same action (a transition in an LTS). An action in full LOTOS, which can exchange data values, is formed of three components: a gate, comparable with a label in an LTS; a list of events; and an optional predicate [8].

Processes synchronize their actions, provided that they name the same gate, that the lists of events are matched, and that the predicates, if present, are satisfied. An event can either offer (!) or accept (?) a value. The synchronization rule of basic LOTOS is replaced by synchronization rules in full LOTOS. In full LOTOS there are three kinds of synchronization:

- Value matching:

  $$B_1 \xrightarrow{a!E1} B'_1, B_2 \xrightarrow{a!E2} B'_2, \quad a \in G \quad \vdash \quad B_1 \,|[\, G \,]|\, B_2 \xrightarrow{a\,[E1=E2]} B'_1 \,|[\, G \,]|\, B'_2$$

  E1 and E2 are expressions and must belong to the same data type. It will succeed if E1 equals E2 from the specification of the common type.

- Value passing:

  $$B_1 \xrightarrow{a!E} B'_1, B_2 \xrightarrow{a?x:S} B'_2, \; a \in G \quad \vdash \quad B_1 \,|[\, G \,]|\, B_2 \xrightarrow{a\,!x=E} B'_1 \,|[\, G \,]|\, B'_2\,[E/x]$$

  Expression E must belong to the data type S. It will succeed, replacing x by E in B_2.

- Negotiation:

  $$B_1 \xrightarrow{a?x:S} B'_1, B_2 \xrightarrow{a?y:S} B'_2, \; a \in G \quad \vdash \quad B_1 \,|[\, G \,]|\, B_2 \xrightarrow{a\,!x=y} B'_1 \,|[\, G \,]|\, B'_2\,[x/y]$$

  It will succeed, becoming *x* = *y* = *v,* where *v* is some value in the specified data type *S*

When a predicate is used, e.g., a[E1 = E2], synchronization can only take place if the result of the predicate evaluates to true, i.e., E1 equals E2.

Example
When data types and data are added to a specification, the actions of the specification contain besides the action name, the variable name and the variable type (see figure 14).

```
P0 :=   start;
        printRequest ? id:Nat;
        printRequestReady ! id;
        stopped;
        stop
```

Figure 14: Specification with LOTOS data type instances

Figure 14 specifies the Print protocol of Figure 8, extended with a variable of the data type Natural. The transition *printRequest* needs a Natural value before it can synchronize, another process has to pass this value. When this condition is met, the transition *printRequestReady* matches the obtained value with the next transition of the synchronizing process. An example of a process that can synchronize with this process is given in Figure 15.

```
P1 :=   start;
        printRequest ! 23;
        printRequestReady ! 23;
        stopped;
        stop
```

Figure 15: Arbitrary LOTOS process

Figure 15 depicts an arbitrary process (*P1*) that can synchronize with process *P0* of Figure 13 (*P0* || *P1*). After the transition *printRequest* the value of *id* is 23. This value matches in the transition *printRequestReady*, so synchronization can be obtained.

An alternative for the construction of *value passing* and *value matching,* as depicted in Figure 14, is a construction with *value passing* and a *constraint* (see figure 16).

```
P0 :=   start;
        printRequest ? id1:Nat;
        printRequestReady ? id2:Nat [id1 = id2];
        stopped;
        stop
```

Figure 16: Alternative specification with value passing and constraint

The process in this figure also synchronizes with process *P1* of Figure 15 (*P1* || *P0*). The action *printRequest* again synchronizes passing the Natural 23 to the variable *id1*. The action *printRequestReady* of *P0* tries to synchronize with *printRequestReady* of *P1,* first passing the Natural value 23 to *id2.* The synchronization is successful when the value of *id1* is equal to that of *id2*, like the constraint.

2.2.3 LOTOS example

Besides the description of the behavior of a LOTOS specification, the total specification is enclosed in specification keywords following the LOTOS syntax [5,6,7,8] (see figure 17).

```
SPECIFICATION Print [start, printRequest, printRequestReady, stopped]
:NOEXIT

LIBRARY
        NATURAL
ENDLIB

BEHAVIOUR
        Protocol [start, printRequest, printRequestReady, stopped]
        :NOEXIT

        WHERE

        PROCESS Protocol [start, printrequest, printrequestready, stopped]
        :NOEXIT
                start;
                printRequest ? id : NAT;
                printRequestReady ! id;
                stopped

        ENDPROC

ENDSPEC
```

Figure 17: LOTOS specification

A specification starts with *SPECIFICATION* and ends with *ENDSPEC*. In between these keywords one or more processes, libraries and transitions can be defined. Each process starts with the keyword *PROCESS* and ends with *ENDPROC*, each library instantiation with *LIBRARY* and *ENDLIB*. EXIT or NOEXIT defines respectively whether the process can terminate successfully or not. Below the statement WHERE an earlier used process is defined.

Declarations of types have to be placed before the use of a type instance. Processes and transitions can be placed in parallel or after each other. Libraries are at compile time pasted in the specification at the place of the library declaration. Hence a library file contains normal LOTOS code with LOTOS syntax.

The ability to simplify means to eliminate the unnecessary so that the necessary may speak

(Hans Hofmann)

3 Method

The verification and analysis of the printer by means of its logfiles requires three ingredients:

- a logfile, which has to be verified and analyzed;
- a specification, which defines the correct behaviour of the printer;
- a relation between the logfile and the specification.

This chapter describes these three ingredients in the developed method

Organization of this chapter
Section 3.1 describes the log model. Section 3.2 describes the specification, a formalization of the requirements of the printer, in different models: reference, synchronization and test-specific models. Section 3.3 describes the verification and analysis of a logfile. Section 3.4 describes an alternative.

3.1 Log model

A logfile contains all actions and details of actions of logged protocols, for example: the time on which the action is performed, the name of the sending or receiving process, the action label and an action identification number. To deal with this information the logfile is transformed into a log model. This transformation is done into two parts: filtering and formalization. Filtering keeps only actions of the protocols to verify and formalization puts these actions in the correct syntax. The transformation of the logfile has three reasons:

- not all logged protocols are formalized in the specification, the formalization of a subset of the requirements only makes sense when also a subset of the logfile is taken;
- not all details in the logfile are used in the specification. The actions in the logfile contain a large amount of details, used for debugging, the actions in the specification do not. These details are not used for the verification and analysis;

- not all statements in the logfile have the same syntax. Since the logfile is originally used for debugging, the logged actions do not always have a standardized structure. The formalization covers the inconsistent formats of the notation of logged actions.

When coding rules are applied strictly and logged actions do have the same syntax, the transformation can be brought back to filtering.

The resulting formalized logfile is called log model. This model contains only actions of protocols modeled in the specification and has a well-defined structure. The structure of the log model satisfies the Sequence format rules [17]. A log model consists of a sequence of transition labels, each possible followed by one or more identification numbers or other naturals (e.g., time).

Since the syntax of the logged actions differs per protocol, the transformation is different for each protocol. If the syntax of a protocol is known, the transformation is done automatically.

Example
Figure 10 (page 27) depicts a log model in which the printer printed one sheet. This model can be represented in a sequence (see figure 18).

```
init;
warmingUp;
standBy;
start;
printRequest !3;
run;
printRequestReady !3;
stop;
standBy;
```

Figure 18: Log model in sequence format

3.2 Specification

To verify and analyze the logfile of a printer, a notion of a correct logfile is needed. The logfile can be verified against the requirements of the printer, which describes the printer behaviour informally, but this description often is not unambiguous. To overcome this problem the requirements of the printer are formalized, which gives a formal specification.

A disadvantage of the formalization of the requirements is their large size. A printer is a complex machine, which has many complex requirements. This means that a simple verification must be preceded by the huge task of formalization. To overcome this barrier AVATR is developed to be able to use a subset of the formalized requirements.

When formalizing the logfile, the actions modeled in the specification must be known in order to verify and analyze the printer on those actions.

The sections below describe the formal specification, which consist of three types of models: reference, synchronization and test-specific.

3.2.1 Reference model

A reference model is a formalization of a small subset of the requirements of the printer. This can be a coherent part, like all the actions of one protocol, but this can also be an arbitrary part of the actions of a process. However, a small coherent part is easier to maintain and more generally applicable than a large model. This is especially true for printers under development, since they might change.

A typical coherent subset of a printer is a protocol. The requirements of a particular protocol, given in English text, consist of a description of actions and responses, illustrated with sequence diagrams.

Protocol actions and responses are modeled in one model without distinction between them, since they belong to the same protocol. By modeling them in the same model, the order of action and response is defined.

Example
From the requirements of the Status protocol a reference model is extracted, which contains all protocol actions and responses in all allowed orders (see figure 19).

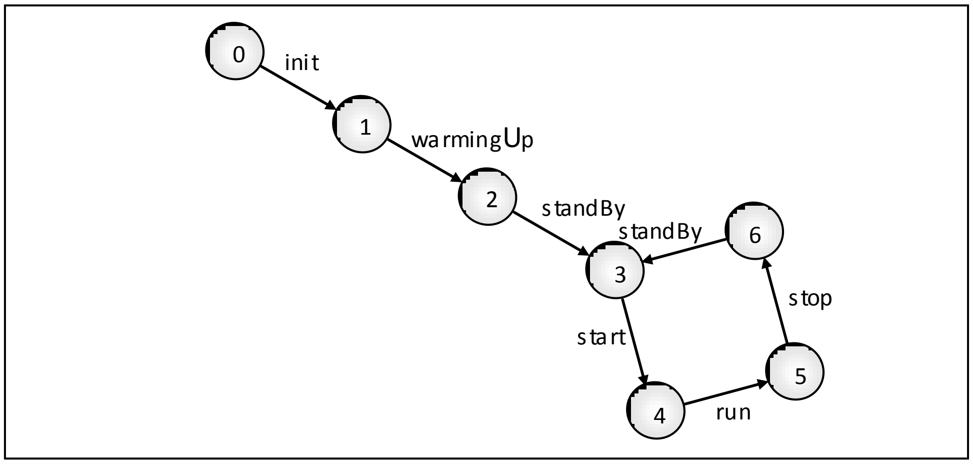


Figure 19: Reference model of Status protocol

At the same manner a reference model of the print protocol is drawn (see figure 20, slightly different from the model given in figure 8 on page 27 because the start and stop actions do not belong to this protocol).

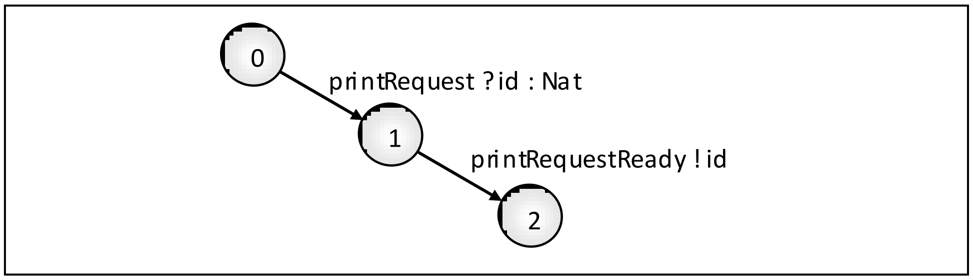


Figure 20: Reference model of Print protocol

In this reference model, each transition has an integer variable to store an identification number. This makes it possible to distinguish transitions within several protocol instances of this protocol.

3.2.2 Synchronization model

A reference model describes a small subset of the printer functionality. To verify a larger part of the printer there are two options:

- model a larger part of the functionality in one reference model;
- create a composition of small reference models.

A model of a larger part of the functionality of the printer shatters the maintainability of the reference models, since there are no restrictions to which extent a model can be enlarged, the overview is easily lost.

A composition of more small reference models preserves maintainability of the approach but loses the obligatory order between the transitions of the different reference models. A full interleaving of small reference models is not by definition correct: some process actions are not allowed in arbitrary order. To overcome this problem, another type of models is defined: a synchronization model.

A synchronization model defines the allowed order of transitions of the different reference models; it does not define new actions. It contains a subset of actions of two or more reference models, with the obligatory order between them. In this model only actions of the related reference models that have order constraints are modeled, i.e., actions which always happen before or after another action. Actions without constraints do not need to be modeled, since they are already full interleaved.

A synchronization model provides flexibility to the specification. Due to this model small general reference models can be created and combined in every way. The composition of a synchronization model and a reference model is given by the parallel LOTOS expression (section 2.2):

reference model$_1$ |[G_1]| *synchronization model* |[G_2]| *reference model*$_2$

or, since ||| and |[]| are associative and $G_1 \cap G_2 = \emptyset$, by:

reference model$_1$ ||| *reference model*$_2$ |[$G_1 \cup G_2$]| *synchronization model*

Where G_1 and G_2 are label sets with order constraints, which contain respectively labels of *reference model*$_1$ and of *reference model*$_2$.

When a synchronization model is used, all the transitions in it must be placed in one of the label sets G_1 or G_2. When the sets G_1 and G_2 are empty, full interleaving is achieved, as if there is no synchronization model. If these sets are non empty, partial interleaving is achieved between the reference model and the synchronization model. If two reference models are completely independent of each other, no synchronization model is needed.

Example

The actions of the Status protocol, Figure 19, and the Print protocol, Figure 20, have an obligatory order. The actions in the Print protocol are only allowed after the Status protocol action *standby*. This obligatory order is drawn in a synchronization model (see figure 21).

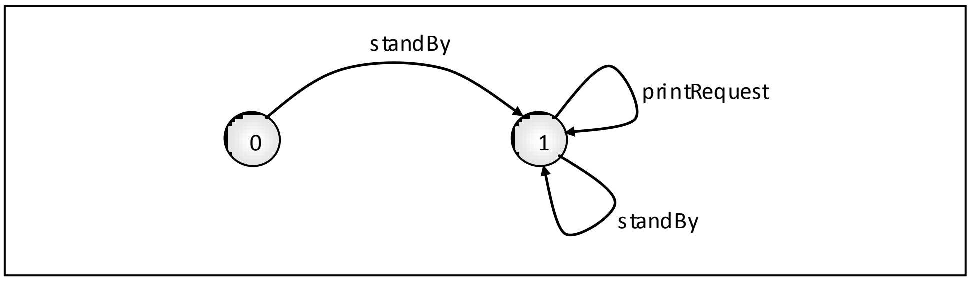


Figure 21: Synchronization model for Status and Print protocol

This figure shows a model that defines the order of the actions *standBy* and *printRequest*. *printRequest* is only possible after a first occurrence of *standBy,* after which the action *standBy* is still allowed.

The models of the Status protocol and the Print protocol and the synchronization model (Figure 21) are composed together to form a total specification (see figure 22) with the LOTOS expression:

Status |[*standBy*]| *synchronization* |[*printRequest*]| *Print*

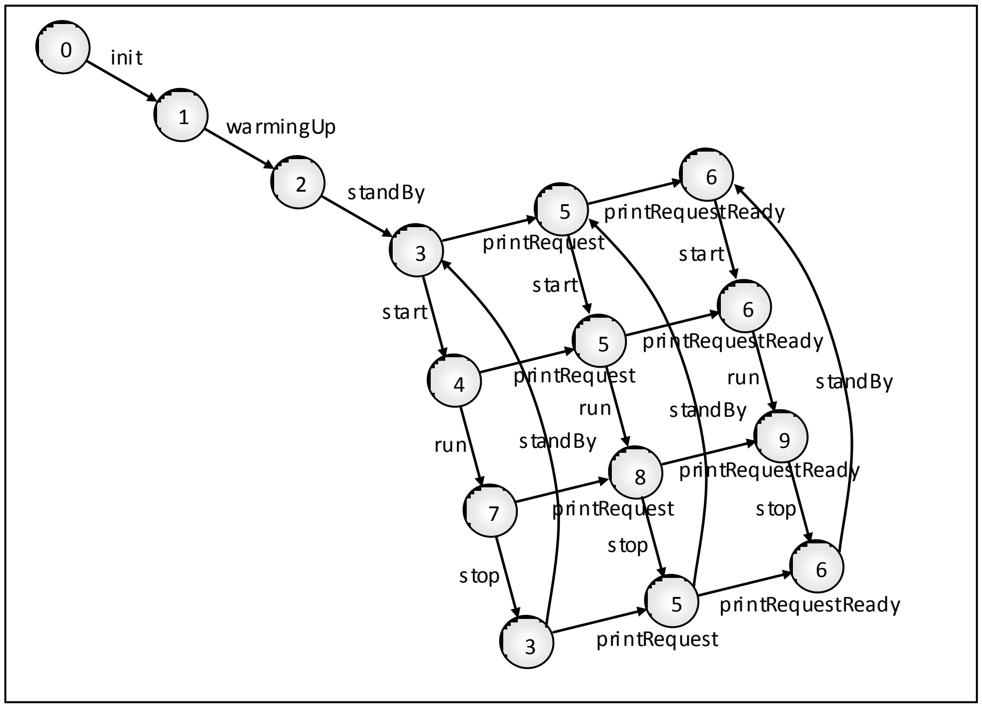


Figure 22: Specification, containing reference and synchronization models

3.2.3 Test-specific model

Reference and synchronization models are sufficient for test-independent verification and analysis, the verification and analysis of test-specific behaviour needs more information, e.g., information about the specific test which was performed while the printer produced the logfile. This extra information is modeled in a test-specific model.

A test-specific model contains actions of protocols on which the test-specific constraints apply. These test-specific constraints are a specific situation in the reference models that describe its specification more closely, e.g., a reference model specifies which actions are needed to print a page, a test-specific model specifies the number of pages in the specification.

The composition of a test-specific model with the reference and synchronization models is similar to that of a synchronization model with a reference model (section 3.2.2).

Example
The specification of Figure 22, describes a printer which prints one page: there is only one action *printRequest* in one trace possible. To verify a print job of more pages, more *Print models* have to be composed. This is for sake of a clear overview not drawn in an LTS. Test specific situations could be the number of printed pages before the second visit of the state *standby*, or the total amount of printed pages.

To verify that the printer prints only one page, and to exclude the printing of no pages or a second page, a test-specific model is created (see figure 23).

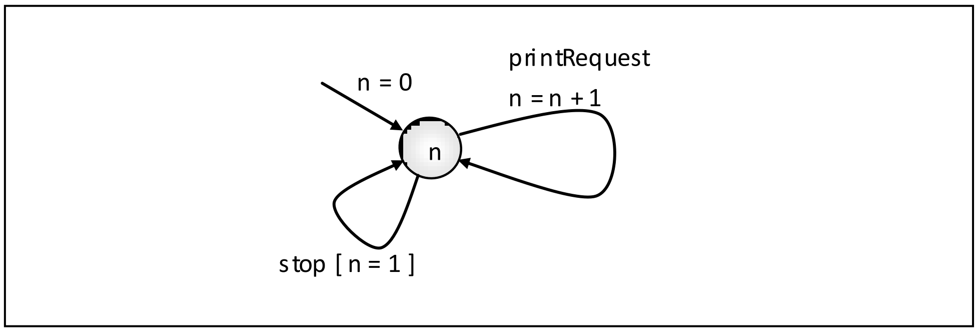


Figure 23: Test-specific model test print of one sheet

This model is synchronized with the LOTOS expression:

specification |[*printRequest, stop*]| *test-specific model*

Every time an action *printRequest* is done, the counter *n* increments by one, every time the action *stop* is done, the counter *n* is checked to be one. When the counter exceeds 1, the *stop* action can not be performed since n is not equal to one and the verification fails. The number of pages can easily be adjusted.

Another example of a test-specific model deals with time. If a printRequest is done, it can be interesting to verify that the corresponding printRequestReady is performed within a given amount of time, for example 100 micro seconds. This can also be specified in a test-specific model (see figure 24).

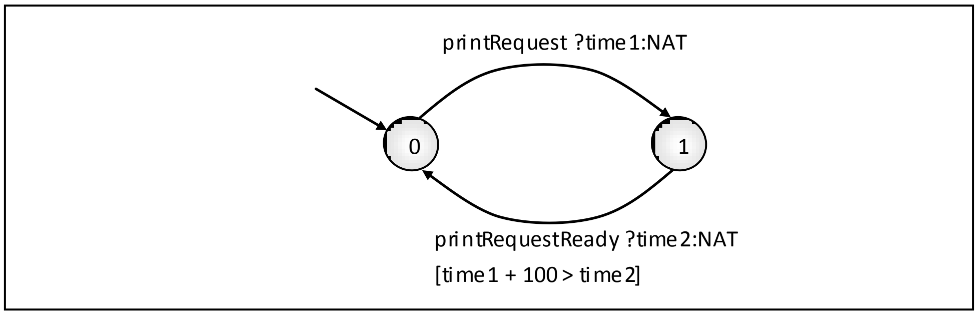


Figure 24: Test-specific model test specific time

3.3 Correctness relation

The log model represents in one sequence the behaviour of the printer. The specification, consisting of reference, synchronization and test-specific models, shows all possible correct sequences of the printer. The relation between this log model and specification determines whether the log model is correct or not (section 1.1.3). This relation is a trace membership relation; a log model is a member of the traces formed by the specification, if the printer behaves correctly (section 2.1.2). The verification of the printer, by means of a logfile, is the check for existence of this relation. The analysis of the printer, when this relation does not exists, determines the first failure.

The specification of the printer is deterministic (i.e., it does not contain states with outgoing transitions with an equal pair: action name, identification number, Section 1.2.2). Hence, the search of the trace inclusion relation between the log model and the specification is a deterministic process, which does not require any search algorithms; there is only one way to match every transition given the previous transitions.

The simplest algorithm to verify the log model with the specification starts with the first log model transition, if this transition matches one of the outgoing transitions from the initial state of the specification; this transition is correct and the new state of the specification is remembered. Subsequently, the next log transition is matched to an outgoing transition of the new specification state. This is repeated until the last log model transition has been reached or until a log model transition does not match with a specification transition. In the latter case, the part of the log model checked so far is printed, which leads to the unexpected transition.

To verify whether the last log model transition corresponds with a correct end transition of the specification, a final transition is appended at the end of the log model and after each correct end transition in the specification. This final transition must have a label different from all other used transition labels in the specification or log model, to distinguish between them. In a model more final transitions can be defined, which are treated the same as other transitions in the model. If more protocol instances are used, more final transitions can be appended. However, for every logfile only one final transition is needed in the specification, since a log model is a sequence of transitions which has only one last transition.

3.4 Verification

Specification models are composed together to form a larger specification (section 3.2.2). Since independent sub-models of the specification are composed in parallel, the composition can describe a very large transition system.

To handle large specifications efficiently, two verification methods are applied:

- the generation of the composition of the specification models on-the-fly with the verification;
- the verification of each specification sub-model separately.

An alternative to these verification options is the composition of the specification models with the log model, resulting in one model which includes the log model.

The sections below describe the on-the-fly composition, the separate verification of sub-models, and the composition of the specification models with the log model.

3.4.1 On-the-fly composition of specification.

The log model consists of one sequence of actions of the printer and the specification consists of all correct sequences (Section 1.1.3). Each log model transition can be matched on one specification transition, starting with the first. Since the specification is deterministic, only one specification transition can match with a log model transition. For each log model transition only one specification transition is relevant and other transitions do not match. These other transitions do not have to be created in the transition system. Hence the on-the-fly created specification transition system is smaller than the original composition of the specification models.

3.4.2 Separate verification of specification models

Besides the on-the-fly verification, the printer functionality can be verified and analyzed separately, by means of separate models. This holds for each model, whether it is a reference, synchronization or test-specific model.

A reference model can be verified stand alone, since it describes only one (stand alone) protocol. For this verification only the transitions used in the specification are formalized in the log model.

The verification of synchronization and test-specific models is done the same way. Only the transitions used in the particular specification models are formalized in a log model, and only the functionality described in the particular specification model is verified.

When a log model is trace included in a composed specification, the individual parts of this specification include traces of 'sub-log' models. These sub-log models contain only the functionality (actions) of the particular model to verify (see section 2.1.2). This is the same as the verification of only the Print protocol or only the Data protocol.

3.4.3 Composition of specification with log model

An alternative for the previous described methods is the creation of a composition of the log model with the specification models and check whether the last transition of this composition is reached.

This method creates a composition of the specification models together with the log model. Since the log model is included in the specification, if the printer behaviour is correct, the synchronization of the log model with the specification is similar to the log model itself. Hence, if the final log transition is reached in this composition, the log model describes a correct path through the printer specification.

The verification consists, besides the synchronization of the log model with the specification, of a check whether the last log model transition can be reached in the composition or not. To be sure the resulting system is in a correct end state, final transitions are added to the logfile and the specification.

If I had eight hours to chop down a tree, I'd spend six hours sharpening my axe

(Abraham Lincoln)

4 Implementation

The AVATR method, described in Chapter 3, is implemented. The described models are implemented in the formal specification language LOTOS and the trace inclusion relation is implemented in the AVATR tool chain.

Organization of this chapter
Section 4.1 describes the motivation for the specification language LOTOS. Section 4.2 describes the individual parts of the tool chain, a LOTOS editor, LOTOS compilers, a preprocessor and a Testlog verifier. Section 4.3 describes two alternative specification languages: μCRL and a general programming language, and five alternative Testlog verifier implementations: TETRA, UPPAAL, TorX, CADP Bisismulator and the μCRL toolset.

4.1 Specification language

AVATR models are described in the formal description language LOTOS [5, 6, 7, 8], which has been introduced in Chapter 2.

Requirements for a language are:

- the language has to be well-documented;
- the language must be able to implement transitions;
- the language must be able to use instances of the data type *Natural* to express identification numbers and time.

Tool support is an advantage.

The language LOTOS satisfies these requirements. It is a well-documented language, since it is a standard of the International Standard Organization. Furthermore it is possible to describe transitions and data types, since the language is designed to specify interaction of processes [5].

Besides these requirements, LOTOS has the following advantages:

- LOTOS supports enforced synchronization, which means that synchronizing transitions can not be taken without synchronizing partner.
- there are tools available which provide the needed trace inclusion functionality.

However, LOTOS has some disadvantages:

- there are no global variables, data type instances have to be passed forward with the call of a process;
- there are no defined data types, only abstract data types can be used. When a data type is needed it has to be defined, which means that each data type instance and each operator have to be defined. This is called an abstract data type. The needed natural numbers range from zero to some billions (timestamps are 32 bits).

4.2 The tool chain

The developed AVATR tool chain, which implements the AVATR method, exists of five tools (see figure 25):

- a LOTOS editor;
- a LOTOS functional compiler;
- a LOTOS abstract data type compiler;
- a preprocessor;
- a Testlog verifier.

This tool chain is built as follows: the output of the LOTOS editor is compiled in the two compilers which are standing parallel to each other. The output of these compilers, together with the files produced by the preprocessor and a configuration file are used in the Testlog verifier. The Testlog verifier generates output that is presented to the user of the tool chain.

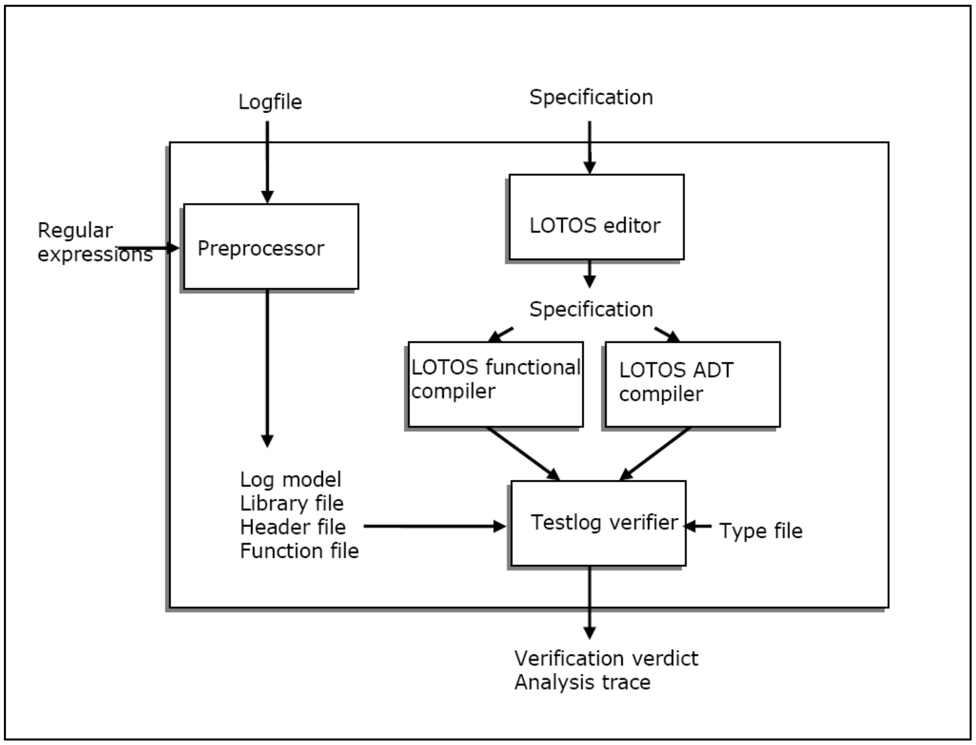


Figure 25: AVATR tool chain

The implementation in a tool chain provides flexibility since individual tools can be changed without changing other parts of the tool chain, provided that the specified interfaces are implemented. Some of these interfaces are specific for the used tools (e.g., preprocessor/ Testlog verifier interface), which makes it hard to replace only one specific tool; instead two tools have to be replaced.

The next sections describe the particular tools in the AVATR chain.

4.2.1 The LOTOS editor

The LOTOS editor can be any text editor, preferably with LOTOS syntax highlighting. In this project the editor VIM (Vi IMproved) [2] is used. VIM is a highly configurable text editor built to enable efficient text editing. It gives the user the possibility to create and edit LOTOS specifications (section 3.2).

Text editors do not give the overview given by graphical editors. However, LOTOS specifications can be converted to LTSs [3]. Since this is not the main issue of this thesis, this is not further described here.

4.2.2 The preprocessor

The preprocessor transforms a logfile into a log model (Section 3.1), and generates files needed by the TestLog verifier:

- a header file; this file is used by CADP Exhibitor, it contains an array with data type instances used.
- a function file; this file is used by the CADP LOTOS compilers, it contains a definition of all used data instances.
- a library file; this file defines the used data type instances in the created specification.

These files are used to specify the abstract data type instances that are used in the verification, which range from 1 to 1024, representing identification numbers (Section 3.2.1), and from 0 to 2^{32}, representing time (Section 3.2.3). The function and library file are mandatory for every verification with the CADP Exhibitor tool since they define the abstract data type instances. The header file and the type file are used to verify and analyze more efficiently when large amounts of data type instances are used. Without these files the verification and analysis will take more time.

The log model, the library file and the function file are respectively created in the sequence format (Section 3.1), the LOTOS format (Chapter 2) and the C format (see Appendix III).

The preprocessor is implemented in the scripting language Perl [18] since this is fast and flexible. It reads the logfile line by line and copies selected printer actions and naturals. Printer actions and naturals are selected by regular expressions, given by an engineer (see Appendix II).

When logfiles are created conform the requirements, the preprocessor can be generated automatically. This is possible since for each protocol the actions are known and the syntax of the actions is uniform. However, in the existing logfiles this syntax is not always the same (Section 3.1), hence it is more flexible to use a scripting language.

Example

An example of a regular expression that transforms the actions of the Status protocol into transitions of the Status model is given in Figure 26.

```
foreach $line (<LOGFILE>) {

$_ = $line;

if (/(\d+):.*(status_(\w*)\)/ )
{
   printLogModel("\"$2 !$1\"\n");
}
```

Figure 26: Preprocessor, Perl script

This expression selects all lines which correspond to a pattern that starts with one or more digits, followed by ':', followed by zero or more characters or digits, followed by the string *status_*, followed by zero or more characters. Parts of this expression are printed in the log model.

4.2.3 The Testlog verifier

The Testlog verifier performs the actual verification and analysis of the log model in relation to the specification (Section 3.3). It is realized with tools of the CADP (Construction and Analysis of Distributed Processes) toolset [3, 15].

CADP is a software engineering toolbox for the design of communication protocols and distributed systems; based on the formal description language LOTOS. It offers a wide range of functionalities, including compilation, simulation, formal verification, and testing. The toolbox is designed in a modular way and puts the emphasis on intermediate formats and programming interfaces (such as the BCG and OPEN/CAESAR software environments), which allow the CADP tools to be combined with other tools and adapted to various specification languages (e.g., EXP, SVL, μCRL).

The CADP tools used in this thesis are:

- Casesar, a compiler that translates LOTOS functional behaviour into C code;
- Caesar.adt, a compiler that translates LOTOS abstract data types into C code;
- Exhibitor, a verification and analysis tool that checks if a sequence file is contained in a specification. In case the sequence file does not correspond to the specification an analysis trace is produced, which leads to the first incorrect action in the sequence file. Otherwise a verdict true is given.

Exhibitor creates a composition of the specification files on-the-fly, which means that during the verification the composition is made when transitions are needed in the verification. This prevents the creation of large composition files, which are partly unused because the log model only needs one trace in the specification model.

Besides the log model and the specification, the CADP Exhibitor tool uses (figure 25):

- a library file (.lib);
- a header file (.h);
- a function file (.f);
- a type file (.t).

The library, header and function file are generated by the preprocessor, since they are dependent of the logfile, see appendix III for an example of these files. The type file is part of the Exhibitor tool; it defines the specified naturals to use. This file is once created and does not have to be modified (see appendix II B).

The verification of a specification, which uses abstract data type instances in combination with value acceptance (e.g., value passing or negotiation, section 2.3.2), enlarges the state space exponentially. Exhibitor performs every outgoing specification transition for every data type instance, defined in the type file. When a matching data type instance is offered (e.g., value matching or value passing, Section 2.3.2) in the sequence file, one of the created outgoing transitions is chosen.

4.3 Alternatives

Alternatives for the formal description language LOTOS are: μCRL or more general languages. Section 4.3.1 describes and evaluates these alternatives.

Alternatives for the CADP Exhibitor tool are: TETRA, TorX, UPPAAL, CADP Bisimulator, and the μCRL toolset. These tools are able to formally support verification and analysis by means of formal descriptions. Section 4.3.2 describes and evaluates these alternatives.

4.3.1 Language

Formal specifications can be given in several languages, even in informal languages. However, formal languages support often formal techniques for verification or simulation and informal languages do not. Below, μCRL is described as alternative for LOTOS, and a general overview of the advantages and disadvantages of general languages are given.

μCRL
μCRL is, like LOTOS, a well-documented language, used to model protocols. It provides possibilities to define abstract data types. μCRL has been extended with features to express time, which is, however, not supported by tools [29].

μCRL does not seem to be significantly different or better than LOTOS; it provides the same functions and uses the CADP trace inclusion tool.

General language
A second alternative is the creation of a new language or the use of a more general language (e.g., XML, Java). These languages can implement transitions and data types but implement a lot more functionality, for example: other data types and conditional expressions. This functionality is not needed and could easily be an obstacle.

Another disadvantage is that there is no tool support for specific functionality (e.g., the trace inclusion relation).

4.3.2 Tool

The verification of the trace inclusion relation can be implemented in several tools, described below.

TETRA
TETRA [19, 20] is a tool which compares observed test results with a reference specification. [19] presents TETRA as an operational test trace analysis system which provides diagnostics in case of non-conformance with the specification. This tool requires the specification and the observed execution trace written in the specification language LOTOS. [20] describes some experiments, obtained on a Sun 4/330 with 32Mb of RAM. The conclusion of this paper is, that it is

possible to handle real-life protocol specifications which cover several thousand lines of LOTOS code.

TETRA could be useful in this project, as it contains the needed functionality to validate test results with a specification. However, since the tool is created in 1989, it is designed for different computers than used at the present time (at Océ-Technologies). Furthermore, it was not possible to obtain a copy of the tool.

TorX

TorX [21, 22, 23] is a tool for specification-based black-box conformance testing. [23] describes the flexible and open architecture. Flexibility is obtained by requiring a modular architecture with well-defined interfaces between the components, this allows easy replacement of components. Openness is acquired by choosing existing interfaces to link the components, this enables integration of ‘third party’ components. TorX provides automatic test generation, test implementation, test execution and test analysis. It does the testing in an on-the-fly manner; each test step is derived on demand when the test execution needs it. [21] and [23] describe the architecture of TorX; it explores given LOTOS specifications with the CADP tool: Caesar. [22] shows tool dependencies which have to be installed to use TorX: Perl, TCL and Expect 5.27.

TORX is an option due to its flexibility and openness. The defined interfaces allow other components to interface with the tool. However, the tool has program dependencies; it uses among others CADP for its trace inclusion functionality. Less tools mean less possible failures and hence this option is abandoned.

UPPAAL

UPPAAL [24, 25, 26] is an integrated tool environment for modeling, simulation and verification of real-time systems. It is appropriate for systems that can be modeled as a collection of non-deterministic processes with finite control structure and real-valued clocks, which communicate through channels or shared variables.

[25] shows the three main parts of UPPAAL:

- a graphical interface that supports graphical and textual representations of networks of timed automata, and automatic transformation from graphical representations to textual format;
- a compiler that transforms a certain class of linear hybrid systems to networks of timed automata;
- a model checker that checks invariant and reachability properties by exploring the state-space of a system, i.e. reachability analysis in terms of symbolic states represented by constraints.

[26] shows the support of diagnostic model checking providing diagnostic information in case verification of a particular real time system fails. [25] shows that, besides the graphical user interface, a subset of XML can be used.

An advantage of UPPAAL is that it is a tool with a graphical interface. It has also the possibility to use a textual representation, using XML. The AVATR method is not possible in this tool, since no inclusion functionality is present;

hence this tool is not used. However, an alternative method, the parallel compositions of the specification with the log model (section 3.4.3), can be applied in this tool.

CADP Bisimulator
The CADP Bisimulator [3, 27] is a tool which performs an on-the-fly comparison of two LTSs modulo a given equivalence/preorder relation. The result of this verification (TRUE or FALSE) is displayed on the standard output, possibly accompanied by a diagnostic. [27] describes the verification method of Bisimulator, which is based upon a translation of the equivalence/preorder checking problem into the resolution of a Boolean Equation System, which is performed on-the-fly.

Bisimulator is less efficient than Exhibitor, since it uses a more general verification algorithm. Implementations in Bisimulator showed that this option is twice as slow as the Exhibitor.

μCRL toolset
The μCRL toolset [28, 29] is constructed around a restricted form of the μCRL language [30], namely the linear process operator format (lpo). The tool μCRL checks whether a certain specifcation is well formed μCRL and attempts to transform it into a linearised form, which can be used by other tools. [29] tells that a major idea of the μCRL toolset is that the toolset must only provide functionality that cannot be easily obtained via the use of other tools. The trace inclusion relation, implemented in the CADP toolset, is one of the not implemented functionalities, since the CADP toolset can be used.

The μCRL toolset does not implement the used trace inclusion relation, hence this option is abandoned.

An undetected error [...] is like a sunken rock at sea yet undiscovered, upon which it is impossible to say what wrecks may have taken place

(Sir John Herschel)

5 Feasibility study

The developed AVATR method, implemented in the AVATR tool chain is used to verify and analyze the behaviour of a printer by means of a logfile, which is obtained by the test of the printer.

This chapter describes four cases that represent all functionality to describe printer protocols, relations between printer protocols, and test specific situations. With the techniques used in these cases other printer functionality can be modeled, verified and analyzed. This chapter gives a proof-of-concept of the developed method and tool chain.

Organization of this chapter

Section 5.1 describes the verification and analysis of three protocols: the Status protocol, the Print protocol and the Data protocol. Section 5.2 describes relations between protocols; between the Status protocol and the Print protocol. Subsequently, Section 5.3 describes test-specific situations: the number of printed pages and the time needed to print a page. Section 5.4 describes the verification of a compositions of reference and synchronization models, in order to create a larger specification.

5.1 Protocols

This section describes the verification and analysis of separate printer protocols, the Status, Print and Data protocol.

These protocols represent all protocols in the printer. When an arbitrary printer protocol can be verified and analyzed, the main functionality of the printer can be verified and analyzed, since the protocols determine a major part of the printer functionality.

5.1.1 Status protocol

The Status protocol exists between the Controller and the Engine (Section 1.2.1). It is one of the most general protocols that is used in various printer types. It provides separation of concerns, i.e., behavior for a specific state or transition of the Engine can be dealt with separately. The Engine can stay in a defined number of states, between which switching is controlled and observed. The role of the Status protocol is to facilitate changing and observing these states.

The Status protocol is the most basic protocol of the printer. It consists of simple actions without identification numbers. Furthermore, there is only one instance of this protocol in the entire printer.

The aim of the verification and analysis of this protocol is to prove the method and tool chain to work in the most basic form; verification and analysis of logfiles with a small simple reference model.

Input
The tool chain input exists of three parts: a logfile, the Status reference model (see appendix I a) and a Perl regular expression (see appendix II a).

Results
Results of the verification and analysis of the functionality of the Status protocol are given in table 1.

Table 1: Results verification and analysis of Status model

	logfile lines	log model transitions	Protocol		Time verification (s)	
			Instances	States/ Inst	Correct	Incorrect**
Log1*	14.942	11	1	12	8	8
Log2*	191.532	35	1	12	9	9
Log3*	706.667	37	1	12	9	9

*the conditions of these verification runs and of these logfiles are found in appendix III

**in these verifications the same logfiles are used except for the last transitions, which are changed into incorrect transitions

Evaluation
The table gives the results of the transformation of the logfile into the log model for only actions of the Status protocol. The table shows that this transformation yields a large reduction in amount of actions/ transitions. Furthermore, the verification times are given, below ten seconds for a correct verification (this includes formalization time for log model) and to find an incorrect transition at the end of the log model also below 10 seconds. The verification of the incorrect log model resulted in an error trace leading to the erroneous transition (section 1.1.3 and appendix III).

5.1.2 Print protocol

The Print protocol is another protocol between the Controller and the Engine. It is responsible for the transfer of print jobs from the Controller to the Engine. To do so, it deals with requests for print jobs from the Controller to the Engine and responses from the Engine to the Controller.

The requests for one print job are:

- one prepare request, to prepare the printer for a new job;
- zero or more print requests, each to request the print of one new page;
- one deliver request, to deliver the set of printed sheets of the last job.

These requests always happen in the above mentioned order; the Controller is not allowed to send requests for two jobs mixed up. Each request from the Controller has a corresponding response from the Engine. Responses do not have a specific order and can happen mixed up, even from several print jobs.

The Print protocol is more complex than the Status protocol, since it contains actions with identification numbers to distinguish different instances of the protocol from each other. In the model of this protocol, identification numbers are modeled as abstract data type instances.

Input
The tool chain input exists of three parts: a logfile, the Print reference model (see appendix I b) and Perl regular expressions (see appendix II a).

Results
Results of the verification and analysis of the functionality of the Print protocol are given in table 2.

Table 2: Results verification and analysis Print model

	Logfile lines	log model transitions	Protocol		Time verification (s)	
			Instances	States/ inst	Correct	Incorrect**
Log1*	14.942	78	12	6	72	75
Log2*	191.532	663	90	6	188	240
Log3*	706.667	107	13	6	71	74

*the conditions of these verification runs and of these logfiles are found in appendix III

**in these verifications the same logfiles are used except for the last transitions, which are changed into incorrect transitions

Evaluation

The Print model, consisting of transition containing labels and identification numbers, can be verified in a reasonable time. The largest verification time of a correct logfile is below 4 minutes for 90 protocol instances, and for incorrect logfiles as well.

5.1.3 Data protocol

A third protocol between the Controller and the Engine is the Data protocol. Since the Print protocol does not download the actual bitmap to print, this is the responsibility of the Data protocol.

In the Data protocol, the Controller sends a download request, which is followed by a request for the bitmap data from the Engine, after which the actual download is performed. The Engine finally sends a download request ready.

The download request has a request id and a bitmap id. The bitmap has an id corresponding with the bitmap id and the request ready has an id corresponding to the request id. Hence several data protocols can be mixed up. When a sheet is printed on two sides (duplex) two instances of the Data protocol are used.

The existence of two id's in Data protocol actions makes this protocol more complex than the Print protocol.

Input
The tool chain input exists of three parts: a logfile, the Data reference model (see appendix I c) and Perl regular expressions (see appendix II a).

Results
Results of the verification and analysis of the functionality of the Data protocol are given in Table 3.

Table 3: Results verification and analysis Data model

	logfile lines	log model transitions	Protocol		Time verification (s)	
			Instances	States/ inst	Correct	Incorrect**
Log1*	14.942	60	20	20	31	35
Log2*	191.532	738	246	246	> 10,000	> 10,000
Log3*	706.667	240	80	80	960	960

*the conditions of these verification runs and of these logfiles are found in appendix III

**in these verifications the same logfiles are used except for the last transitions, which are changed into incorrect transitions

Evaluation

The Data model consists of transition containing labels with two identification numbers. Hence when the log model needs a lot of protocol instances to be verified, the time to do this increases. The largest verification time of a verified logfile is below 16 minutes for 80 protocol instances, more than 3 hours for 246 protocol instances. This is due to the way CADP works with data types (Section 4.2.3).

5.2 Relations between protocols

Besides verification and analysis of standalone protocols, the constraints between different protocols have to be verified in order to verify the whole functionality of the printer.

Constraints between protocols consist of obligatory orders of specific transitions of two protocols (e.g., a specific protocol action always precedes an action of another protocol). These constraints are modeled in synchronization models (Section 3.2.2).

Each synchronization model gives the relation between two reference models. Two reference models and a corresponding synchronization model can be composed together to form a larger specification of the printer.

In this section one synchronization model is described; the Status Print synchronization model.

5.2.1 Status Print synchronization model

An example of a synchronization model is that between the Status model and the Print model (see appendix I d). Print protocol requests are not allowed before the action standby of the Status protocol.

Input
The tool chain input exists of three parts: a logfile, the Status Print synchronization model (see appendix I d) and Perl regular expressions (see appendix II a).

Results
Results of the verification and analysis of the functionality the Status Print synchronization model are given in Table 4.

Table 4: Results verification and analysis Status Print synchronization model

	logfile lines	log model transitions	Model		Time verification (s)	
			Instances	States/ inst	Correct	Incorrect**
Log1*	14.942	15	1	2	13	13
Log2*	191.532	60	1	2	15	15
Log3*	706.667	24	1	2	13	13

*the conditions of these verification runs and of these logfiles are found in appendix III

**in these verifications the same logfiles are used except for the last transitions, which are changed into incorrect transitions

Evaluation
Since the Status Print synchronization model is a very simple model, the verification and analysis time is short. The table shows an equal time for the verification of a correct and incorrect log model.

5.3 Test specific situations

Test-specific situations are not modeled in reference models or synchronization models, since they can change per test output (Section 3.2.3). For test-specific models the synchronization rules apply.

5.3.1 Number of sheets test-specific model

One test-specific model is the model to test the number of printed sheets. This model contains the transition which is given after the print of a sheet, printRequestReady, and a counter which is incremented for every time this transition happens

Input
The input of this verification is the number of sheets test-specific model (see Appendix I F), the logfile and the Perl regular expressions (see appendix II a).

Results
Results of the verification and analysis of the functionality of this test-specific model are given in Table 5.

Table 5: Results verification and analysis Number of sheets test-specific model

	Logfile lines	log model transitions	Model		Time verification	
			Instances	States/ inst	Correct	Incorrect**
Log1*	14.942	12	1	1	11 s	11 s
Log2*	191.532	242	1	1	16 s	16 s
Log3*	706.667	40	1	1	13 s	13 s

*the conditions of these verification runs and of these logfiles are found in appendix III

**in these verifications the same logfiles are used except for the number of prints, this numbers are changed into incorrect numbers

Evaluation

This model does not count with identification numbers or other data type instances. Hence the verification time is approximately 15 seconds. The difference in time is small, since the used specification is in each case the same and only one instance if this model is used. The incorrect verification takes the same amount of time, since the same transitions in the model are taken.

5.3.2 Time test-specific model

Another test-specific model is the model that verifies and analyzes the time between two actions. This test-specific model consists of two transitions between which the time is verified (see appendix I g).

Input
The input of this verification is the number of sheets test-specific model (see Appendix I F), the logfile and the Perl regular expressions (see appendix II a).

Results
Results of the verification and analysis of the functionality this test-specific model are given in Table 6.

Table 6: Results verification and analysis Time test-specific model

	Logfile lines	log model transitions	Model		Time verification (s)	
			Instances	States/ inst	Correct	Incorrect**
Log1*	14.942	22	12	2	15	16
Log2*	191.532	483	90	2	323	344
Log3*	706.667	78	13	2	82	94

*the conditions of these verification runs and of these logfiles are found in appendix III

**in these verifications the same logfiles are used except for the times of the last transition, these times are changed into incorrect ones.

Evaluation

The Time test-specific model only checks whether the time of the *requestReady* is larger than the time of the *request.* This is not a very valuable verification since this is always the case. However, the model can be made more precise. Table 6 shows that the verification time needed for this model has a relation with the number of log model transitions.

5.4 Composition

Besides the standalone verification and analysis, reference, synchronization and test-specific models can be composed together, forming one specification. The composition of the specification can be verified at once, but because of the larger state space (Section 3.3) and more abstract data type instances (Section 4.2.3) this verification is not faster.

Input

The tool chain input exists of three parts: a logfile, the composed specification, consisting of the models: Status, Print, Data and StatusPrint (see Appendix I) and Perl regular expressions (see Appendix II a).

Results

Results of the verification and analysis of the functionality of this composition specification are given in Table 7.

Table 7: Results verification and analysis composition specification model

	Logfile lines	log model transitions	Time verification (s)
Log1*	14.942	704	10,800
Log2*	191.532	890	> 14,400
Log3*	706.667	2010	> 14,400

*the conditions of these verification runs and of these logfiles are found in appendix III

Evaluation

The table shows that it is possible to verify a large composition of the specification. However, the second and third logfile emphasize that this verification is not as efficient as the verification of the specification models independently. The reason for this is the abstract data type verification method of the CADP tool chain (Section 4.2.3).

The purpose of analysis is not to compel belief but rather to suggest doubt.

(Imre Lakatos)

6 Evaluation, conclusions and recommendations

This chapter evaluates the AVATR method and tool chain, it discusses alternatives and gives a conclusion and recommendations.

Overview of this chapter
Section 6.1 discusses the chosen method and tool. Section 6.2 gives the conclusions of the AVATR method and tool and section 6.3 gives recommendations.

6.1 Evaluation

The evaluation of the AVATR method and tool chain is done in three parts:

- transformation of the logfile;
- creation of the specification;
- comparison of the formalized logfile with the specification.

6.1.1 Formalization of the logfile

The formalization of the logfile is done by examining each line of it with a Perl script, which contains regular expressions (see appendix II a). The script creates a log model and corresponding library, header and definition files (see appendix III).

The use of Perl scripting language has the following advantages:

- it is fast; a logfile of 25 megabyte is formalized in half a second with actions of different protocols;
- it is flexible; a Perl script can easily be adjusted;
- the Perl interpreter is free and a standard part of several engineering toolboxes;

Since this is fast and efficient, no alternatives are examined.

6.1.2 Creation of the specification

Specification files are created in the formal specification language LOTOS.

Method

Printer requirements are formalized in reference, synchronization and test-specific models, together forming a specification of the printer. These models provide:

- maintainability; changes in one of the protocols, can be made to one of the reference models and possibly to one of the synchronization models or test-specific models, without changing the rest of the specification;
- flexibility; when only a specific part of the requirements has to be verified, only this part of the requirements can be used in the verification and analysis.

Furthermore, the meaning of the specification models is intuitive and easy to grasp: reference models describe a coherent set of printer actions (typical a protocol), synchronization models describe actions needed for synchronization of reference models and test-specific models describe test-specific part of the printer functionality.

However, the analysis whether the last transitions of a protocol has been taken, is performed with the help of a final transition (section 3.3), this is less intuitive. For a precise analysis more final transitions have to be added. The alternative method (creation of composition with log model, section 3.4.3) has the same issue here.

Language

The creation of these specification models is done in the formal specification language LOTOS.

The use of LOTOS has the following advantages:

- it is a well-documented language;
- it is a language dedicated to model protocols;
- it is a language that only supports enforced synchronization; each process on a gate, must participate in any communication occurring on that gate (Section 2.2.1). This is used for the composition of reference, synchronization and test-specific models;
- it is a language for which tools are available.

However, it is not always easy to express desired behaviour in LOTOS since:

- there are no global variables, this could be useful to verify global properties, (e.g., timing since the start of the verification);
- data types are rather restricted, all instances of a specific data type must be defined individually.

An alternative for the specification language LOTOS is the formal language μCRL. This language, like LOTOS, is well-documented, used to model protocols and provides possibilities to define abstract data types. μCRL has been extended with features to express time, which is, however, not supported by tools [29]. This language has, however, the disadvantage that it not implements enforced synchronization (Section 4.3.1).

A second alternative is the creation of a new language, or the use of a more general language (e.g., a modeling language or a programming language). This has the disadvantage that there is no tool support for the trace inclusion relation.

Tool
The LOTOS models are created in the textual LOTOS editor, VIM.

Since the individual specification files are small, an overview of the specification files is easy preserved. Due to syntax highlighting, LOTOS files are easily read.

A nice option would be a graphical editor, which is not available for LOTOS. LOTOS files, created in a text editor, can be transformed to a graphical representation, which however cannot be graphically edited.

6.1.3 Comparison of the formalized logfile with the specification

The relation between the specification and the log model is a trace inclusion relation, verified with the CADP Exhibitor tool. This section discusses the method which determines this relation and the tool that implements it.

Method
The trace inclusion relation (section 2.1.2) between log model and specification is verified with an on-the-fly composition of the specification (section 3.3.1). This has the advantage that it is:

- fast, no composition of specification models has to be created before the verification;
- small, large parts of the specification are not used, since only one sequence is visited.

The alternative for this method is the parallel composition of the specification with the log model (section 3.4.S). This composition results in a system similar to the log model, if the behaviour of the printer is correct. This alternative consists, besides the composition, of a check whether a final transition is reached or not. This transition has to be added in the log model and in the specification, even when it is not important to know whether last protocol transitions are used or not, which is a disadvantage.

Tool
The verification and analysis is done with the CADP tool set. The use of CADP has the following advantages (section 4.2.3):

- the tool has good tool support;
- the tool has an user-friendly interface;
- the possibility to use LOTOS specifications;
- the possibility to define and use abstract data types.

However, CADP has some disadvantages:

- it does not implement all functionality of the specification language LOTOS, e.g., it is not possible to create a parallel composition in combination with a recursive process;
- the use of abstract data types slows down the verification and analysis;

- is has yearly license costs.

Section 4.3 gives five alternatives for CADP Exhibitor:

- TETRA, this tool was not available;
- TorX, this tool uses CADP for LOTOS parsing
- CADP Bisimulator, this tool is slower than Exhibitor
- μCRL toolset, uses CADP for trace inclusion functionality.

The remaining alternative is the tool UPPAAL. This tool does not implement functionality to compare LTSs with each other. However, it is possible to use this tool in combination with the composition alternative of the chosen method (Section 3.4.3).

6.2 Experience

In the feasibility study (Chapter 5) three protocols, one synchronization model and two test specific models are created by which printer logfiles are verified and analyzed.

This section describes experiences about the creation of models and the acceptance in Océ.

6.2.1 Creation of specification

The protocols in a printer differ in complexity. Besides that, creation of models from protocols is dependent of the knowledge of the protocols and of the interaction of protocols.

The Status model is described in an LTS representation in the Status requirement document. Furthermore, the Status protocol does not use identification numbers and only one protocol instance exists in the printer. The creation of the Status model could be copied from the Status requirement document, which can be done within an hour.

The Print and Data protocol are more complex. They consists of identification numbers and more instances of the protocol can be used simultaneously. The requirement documents of these protocols are more concise than that of the Status protocol. Some exceptional cases are illustrated with a sequence diagram but no LTSs are given. Hence, the creation of an LTS from these protocols is more time consuming. The time needed for the creation of a model for these protocols is, depending of the knowledge of the protocols, between one and five hours.

Synchronization and test-specific models are typical small and less complex. For these models also holds that more knowledge of protocols and relations between them decreases the time needed for creation the models.

The models created do not take into account exceptional behavior and error handling. These aspects can be modeled the same way, since it is part of printer protocols.

6.2.2 Detection of failures

The AVATR method and tool chain are developed to detect incorrect printer behavior by means of logfiles. The feasibility study shows verification times with incorrect verification results. These incorrect results are caused by injected incorrect log transitions, which where all detected correctly.

Besides logfiles with injected failures, logfiles with existing incorrect actions are verified. In these logfiles the place of the failure already was known. With the AVATR method these failures are detected when they were in one of the protocols modeled (incorrect status transitions). Not modeled protocols are not verified and hence no failures are found in these protocols.

New failures are not found. Every time a new failure appeared the specification model had to be adapted.

6.2.3 Acceptance in Océ

Feedback from Océ engineers on the developed method and tool chain comes down to a calculation of the required effort and gained profit.

For Océ engineers a new method is useful when it proves its value. However the developed method and tool chain only discovered known (mostly injected) failures so far. Furthermore, the verification needs a description of requirements in a formal specification language, which takes effort and time. It is sometimes a cumbersome task since requirement documents can be concise. A third aspect is that the generation of a LOTOS specification (textual) is different of the drawing of a LTS. Hence it takes even more time and effort to learn the LOTOS and the AVATR tool chain.

On the other hand, automatic verification and analysis is seen as valuable, since it is possible to verify and analyze printers automatic and more thorough. Failures can be automatic found and protection mechanisms in the printer can be decreased.

6.3 Conclusions

This thesis describes the AVATR method and tool chain as developed at Océ-Technologies. The AVATR tool chain is capable of automatic verification and analysis of test results of a printer of Océ. The feasibility study, carried out as part of this project shows acceptable running times, even for many protocol instances. Using AVATR, logfiles can be verified and analyzed more thoroughly than with existing methods.

Results obtained in the AVATR project are the following:

- A method has been developed to transform a logfile into a log model. This transformation filters unneeded details out and formalizes used printer actions.
- A method has been developed to create a specification, consisting of reference, synchronization, and test-specific models, which describes (relevant) parts of printer requirements on a maintainable and flexible way.

- A method has been developed to verify and analyze the log model with the specification.

A proof of concept has been given for which the developed method has been implemented in a tool chain. Results of the implementation show that it is possible to verify and analyze printers of Océ by means of logfiles.

The running times for these verification and analyses are acceptable, even for many protocol instances. The use of large amounts of data type instances, however, slows down the verification.

6.4 Recommendations

The developed AVATR method and tool chain are able to verify and analyze printer logfiles. However to fully profit from AVATR some improvements can be made.

Small models without many instances in the printer (e.g., Status model, Print model) can be verified in acceptable running times, mostly below 15 minutes. The verification of larger models or compositions with many instances in the printer (Data model, composition of specification) exceed three hours, which is unacceptable. Hence it is recommended to search another tool for verification of these specification models.

The second recommendation is about the documentation. It should contain a clear unambiguous description of the protocols described, preferably in LTS notation. This can be helpful for engineering and debugging but is also helpful for easily adaptation of the AVATR method.

The third recommendation is that naming conventions are adapted to the requirements as well as to the logfile. Currently there are differences between descriptions in requirement documents and the logfile (e.g., in the Print protocol requirements the action downloadrequestReady is the same as the log action downloadready), and between different log statements in one logfile (e.g., notations for 'identifier' are: *id*, *Id*, *requestId*, *id:*, *id=*).

Furthermore, an interface should be made between a graphical tool and the AVATR tool chain, to create specifications more intuitive in transition systems. For this graphical tool RoseRT can be used, which is part of the existing development environment.

Finally, it is recommended to create a coupling between the AVATR tool chain output and the original logfile. The AVATR tool chain directs to transitions in the transformed logfile, the log model. A statement in a log model is less insightful than a statement in the original logfile since in the logfile information is present from other protocols.

Knowledge of what is does not open the door directly to what should be.

(Albert Einstein)

References

[1] J. Van der Velden, *Van nature innovatief,* Almedia, 2007.

[4] H. Gravel, M. Jorgensen, R. Mateescu, C. Pecheur, M. Sighireanu, B. Vivien, *CADP'79 – status, Applications and Perspectives,* I. Lovrek (Ed.), Proceedings of the 2nd COST 247 international workshop on applied formal methods in system design, 1997.

[5] T. Bolognesi, E. Brinksma, *Introduction to the ISO specification language LOTOS*, computer networks and ISDN systems, 14(1): 25-59, 1978.

[6] L. Logrippo, M. Faci, M. Haj-Hussein, *An introduction to LOTOS: Learning by examples*. Computer networks and ISDN systems 23: 325-342, 1992.

[7] J. de Meer, R. Roth, S. Vuong, *Introduction to Algebraic specifications based on the language ACT ONE,* Computer networks and ISDN systems 23: 363-392, 1992.

[8] J. A. Manas, *A Tutorial on ADT semantics for LOTOS users Part I: Fundamental Concepts*, Dept. Ingenieria Telematica Ciudad Universiaria E-28040 Madrid, 1988.

[9] J. M. T. Romijn, *Analysing Industrial Protocols with Formal Methods*, PhD thesis, University of Twente, 1999.

[10] B. Broekman, E. Notenboom. *Testing embedded software,* Addison-Wesley, 2005.

[11] J. P. Bowen, M. G. Hinchey, *The use of industrial-strength formal methods,* Proceedings of 21st International Cumputer Software and Application Conference, 1997.

[12] E.Ciapessoni, A.Coen-Porsini, E.Crivelli, D.Mandrioli, P.Mirandrola, A.Morzenti, *From formal models to formally-based methods: an industrial experience,* ACM Transactions on Software Engineering and Methodology, Volume 8, Issue 1, pages 79-113, 1999.

[13] M. Stoelinga, Testing rechniques, Universiteit Twente, Enschede, 2007.

[14] R. J. van Glabbeek, *The linear time – branching time spectrum*. In J.C.M. Baeten and J.W. Klop, editors, CONCUR'90, Lecture Notes in Computer Science 458, pages 278-297, Springer-Verlag, 1990.

[15] R.J. van Glabbeek, *The linear time – branching time spectrum II (the semantics of sequential systems with silent moves)*. In E. Best, editor, CONCUR'93, Lecture notes in Computer Science 715, pages 66-81, Springer-Verlag, 1993.

[16] H. Ehrig, B. Mahr. *Fundamentals of algebraic specification -1,* Springer-Verlag, Berlin, 1985.

[19] G. V. Bochmann, O. Bellal. *Test result analysis with respect to formal specifications*, Proc. 2nd Int. Workshop on Protocol Test Systems, pages 272-294, Berlin, 1989.

[20] G. V. Bochmann, D. Desbiens, M. Dubuc, D. Ouimet, F. Saba, *Test result analysis and validation of test verdicts,* The proceedings of the Workshop on Protocol Test Systems, McLean, Virginia, Oct. 1990.

[21] A. Belinfante, J. Feenstra, R.G. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, and L. Heerink, *Formal Test Automation: A Simple Experiment*. In G. Csopaki, S. Dibuz, and K. Tarnay, editors, Int. Workshop on Testing of Communicating Systems 12, pages 179-196, Kluwer Academic Publishers, 1999.

[23] J. Tretmans, E. Brinksma. *TorX: Automated Model Based Testing,* First European Conference on Model-Driven Software Engineering, pages 31-43, Nuremberg, Germany, December 2003.

[24] G. Behrmann, A. David, and K. G. Larsen, *A Tutorial on UPPAAL*, Proceedings Formal Methods for the Desings of Real-time Systems, Lecture notes in Computer Science, volume 3185, November 2004.

[26] K. G. Larsen, P. Pettersson and W. Yi, *Uppaal in a Nutshell*, Springer International Journal of Software Tools for Technology Transfer 1(1+2), 1997.

[27] D. Bergamini, N. Descoubes, C. Joubert, and R. Mateescu, *BISIMULATOR: A Modular Tool for On-the-Fly Equivalence Checking*, Lecture notes in Computer Science, volume3440, April 2005.

[28] A.G. Wouters, *Manual for the muCRL tool set,* version 2.8.2, Technical Report SEN-R0130, CWI, Amsterdam, 2001.

[30] I.A. van Langevelde, *A compact file format for labeled transition systems*, Technical Report SEN-R0102, CWI, Amsterdam, 2001.

Internet

[2] VIM – http://www.vim.org/

[3] CADP – http://www.inrialpes.fr/vasy/cadp

[18] Perl – http://www.perl.org/

[17] Exhibitor – http://www.inrialpes.fr/vasy/cadp/man/exhibitor.html

[22] TorX – http://fmt.cs.utwente.nl/tools/torx

[25] UPPAAL – http://www.uppaal.org/

[29] mCRL toolset – http://homepages.cwi.nl/~mcrl/mutool.html