

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

Reversed
Engineering Based
on Log data

By
Maarten Damen

Supervisors:

dr. ir. Michel A. Reniers (TU/e)
dr. Wouter J.W. Geurts (Logica)

Eindhoven, July 2009

Abstract

Within the software development business almost everyone knows that theoretically modelling a system in the beginning of a project pays off during the project. Still most companies do this only for highly reliable and critical systems, because designing models in the beginning of a project is hard and considered to be time consuming.

Models become more important, not only to create new software systems but also to perform maintenance on existing systems and to prove a certain level of quality of a system. Therefore lately more and more applications become available to do reversed engineering based on the systems code. Since it is not always the case that the source code of a system is available and source code in most cases only provides a static model of the system or at best a state model if the code is well structured, it is interesting to know if it is possible to create a model of a system from its log data.

In this document it is shown how from log data, containing only input and output messages of a single process, a model can be constructed. In the first place it is shown how time driven messages can be found, both continuous and discontinuous. After that some ideas are sketched of how the spurious behavior of the system can be modelled. To describe the resulting model the usage of some kind of Timed I/O-automata is chosen, since this could combine the inputs and outputs directly to each other and offered the possibility to separate the timing aspects of the model.

The results for the time driven part were very good, for the tested processes all timers are found, including the responses to the messages that are time driven, which are not distinguished in this paper. There was one case where the results of the method given in this paper were unexpected and where it could not be decided if the messages were time driven or not. This was caused by two timers sending the same message with different periods. For the spurious behavior, the results are a bit less clear since the method is not completely finished for this part and there are many points of discussion left.

Preface

During my study at the Avans Hogeschool in Breda I got interested in the more theoretical part of the software development process. At this point I decided to do a Master study Computer Science and Engineering at the Technische Universiteit Eindhoven. While performing this master program I experienced the design, analysis and verification oriented courses as the most interesting, although they were pretty complex and time consuming for me.

Since for completing the master study at the TU/e it is needed to fulfill a master project of 30 ECTS, that leads to a masterpiece showing my design and research capabilities, I decided to talk with Dhr. Groote who is the head of the “Ontwerp en Analyse van Systemen” (OAS), Dutch for design and analysis of systems, area of expertise. During this initial talk Dhr. Groote came up with this assignment that was originated from the TWINS project where Logica and the TU/e both participated in.

This thesis is the results of the master project as conducted at Logica Netherlands. The project has been performed within the graduation division of Logica Eindhoven, Working Tomorrow, which offers an opportunity for graduation students to fulfill their master project or final internship within a company as Logica. For the past ten months I have been working on Reversed Engineering Based on Log Data, which I have experienced as a quite difficult but very interesting assignment. In these past ten months I learned a lot about the possibilities and the difficulties of reversed engineering of software of which the source code is not available. Although I did not manage to construct a complete model that covers all different aspects of the system in the time I got, I think it gave me a good idea of the possibilities and the practical difficulties that exist within a project like this.

In this preface there are a few people I wish to thank for the support during my master project. With the risk of forgetting someone I would like to express my gratitude to them here:

- First of all I would like to thank Michel Reniers for his supervision from the TU/e, for coming up with ideas of how to continue during our weekly meetings and for his willingness to be part of my examination board.
- Second, Wouter Geurts for his supervision with respect to the content of the subject, making information and (log) data of the system available to me, being the system expert and his willingness to be part of my examination board.
- Lambert Mühlenberg and Frank Buve for the process supervision they gave me during my internship at Logica Eindhoven.
- Pieter Cuijpers for his willingness to be part of my examination board.

- Anton van Gelderen for making it possible to conduct my master project at Working Tomorrow Eindhoven.
- Then I would like to thank the other graduates at Working Tomorrow Eindhoven for the pleasant time we had in and outside the office.
- Last but not least I want to thank my family and friends for their great support during my master project, but also during my entire time studying.

Maarten Damen

Eindhoven, July 14, 2009

Contents

1	Introduction	1
1.1	Modelling	1
1.2	Related work	2
1.3	Structure of the document	4
2	The BOS System	5
2.1	Introduction	5
2.2	Safety Critical System	7
3	Preliminaries	9
3.1	I/O automata	9
3.2	Timed I/O automata	9
4	Logdata from BOS	11
4.1	Introduction	11
4.2	Test cases	11
4.3	Operational mode	14
4.4	Generalization	16
5	Analysis	17
5.1	Time	17
5.2	Distinguishing messages	17
5.3	Time stamps	18
6	Timing	21
6.1	Introduction	21
6.2	Continuous timing	23
6.3	Discontinuous timing	28
7	Spurious behavior	33
7.1	Introduction	33
7.2	Extracting a model	34
7.3	Procedure by example	36
7.4	Content of a message	38
8	Conclusion and future work	41
8.1	Conclusion	41

8.2 Future work	42
Bibliography	47

List of Figures

1.1	Example of a workflow process modelled as a Petri net, taken from [28].	4
2.1	Geographical place and different views of the Europoortkering.	6
3.1	Example of an I/O-automaton.	9
3.2	Example of a timed-I/O automaton	10
4.1	The two different environments the system can run.	12
4.2	Example of a part of a test suite log file.	13
4.3	Graphical visualization of constructing a log file.	15
4.4	Example of a part of an operational mode log file.	15
6.1	Message occurrence compared with the time, for each message type.	22
6.2	Classification of message types.	22
6.3	Example of the range $[t - \delta, t + \delta]$	25
6.4	Example of a continuous time driven message m with period Π	27
6.5	Part of the model that can be constructed after continuous timing.	28
6.6	The continuous time driven part for one process from the BOS system.	28
6.7	Example of continuous time driven blocks.	29
6.8	Example of a discontinuous time driven message m with period Π	31
6.9	Part of the model that can be constructed after continuous timing.	31
6.10	The discontinuous time driven part for one process from the BOS system	32
7.1	Example of a situation where a state change is applied.	34
7.2	Example of a situation where no further state change is applied.	35
7.3	Examples of a situation where the output comes after a large period of time.	35
7.4	Example of how a loop can be constructed.	36
7.5	Visualization for the procedure in a few steps.	37
7.6	Part of the model that can be constructed after continuous timing.	38
7.7	Example of how content has influence on the model, in the first case without content and the second case with content taken into account.	39
7.8	Example of a spurious model from a log file without taking content into account.	39
7.9	Example of a spurious model from a log file with taking content into account.	40
8.1	Example of two timer sending the same message m with different periods.	43
8.2	Example of first case where the period of the timer changes.	43
8.3	Example of how a time driven message with a reply can be modelled.	44

Chapter 1

Introduction

1.1 Modelling

The last decades have shown great advances in model-based techniques for specification, implementation, verification and validation. These techniques include model checking, model-based code generation and model-based test-case generation [8, 16, 20]. The main obstacle for a more widely usage of techniques like these is the state space explosion which causes the models to become unmanageably large. This happens in many practical cases, and for this reason the state space explosion is the main interest of current research on model checking [8]. Another problem with these techniques is that they assume that a formal model of the system under study is available [6], which ideally should have been made at the design phase of the system. However even though everyone knows that modelling a system at the design phase of the development pays off during the project, this is not done in many cases. The formal modelling of a system at the design phase is hard and considered to be time consuming. This is the reason that it is only done for safety critical applications, i.e. applications for which failure is unacceptable, e.g. medical instruments, space instruments, traffic control systems and other examples too numerous to list. This means that in many cases there is no formal model of the system available and even if it is available in many cases it is out-dated, incomplete and not consistent with the system anymore.

Lately more and more techniques have become available to create models afterwards from the systems source code. Many of these techniques give a static or dynamic model, but can also be used to generate abstract models of the system [9, 18]. Problems with these techniques arise when the system consists of combined hard- and software, or the softwares source code is not available, e.g. due to third-party software. Also the static analysis of source code is heavily depending on the implementation language, and the coding style that is used.

In practice this means that a model has to be made out of other observations from the system. In many cases there is no other observable behavior of the system except looking at its traces, i.e. the systems sequences of input and output actions. These sequences can be observed from the system directly, or from log data that is saved by the system. A tool that assists in constructing a model from these sequences will be easier to adopt to new programming languages and new programs, since it is not depending on the source code but on the input/output traces of the system and will not need the source code at all.

This gives the problem definition for this research project: “Is it possible to extract a useful model from given log data of an existing system?” With a useful model it is meant that it should at least contain the traces that are available in the log file and should be

manageable in size. I.e. a tree where every possible trace gets its own path, would fulfill the property of being able to generate these traces. However it will get extremely large and have less predictive value for traces that were not available in the log data. The predictive value of a model can be described as the ratio of all correct traces in the model minus the traces that were in the log file, divided by all possible traces in the model.

1.2 Related work

Angluin's learning

Angluin [4] described a method in a seminar paper for learning finite-state automata from a system, under the assumption that it is possible to ask the system whether a string is a member of the language of the automaton or not. Practically this means that the system should have two characteristics:

- It should be possible to send a string to the system.
- The system should give a signal whether it could execute the string or not.

From a finite state machine traces can be generated that may occur in the system. Steffen et al. [15] has used this technique successfully to generate test sequences and Peled et al. [14] used the method described by Angluin for developing techniques for conformance testing of finite automata. Angluin's technique has also been used by Berg et al. [6] in rather simple real-world examples like small buffers, schedulers and mutual exclusion protocols. However they also tested it on larger samples and they claimed that they failed since finding a counterexample took too much time, because the state space got too large.

For the purpose of this research project Angluin's technique can not be used, since the system under interest is operational and may not be interrupted for research purposes. This means that it is not possible to request data from the system or to send strings to check whether they could be executed or not. Also the traces and messages considered in this project are a bit more complex as shown in Chapter 4.

The Nerode Algorithm

The Nerode algorithm [21] can be used with a finite amount of data to model an unknown black box, e.g. a process of which nothing is known, if the number of states needed to model this unknown black box is specified and the experiments which produce the data, the input sequences, can be chosen. Arbib and Manes [5] discuss generalization of this technique to abstract machines in a category theoretic framework and E. Mark Gold [12] adapted the algorithm to the problem of automaton identification from requested data. Although there are differences in these techniques, all these techniques assume the following properties:

- The black box is available.
- Any finite number of experiments, chosen at will, can be performed on the black box.
- The black box can be reset to its initial state before each experiment.

This means that this algorithm or a variant of it can not be used in this project, since the processes of the system, the black boxes, are not available. The data for this project is given, so the experiments can not be chosen at will, which makes the method unapplicable.

Automaton Identification from Given Data

In 1973 Trakhtenbrot and Barzdin [25] show how a finite automaton can be constructed from given sets of accepted and rejected words as long as the sets are m -complete. The sets being m -complete means that all possible words with at most length m from the language Σ^* are contained in either the set of accepted words S_+ or rejected words S_- . Trakhtenbrot and Barzdin show that if a deterministic finite automata \mathcal{A} has n states it follows that \mathcal{A} can be constructed from a m -complete sample, with $m \geq 2n - 2$. Later E. Mark Gold [13] and Dana Angluin [3] showed that in general the discovery of a minimal automaton satisfying a given set of accepted and rejected words is NP-complete.

The methods described in these works are not applicable for this research project since the log files only give accepted sequences of messages. Taking for S_- the complement of what is in the log files, might give a model, but it will have no predictive value at all since no new traces can be generated. Of course the set of rejected words S_- can also be seen as an empty set in this project, but then this method will still not work. When for S_- the empty set is chosen, the number of states n needs to be known, which is not the case, so n needs to be taken very large. When n is very large the sample should be m -complete for a large m , which can not be known and is very likely not the case with an empty S_- . This means that the method can not be used for this research project.

Workflow mining

Van der Aalst et al. [27, 28, 29] described another method to construct a model from traces of a process, which is called workflow mining. The idea of workflow mining is to create a workflow design from a workflow log. Such a workflow log consists of events that are logged following the assumptions that:

- Each event refers to a task, a well defined step in the workflow.
- Each event refers to a case, a workflow instance.
- Events are totally ordered, it is always possible to say for two steps in the workflow which appeared first.

A model can be created from these workflow logs by counting the frequencies that a *Task* α is followed by a *Task* β , possibly with one or more tasks in between, and visa versa. By doing this for all tasks it is possible to discover direct and indirect successor relations, a *Task* α always appears before a *Task* β . Choices between tasks, *Task* α is sometimes followed by *Task* β and sometimes by *Task* γ . Parallel tasks, *Task* α and *Task* β appear in any order. From this a model can be constructed that for example might look like the model in Figure 1.1.

Although the process in this case is a business process, where tasks are executed, the idea is more or less the same as for a software process. Since the technique that is used is based on the events that occur and traces from a system can also be seen as event logs, this technique might be applicable in some cases to generate a model from traces. In this research project this technique could not be used, since it is unable to handle time driven events, events that are triggered by the passing of time.

Now it might be possible to apply this method after filtering out the time driven events, so that only the spurious events are left over. The problem then is that the technique used

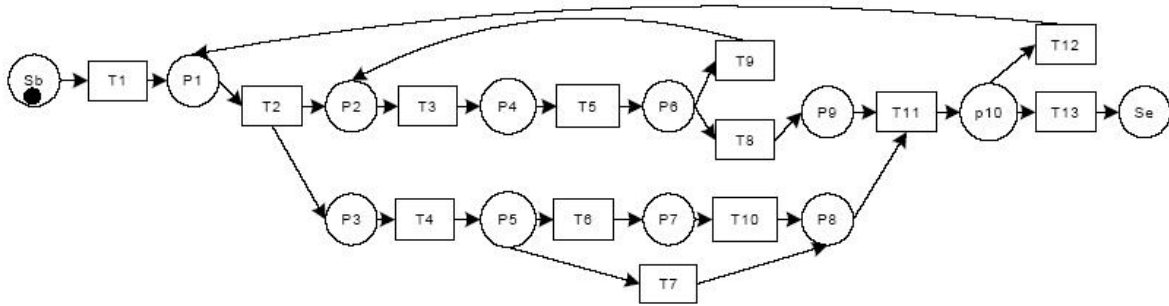


Figure 1.1: Example of a workflow process modelled as a Petri net, taken from [28].

by workflow mining assumes that an event, e.g. *Task α* , only can occur in one place in the model, which is in this case not realistic. Besides that it is not realistic it also gives rise to problems, since when an event may occur only at one place in the model the past get lost. This means that all events that happened before get irrelevant for the future, which is always the case of course for a certain state in a model. A problem occurs when there is an event that keeps returning all the time, in the worst case after each other event, but is not time driven. All these events are mapped together and all combinations of traces become possible, that might not be possible in the system.

1.3 Structure of the document

In the next chapter the system under interest for this research project is introduced. The reason why it is created, why it is safety critical and some of the characteristics of the system are given. Chapter 3 gives and explains some definitions that are used. In Chapter 4 the format of the data that is used is explained. Chapter 5 analyzes the data and some of its properties. Chapter 6 and 7 give the different aspects of creating a model from this data and the thesis concludes with the conclusions and future work in Chapter 8.

Chapter 2

The BOS System

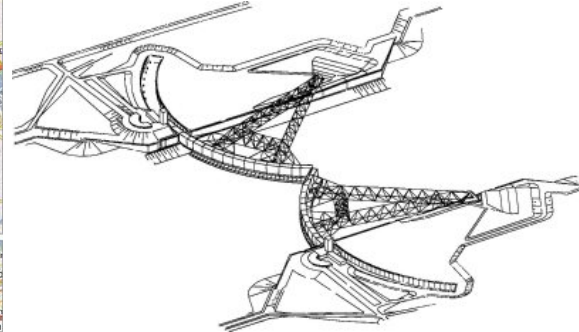
2.1 Introduction

This chapter gives an overview of the BOS system, the reason why the Maeslantkering is built, the requirements that there were for the development, a short description of the hardware and why this is a safety critical system for which the use of formal modelling is helpful. Then a description of how the software is built up will be given, that it consists of small detached and simple components, that these components communicate over channels, that this communication over the channels is logged by channel logging and that these components are formally modelled so that an extracted model can be compared with the original one. Information to write this chapter mainly comes from Geurts et al. [11], Tretmans et al. [26] and Wijbrans et al. [30].

The Netherlands are located in a low delta near the North Sea into which important rivers such as the Rhine, Meuse and IJssel flow. The shape of the Netherlands has been created through the ages from the struggle against the sea. For this reason the great flooding of 1953 in the southwest part of the Netherlands that reached far into Noord-Brabant was a great shock to the Netherlands. It was demonstrating once again that the country was vulnerable and that people still were not safe from the water. Even through the first ideas of the Delta Plan were already thought of by Johan van Veen [23, 24] a few years after the finishing of the Afsluitdijk in 1933 and the first dam, the Brielsegatdam, which is part of what we now know as the Delta Works was already built before the flooding, the flooding was decisive to construct the Delta Works and shorten the coast line with about 700 kilometers. This Delta Plan described ideas like building a network of dams in Zeeland, raising and upgrading/reenforcing existing dikes, such that these Deltaworks reach the “normfrequency” of $1/(10.000 \text{ years})$, meaning a designed resistance against the stormcharacteristics that have a frequency of occurring more than once every 10.000 years.

The realization of the other parts of the Delta Works started quite soon after the flooding. The dam network in Zeeland was finished in 1986, which made the Nieuwe Waterweg the new weakest spot in the defence. The problem with this place is that creating a dam or even a barrier such as the Oosterscheldekering, like at other places in the Delta Works, was not a possibility since it is one of the major and important shipping routes to Rotterdam and it is also the major outlet for water from the Rhine, see Figure 2.1(a). At the same time leaving it completely open is a great risk for the flooding of Rotterdam and surrounding areas. To protect this area from flooding a storm surge barrier is created between Hoek van Holland and Maassluis, which is called the Maeslantkering, see Figure 2.1(c), and one located in the

Hartelkanaal near Spijkenisse, called the Hartelkering, see Figure 2.1(d), which together are called the Europoortkering.



(a) Location of the Maeslantkering and the Hartelkering, taken from Google Maps.

(b) Design drawing of the Maeslantkering, taken from <http://www.deltawerken.com>.



(c) Photograph of the Maeslantkering, taken from <http://www.deltawerken.com>.

(d) Photograph of the Hartelkering, taken from <http://www.deltawerken.com>.

Figure 2.1: Geographical place and different views of the Europoortkering.

From the problems discussed a few requirements are constructed that eventually led to the design, see Figure 2.1(b), of the movable storm surge barrier in the Nieuwe Waterweg as it is today. These requirements are the following once:

- Rotterdam and surrounding areas should be protected from flooding.
- The harbor of Rotterdam should be reachable except when the wether is very bad, e.g. with high flooding risks.
- Rotterdam should not be flooded by water coming from the Rhine.

The Maeslantkering, Figure 2.1(c), consists of two hollow walls, also called sector doors, that are put away into docks that are built in the river banks when the wether is not extremely bad and there is no risk of flooding. The sector doors are connected to pivot points by steel arms that both are as large as the Eiffel Tower, since they should be able to resist huge forces from incoming water in very bad wether circumstances. Only when storms are expected

that have a flooding risk, the Maeslantkering will be closed. This closing of the storm surge barrier consists of filling the docks where the sector doors rest with water, which makes the hollow doors float, moving them to the center of the Nieuwe Waterweg and fill them with water, which makes them sink. The movable design of the Maeslantkering has another great advantage, namely the construction and maintenance of the storm surge barrier does not interfere with the ship traffic.

The Hartelkering, Figure 2.1(d), consists of two eclipse formed walls that can move up and down. Under normal circumstances the bottom of these walls is just a bit higher than the height of the bridge next to it, see Figure 2.1(d), which is about 14 meters above the normal sealevel and ensures that there is no disadvantage for the ship traffic. When closed, which is in the general case at the same time as the Maeslantkering, the walls of the Hartelkering are just a little bit higher than the raised sealevel. This means that water waves will come over it, which is not a problem, since otherwise the complete Europoort area in front of the Hartelkering would risk flooding.

The main requirement of the system was that the storm surge barriers should be as reliable as a dam. For this reason it was considered to be the safest to let a computer program control the opening and closing of the barriers. Note that both storm surge barriers are connected to the same system and that in the general case they are closed and opened at the same time.

2.2 Safety Critical System

The system that controls the opening and closing of the two storm surge barriers is called BOS (Beslis & Ondersteunend Systeem), which is a Dutch abbreviation for decision & support system. It will decide without interference of any person, based on weather forecasts, model-based water predictions and the weather of the past days, when to close and open the storm surge barriers. This system is a safety critical system since not closing the barriers when needed is clearly a great safety problem, since it would lead to flooding of Rotterdam and surrounding areas. But also closing it when it is not needed is a safety problem, since it would lead to millions of euros economical damage, because ship traffic will be restricted, while also there is the risk of flooding from water from the Rhine when the water is blocked by the barrier.

The requirement of a barrier that is as reliable as a dike, having the same normfrequency, imposes quality criteria to the central decision system. Additional on the normfrequency regarding protection of land from flooding, additional, sharper, requirements have been set on blocking the Nieuwe Waterweg. This might seem surprising might, but not opening the barrier can cause flooding by water from the Rhine, millions of euros economical damage by restricted ship traffic and also destruction of the complete barrier, if due to water flowing from the Rhine the pressure on the inside, land-side, is higher than the pressure on the outside, sea-side. To make sure the system satisfies these quality criteria the design and development of the BOS has been guided by the IEC 61508 standard [19]. This standard describes different categories for *Safety Integrity Levels* (SIL). According to this categorization BOS belongs to SIL4 which is the highest level.

To get this level some “highly recommended” techniques had to be applied and one of these techniques is formal modelling. The BOS system consists of small detached and simple components called processes that communicate with each other and the “outside” world over channels by message passing. This incurs the risk of deadlocks and bad data due to

synchronization errors. To prevent this PROMELA [17] was used for modelling the interaction between the processes and the interaction between BOS and the “outside” world. To specify the functions performed by the processes Z [22] was used. Hence the behavioral view was modelled by PROMELA and the functional view using Z. The rationale for selecting these formal methods is described in Chaudron et al. [7].

The usage of formal methods in this project has helped to achieve quality software, and makes it an interesting system for this research project, but standard IEC 61508 that was used to guide the design and the development of BOS cannot give any guarantees that the software has a certain quality. Also it is important to note that only very small parts of the BOS system were dealt with the highest level of formality and that not a single line of program code was completely proven correct. These two points together mean that it is impossible to determine the reliability of the software itself. For this reason it is interesting to know, if it is possible to create a model from the messages between the processes that are logged by channel logging. An ultimate goal would be to check this extracted model with the original model and to check if it is possible to say something about the quality of the software reached.

Chapter 3

Preliminaries

3.1 I/O automata

To describe the model extracted from the log files input/output-automata are used. I/O-automata, as described by Lynch and Tuttle [2], provide a mathematical model for discrete event driven systems consisting of concurrently-operating components. Such a system does not simply compute some function from its input and halts, but it continuously receives inputs from its environment and reacts on these. By using I/O-automata each component can be modelled as an I/O-automaton and these automata are connected to each other by input/output combinations, since the output from one automaton is the input for another automaton.

The I/O-automata used here combine inputs and outputs directly to each other. The notation used is that the automaton consists of states that are connected by transitions. These transitions are labeled by combinations of inputs and outputs, where the input is before the slash and the outputs comma separated after the slash, see Figure 3.1. There is always at most one input for a transition followed by zero or more outputs. In case there is no input or output for a certain transition this will be visualized by a hyphen. Also there is an initial state that is marked by the incoming arrow and gives a starting point for the system.

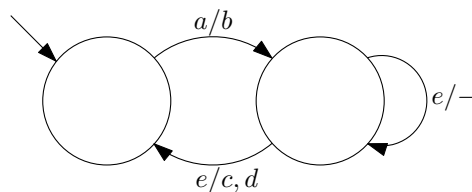


Figure 3.1: Example of an I/O-automaton.

3.2 Timed I/O automata

To be able to express timing in the I/O-automata described in Section 3.1, the I/O-automata are extended with known techniques of modelling time in timed automata, as described by Alur et al. [1] and Dill et al. [10], in the following way. In the state that can trigger a time depending message a local timer is added and a clock constraint is given, see Figure 3.2. This clock constrained is called an invariant and time can elapse in the state as long as the invariant

is true. On the transition a guard is given that expresses when the transition is enabled. This guard is also a clock constraint and the transition can only be taken if the clock satisfies the guard. The combination of these two ensures that the transition is triggered after a fixed amount of time. Furthermore there is an output or input for the transition and a reset for the timer which in a formal way resets the timer so it can be triggered again. A transition may be labeled by a number of timer resets. The timed I/O-automata also have an initial state marked with an incoming arrow and the initial values for the timers will be zero.

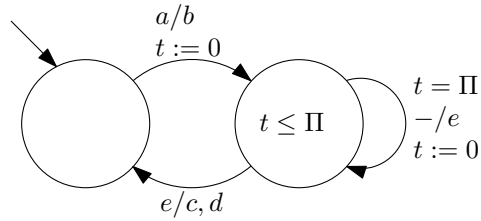


Figure 3.2: Example of a timed-I/O automaton

Chapter 4

Logdata from BOS

4.1 Introduction

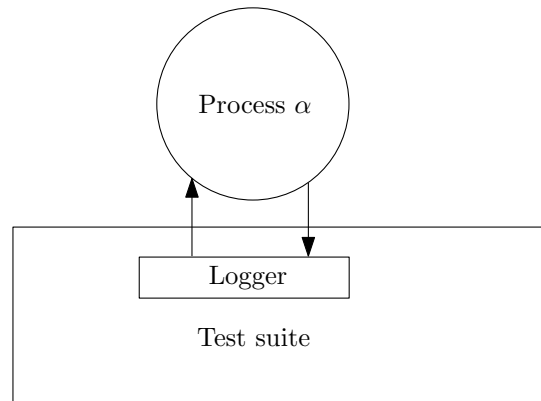
The subsystems from BOS can run in two different environments. The first type of environment is a test suite in which a single process of a subsystem can be tested, see Figure 4.1(a). In this situation all communication from or to a certain process, α in this example, goes to respectively comes from the test suite. All this communication is logged in one file and the test suite pretends to be all other processes that α communicates with. The second type of environment is an operational mode, see Figure 4.1(b), in which the system runs normally and works as described in Chapter 2. In this case processes communicate with each other by writing in each others buffers. So if a process β sends a message to α over a channel `ch1` it will do so by writing it into the buffer `α .ch1` of α , and at the same time it will write the message to a `.out` log file, `α .ch1.out`. When α reads the message it will write it to its own log file with the `.in` extension, `α .ch1.in`, in this case the same message is logged twice. So both environments will generate their own type of log files and in sections 4.2 and 4.3 the differences and similarities of both types will be discussed and a method to parse them to one format will be given.

The log files from the operational mode contain much information about common behavior and timing aspects of the system, the only point of discussion is what part of the spurious behavior can be found back in these log files. Since the system is a safety critical system that only performs its safety critical behavior incidentally, parts of the spurious behavior of the system may not be contained. For this reason it is very helpful to have the log files from the test cases since these contain much more of the spurious behavior.

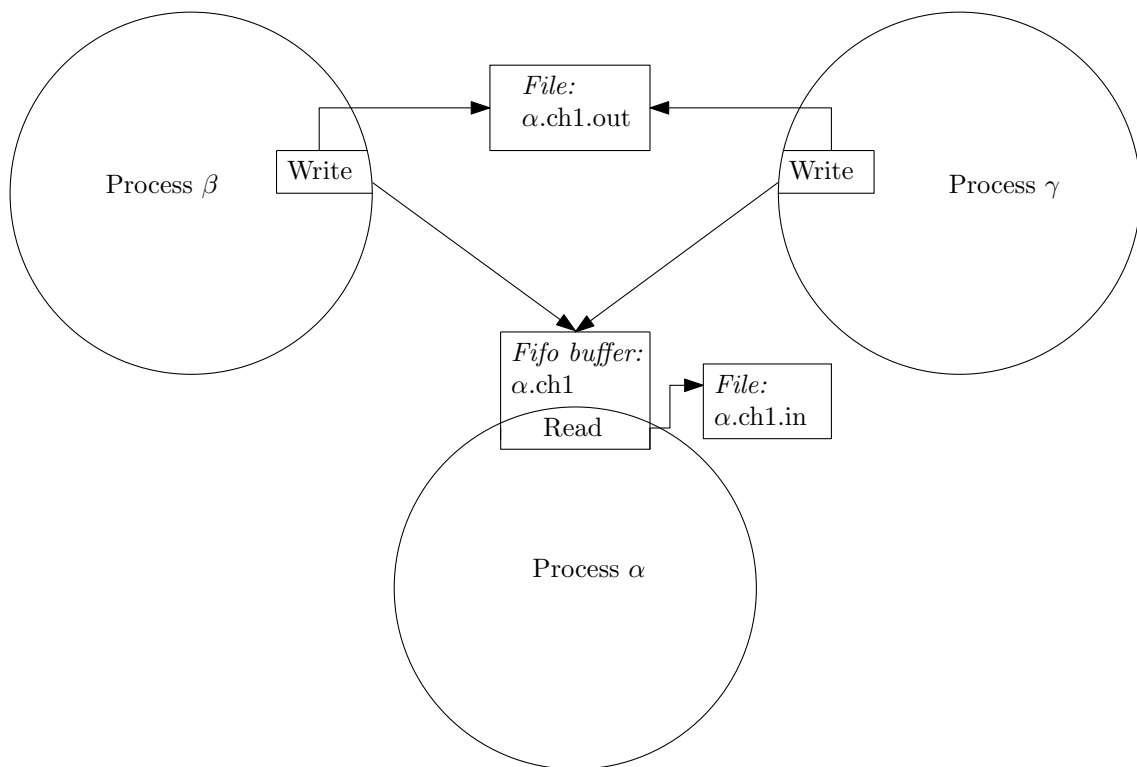
The completeness or incompleteness of traces in the log files compared with the possible traces in the system will be impossible to detect. Since this has influence on the model that can be extracted from these traces, it is good to notice that the model should at least be able to generate the traces that were in the log file. If there are messages that are possible in the system, but do not appear in the log files at all they will not appear in the model of course.

4.2 Test cases

The first type of log files are the log files created from the test suite. When the process of interest runs within the test environment a single log file is created for this process that logs all communication involved with the process under test. The messages are ordered on their occurrence, so messages that are sent first are in front of messages that are sent later, note



(a) Process α in a test environment.



(b) Process α in an operational mode.

Figure 4.1: The two different environments the system can run.

```

01:35:16(CHKPNT) TC_0
01:35:20: GUpDISDispatcher.BOcInputQueue <= CGUsTrigger [ gutHARTELKANAALSTATUS ]
01:35:21: IBpPMAProcesManager.IBcPMAResp <= CIBsPMAResp [ 1 ipsGEINITIALISEERD 0 ]
01:35:21: IBpPMAProcesManager.IBcPMACmnd => CIBsPMACmnd [ ibcSTART ]
01:35:21: IBpPMAProcesManager.IBcPMAResp <= CIBsPMAResp [ 1 ipsOPGESTART 1 ]

01:35:21(CHKPNT) TC_1
01:37:01: KEpHKIInterface_1.BOcInputQueue <= CKEsHKBStuurOpdracht [ hkcEINDEVOORBEREIDING ]
01:37:01: IBpMRGMeldingsRegistratie.IBcMeldingen <= CIBsMeldingen [ 1 3 (HKB)42002
bscHARTELKERING 1184801821 21 (hkcEINDEVOORBEREIDING) ]
01:37:01: KEpHKIInterface_1.KEcHKIRespons => CKEsHKIGelukt [ 1003 hkcEINDEVOORBEREIDING
vbsVOORBER_KERING_AFGEROND hksHARTELKERING.GEOPEND sssSCHEEPVAARTSIGNALERING.GEACT 4g
3g ]

```

Figure 4.2: Example of a part of a test suite log file.

that this means that incoming and outgoing messages are mixed, i.e. they both occur in one log file.

Figure 4.2 gives an example of a part of such a log file and it shows that a log file can consist of three different types of lines, an empty line, a comment or synchronization line, that looks like "01:35:21(CHKPNT) TC_1", and a line that contains a message. The first two options are not interesting, because they do not have any value for the purpose of creating a model. But if a line contains a message then it contains useful information about what the process does. Such a message is built up from the following elements:

- First it has a time stamp, which is the time the message is logged.
- The time stamp is followed by a ":" .
- Then it has a process name, which is the other process involved (so not the process under test), see Section 2.2.
- The process name is followed by a "." .
- The dot is followed by a channel name, which indicates over which channel the message is sent.
- Then a direction of the message is given, i.e. "=>" indicates an incoming message for the process under test and "<=" indicates an outgoing message for the process under test.
- After the direction comes the message type, this says what kind of message is sent.
- And finally the message contains content which is enclosed in brackets "[" and "]" .

Since the log files from the systems under test are all built up in the same way as shown in Figure 4.2 a grammar can be given that specifies the structure of the files. This grammar looks as follows:

```

<LogFile> ::= <Line><LogFile>| ε
<Line> ::= "eol"|<Comment>"eol"|<Message>"eol"
<Comment> ::= <TimeStamp>"(CHKPNT)␣TC_"<Number>
<Message> ::= <TimeStamp>":␣"<Process>".␣"<Channel>"␣"
               <Direction>"␣"<MessageType>"␣[␣"<Content>"]"
<Content> ::= String"␣"<Content>|<Number>"␣"<Content>| ε
<TimeStamp> ::= <Digit><Digit>":":<Digit><Digit>":":<Digit><Digit>
<Process> ::= String
<Channel> ::= String
<Direction> ::= "<="|">"
<MessageType> ::= String
<Number> ::= <Digit><Number>|ε
<Digit> ::= "1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"|"0"

```

Where "eol" is the end of a line, meaning this is an empty line and *String* is a sequence of symbols which can be all characters, capitals, digits and some special symbols, "_ () - : \".

4.3 Operational mode

When the system runs in operational mode it is also possible to create log data from it, but now it keeps for each process and channel combination that exists a separate '.in' and '.out' log file, e.g. "hkb.BOCInputQueue.in" and "hkb.BOCInputQueue.out". The name of the file gives some of the needed information, namely the part before the first dot gives the process that owns the channel, the part between the dots is the name of the channel and the part after the last dot (in or out) says if the messages are input or output. A process under interest writes each incoming message in its own '.in' file and writes each outgoing message in the '.out' file of the process the message is sent too, see Figure 4.1(b). So to collect all the messages concerning one process it is needed to get all the messages in the '.in' files of which the file name starts with the name of that process and to check all the '.out' files for messages that were sent from the process, see Figure 4.3. By taking the messages from the '.in' files as incoming messages, the messages from the '.out' files as outgoing messages then by parsing the messages the same structure as for the test suite logging can be derived. Note that no information is lost in this parsing, since all information from the file name is used in the new format too, the process name as the other process involved, the channel name as the channel the message was sent over and the ".in" and ".out" as the direction.

Figure 4.4 shows an example of an operational mode log file, in this case a '.in' file. The log files from the operational mode are unlike the test case logs built from one type of line. Such a line, as shown in Figure 4.4, is built up from the following elements:

- It starts with a date, which is the date, according to the systems clock, at which the message is logged.
- Then it has a time stamp, which is the time the message is logged.
- The time stamp is followed by a process name which is the process that sent the message.
- Then there is some kind of sequence number.
- This number is followed by the message type, which says what kind of message is sent.

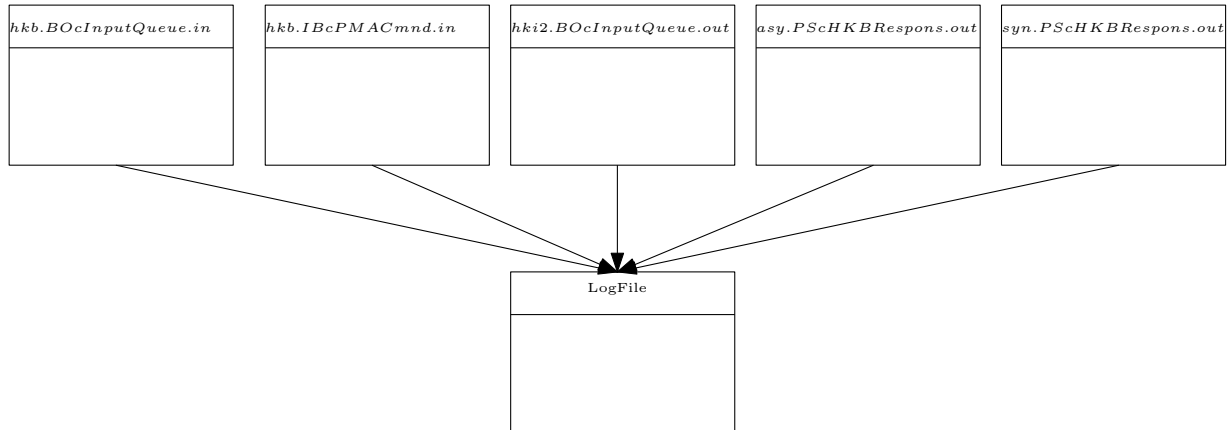


Figure 4.3: Graphical visualization of constructing a log file.

```

76-01-07 03:20:01 KEpHKBBestHartelkering 25 CKEsHKIGelukt [189829201 hkcBESSTATUS
  vbsVOORBER.KERING.OPGEHEVEN hksHARTELKERING.GEOPEND sssSCHEEPVAARTSIGNALERING.GEDEEA 2000g
  2000g ]
76-01-07 03:20:25 KEpHKBBestHartelkering 26 CKEsHKBWijzigAutor [hkaGEENCOMMUNICATIE ]
76-01-07 03:30:07 KEpHKBBestHartelkering 31 CKEsHKBWijzigAutor [hkaBLOKKERENTOEGESTAAN ]

```

Figure 4.4: Example of a part of an operational mode log file.

- And finally the message contains content which is enclosed in brackets like "[" and "]".

Also for the operational mode log files a clear grammar can be given of which string and number are the same as for the test case log files, see Section 4.2, which looks as follows:

$$\begin{aligned}
 \langle OMLogFile \rangle & ::= \langle OMMessage \rangle \langle OMLogFile \rangle | \epsilon \\
 \langle OMMessage \rangle & ::= \langle Date \rangle " " \langle TimeStamp \rangle " " \langle Process \rangle " " \langle SequenceNumber \rangle " " \\
 & \quad \langle MessageType \rangle " " [" \langle Content \rangle "] eol \\
 \langle Date \rangle & ::= DigitDigit "-" DigitDigit "-" DigitDigit \\
 \langle SequenceNumber \rangle & ::= Number
 \end{aligned}$$

With this grammar the process of transforming messages from the '.in' and '.out' files to a file which has the same grammar as the test suite log files can be defined as follows:

For a certain process α all '.in' files that start with the name α are opened and all '.out' files that do not start with the name α are opened too. Then by a technique that looks like the one that is used in merge sort message are taken based on their time stamps, so the one that occurs first is taken first, and put in a new file. For the '.in' files all messages are taken and for the '.out' files a filter is used on the process that is in each message, which should equal α . The new message line for a message from a '.in' file is now built up in the following way:

$\langle TimeStamp \rangle " " : " \langle Process \rangle " . " Channel " " \Rightarrow " \langle MessageType \rangle " " [" \langle Content \rangle "] eol "$,

where Channel comes from the file name of the '.in' file between the dots. Messages from the '.out' file where $\langle Process \rangle$ equals α are transformed to a new message line as follows:

$\langle TimeStamp \rangle " " : " ProcessName " . " Channel " " \Leftarrow " \langle MessageType \rangle " " [" \langle Content \rangle "] eol "$,

where ProcessName and Channel come from the file name, ProcessName is the part before the first dot and Channel is again the part between the dots. Doing this for all messages in the files a new log file is constructed that has the same structure and grammar as the log files from the test suite and the advantage that all communication, incoming and outgoing,

for one process is contained in one log file.

4.4 Generalization

For log files in a more generalized form it is important to say what kind of log data is needed to apply the method described in the following chapters. As said in Chapter 1 within this research project log data from inter process communication, communication between the processes, is considered. The method described in this thesis is based on the assumption that the log data consists of this communication. So there is some general information that should be logged in the log file to be able to apply this method.

First the “data” that is sent from one process to another should be logged. With data not only the message type is meant but also the content if there is any, note that in many situations it is the choice of the developer of a system to see some part of the data as content or as a different message type. E.g. the messages “*StartTimer*” and “*StopTimer*” are in some way equivalent to “*Timer[stop]*” and “*Timer[start]*”, while in the first case for every action a separate message type is constructed and in the second case there is other content.

Besides the data the name of the process that sends the data and the process that receives it should be logged and it should be made clear which process is sending and which is receiving the data. Finally there should be a time stamp or something that describes the moment the message is sent. This time stamp does not necessarily has to be the real time, but can e.g. also be the amount of time passed since the start of the system.

Chapter 5

Analysis

5.1 Time

The log files as described in Chapter 4 contain a flow of events that happened over time. All these events have a timestamp that describes the moment that the event happened. To be able to receive a higher level of abstraction and to add more predictive value to the model it is needed to abstract from the precise moments in time that is logged in the log file. To determine in example time driven events, as done in Chapter 6, just the time differences can be used.

Another reason to abstract from the precise moments in time that an event occurs is that it is a reactive or open system. This means that it can continuously receive inputs during the operation of the system. Which means that events are not depending on the precise moment in time that they were sent, but on the amount of time that has elapsed since the last (same) event or on the last input event that is given to the system.

5.2 Distinguishing messages

To be able to create a model from the log data it is needed to be able to say if two random messages from a log file are the same or different. This is important because if it is possible to say that two messages are the same, a higher level of abstraction can be reached and with that more predictable value is added to the model. To be able to distinguish messages it is important to define when they are the same and when they are different, in this section it is explained how messages are distinguished and where points of discussion are.

Since a message exists of a time stamp, a process name, a channel name, a direction, a message type and some content, there are quite a few different possibilities to say that a message differs from another message and there is no general correct answer. Some of the elements that seem logical choices to be taken into account are the process name, the channel name, the direction and the message type. These seem logical choices for the following reasons:

- The message type is the part of the message that should be taken into account most certainly to be able to distinguish different messages. The message type is the main part of the message and describes the message itself. For this reason there is no doubt that this should be taken into account.
- The process name seems logical since it describes the other process involved for this message. When creating a model of the system it seems important to be able to detect

if the same message is sent to more than one process and for this reason it will be taken into account when distinguishing messages.

- For the channel name more or less the same holds as for the process name. Although it might seem strange that a message of the same message type is sent to the same process over a different channel, it seems important to be able to notice this if it happens, since it would mean that the receiving process would have at least two channels to communicate the same data over with the other process. The only way to be able to notice this is if the messages can be distinguished from other messages with the same message type and process name, so that is why the channel name will be taken into account for message distinguishing.
- When it is possible to distinguish messages by message type, process name and channel name it is questionable how much value is added by taking the direction into account to distinguish messages. Of course the direction is used to describe timing behavior in Chapter 6 and spurious behavior of the system in Chapter 7, but it is not sure that the direction is needed to distinguish messages. On the other hand if all the other parameters are the same, e.g. when a process echoes the same message back over the same channel to the same process, it will be useful to detect this, what only can be done by taking the direction into account.

Besides these elements there are two elements left of a message in a log file that are not discussed yet, namely the time stamp and the content of a message. The time stamp will be used to extract properties about time driven messages from the log file, as we will see in Chapter 6. It gives rise to the ordering of the messages in the log file, is useless to distinguish messages with since in that case almost all messages will be different, see Section 5.1. For this reason they will not be taken into account to distinguish messages.

The other element that was not discussed until now is the content of a message. The content of the message has a different number of elements for each message type, which can also be zero. These content elements are specified per message type, but do not have a general structure. E.g. some elements are unix time stamps, ten digits, others can be water heights, also digits, and again others can be names of elements of enumerations, which are strings. This means that it will be hard to say in general and without (human) knowledge of the system what the elements of the content mean and whether they are of importance to take into account to distinguish messages.

5.3 Time stamps

The time stamps of messages from the log files are, as we will see in Chapters 6 and 7, used in different places within the process of extracting a model. For this reason it is important to devote a short word on the accuracy of these time stamps. When taking a closer look into the time stamps it can quickly be noticed that they exist of hours, minutes and seconds. This means that the time stamps are rounded to seconds, note that it is not known in which way this is done and that it is also not important to know how this is done. For the goal of extracting a model from the log files it is just needed to take into account that there may be differences caused by this rounding.

Also it is possible that there is a small difference in when a message is sent and when it is logged. This can happen for different reasons, e.g. it can be caused by delays on the network

or in the logging unit. Since it is not known how large such delays can be, if they exist at all, and also they cannot be distinguished from the rounding differences meant earlier, it will be necessary to take these into account at once.

Chapter 6

Timing

6.1 Introduction

As described in Chapter 4, log files from the operational mode can get large, e.g. contain more than 20.000 lines. To shrink the size of the log files the following hypothesis is used:

Since the BOS is a safety critical system that only performs its safety critical behavior incidentally, much of the behavior of the system will be time depending. If this is the case, detecting time driven messages as a first step in creating a model from the system will decrease the size of the log files significantly.

Time driven messages are messages that are not generated as a reaction to any other message that is sent to or from the process, but that are sent as result of the expiration of a timer. One of the properties of a time driven message will be that the time intervals between two sequential messages of the same type, as discussed in Section 5.2, will be the same for all time intervals, except for a small fluctuation caused by inaccuracy of the time stamp, see Section 5.3.

It is important to detect these time driven messages at the beginning of the process of extracting a model from the log data, since they have great influence on the “complexity” of the model that can be extracted for the process under interest. This has two reasons. The first reason is that, as just mentioned, BOS only performs its critical behavior incidentally. This means that the log files will almost completely consist of time driven messages in case there is no reason for its critical behavior. So extraction of the time driven messages results in smaller log files. Of course there can be some processes that have no time driven messages, incoming or outgoing, involved at all. In that case the log files will be almost or completely empty in case there is no reason to perform the mentioned critical behavior. The second reason why detecting the time driven messages first has great influence on the complexity of the model is because the assumption is made that time driven messages can be sent independent of the other behavior of the process. E.g. if there is a timer in an external process that sends messages to check a certain property, it will keep doing so regardless of the state of the receiving process. The same holds for a timer in the process under interest that checks a property or a message like a heartbeat signal that just notifies that the process is still running normally. A timer that is behind such a signal keeps sending the same messages after a predefined amount of time, the expiration time of the timer, regardless of the messages that other processes sent to the process. Leaving these time driven messages in the log files, would

lead to the same messages being triggered in all different states and would have influence to the input/output relations of messages, as described in the beginning of Section 7.2.

Like in many cases there are exceptions as shown in Section 6.3. It is possible that some kind of message enables/disables the time driven messages, for example by starting and stopping the timer of a process. Figure 6.1 shows how this looks when the occurrence of a message is compared with the time. The two dotted lines at the bottom of the graph are the occurrences of regular time driven messages against time. The two dotted lines above these show a time driven message with breaks. If this starting and stopping of a time driven message occurs it is important to detect the message that causes this to happen, since this clearly indicates a state change in the system.



Figure 6.1: Message occurrence compared with the time, for each message type.

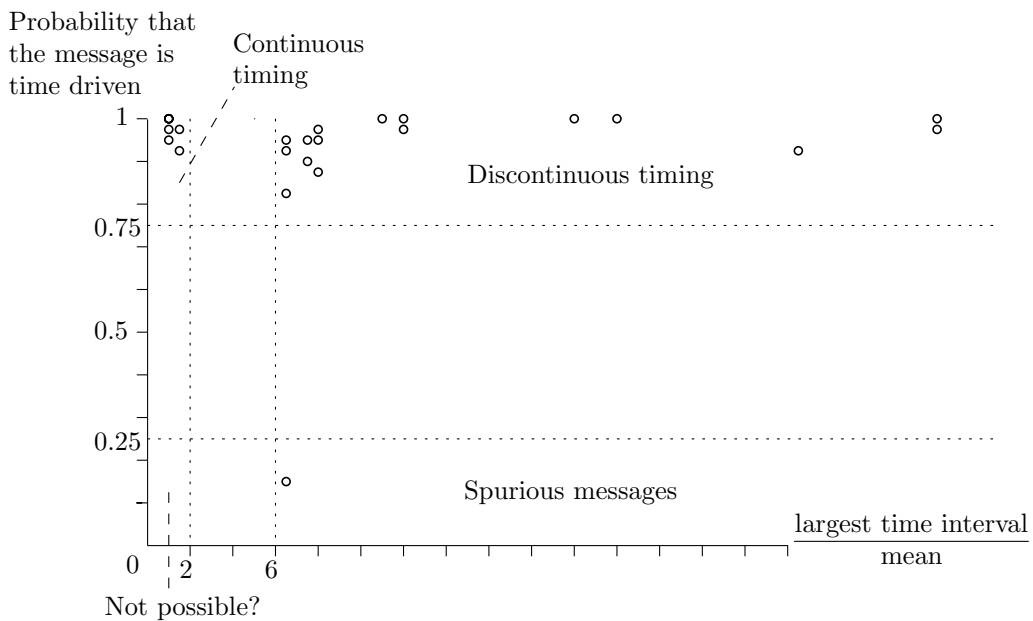


Figure 6.2: Classification of message types.

Figure 6.2 shows the different options that exist for messages with respect to being time driven. The y-axis shows the probability that a message is time driven and the x-axis shows the largest time interval divided by the mean value of the time intervals. How these values are determined will be discussed in the next two sections. To be able to determine these values and get conclusions from these, a sufficient number of messages, i.e. ten or more, of the same type should be available. This is not a fixed number, but ten seems to be a good choice. This is not a problem since if a message is time driven there will be enough messages available and if this is not the case the messages will be treated as spurious messages. Here a rough overview of how the messages can be split into groups is given:

Continuous timing If the largest time interval is equal or at most two or three times larger than the mean value of these intervals and the probability of being time driven is high, the message will probably be continuous time driven. How these messages can be found in a log file will be discussed in Section 6.2.

Discontinuous timing When the largest time interval is more than two or three times larger than the mean value of the intervals and the probability of being time driven is also high, this means that probably the timer has been turned off for a while before it continued again. The stopping of the timer causes a large time interval, while the mean value of the time intervals is just a bit larger. How these discontinuous time driven messages can be found will be discussed further in Section 6.3.

Spurious messages If the probability of being time driven is low or the number of messages is very small, i.e. less than ten, this indicates that the message is a spurious message, probably caused by other behavior of the system which is not depending on timers. These are the messages that are left after the time driven messages, continuous and discontinuous, are filtered from the log file.

Other The last possibility of a small largest time interval divided by the mean value of the time intervals and a low probability of being time driven probably will not happen and if it should happen these messages will be treated as spurious messages.

6.2 Continuous timing

Continuous time driven messages can be detected by comparing all time intervals that exist between two consecutive messages of the same type. Messages are of the same type if they are following the description given in Section 5.2. This will be denoted as $m \sim m'$, meaning that m and m' are of the same type.

A log file consists, as described in Chapter 4, of a list of pairs of time stamps and messages. For this reason a log file L can be seen as a list of pairs $[T \times M]$, where T are time stamps and M are messages, so $L := [T \times M]$. From this a list of messages of the same type can be filtered by a function *filter*:

$$\begin{aligned}
& \text{filter} : M \times L \rightarrow L \\
& \text{filter}(m, []) = [] \\
& \text{filter}(m, (t, m') \triangleright xs) = \begin{array}{l} \text{if } m \sim m' \rightarrow (t, m') \triangleright \text{filter}(m, xs) \\ \quad \square \quad m \not\sim m' \rightarrow \text{filter}(m, xs) \\ \text{fi} \end{array}
\end{aligned}$$

So if a list of messages of the same type is available, it can be checked if the messages in the filtered list are time driven by comparing all the time intervals of two consecutive messages in the filtered list. In the ideal case all time intervals of a continuous time driven message are the same. So a function *get_int* that computes a list of time intervals of all sequential messages by subtracting the time values from each other, results in a list of the same values with a size that is one smaller than the filtered list, except if the filtered list is empty. A function that can do this will look as follows:

$$\begin{aligned}
& \text{get_int} : L \rightarrow [T] \\
& \text{get_int}([]) = [] \\
& \text{get_int}((t, m) \triangleright []) = [] \\
& \text{get_int}((t, m) \triangleright (t', m') \triangleright xs) = (t' - t) \triangleright \text{get_int}((t', m') \triangleright xs)
\end{aligned}$$

If the ideal case is true and a filtered list of messages of one type is time driven, all the intervals between two sequential messages are the same. So the function *get_int* will return the same value for all pairs of sequential messages in the filtered list. If this is the case a function *count_int* selects interval values and the function *counti* counts how many times a selected interval value occurs. The function *skip* ensures that every interval value is taken into account just once, by filtering out all other occurrences of that value after they are counted and the function *moi* selects the most occurring interval value by returning a pair of the interval value and the number of times it occurs. These functions are given below:

$$\begin{aligned}
& \text{count_int} : [T] \rightarrow [T \times \text{Int}] \\
& \text{count_int}([]) = [] \\
& \text{count_int}(t \triangleright xs) = (t, \text{counti}(t, t \triangleright xs)) \triangleright \text{count_int}(\text{skip}(t, xs))
\end{aligned}$$

$$\begin{aligned}
& \text{counti} : T \times [T] \rightarrow \text{Int} \\
& \text{counti}(t, []) = 0 \\
& \text{counti}(t, t' \triangleright xs) = \begin{array}{l} \text{if } t = t' \rightarrow 1 + \text{counti}(t, xs) \\ \quad \square \quad t \neq t' \rightarrow \text{counti}(t, xs) \\ \text{fi} \end{array}
\end{aligned}$$

$$\begin{aligned}
& \text{skip} : T \times [T] \rightarrow [T] \\
& \text{skip}(t, []) = [] \\
& \text{skip}(t, t' \triangleright xs) = \begin{array}{l} \text{if } t = t' \rightarrow \text{skip}(t, xs) \\ \quad \square \quad t \neq t' \rightarrow t' \triangleright \text{skip}(t, xs) \\ \text{fi} \end{array}
\end{aligned}$$

$$\begin{aligned}
\text{moi} &: T \times \text{Int} \times [T \times \text{Int}] \rightarrow T \times \text{Int} \\
\text{moi}(t, i, []) &= (t, i) \\
\text{moi}(t, i, (t', i') \triangleright xs) &= \begin{array}{l} \text{if } i \geq i' \rightarrow \text{moi}(t, i, xs) \\ \quad \square \quad i < i' \rightarrow \text{moi}(t', i', xs) \\ \text{fi} \end{array}
\end{aligned}$$

From the *moi* function it is also possible to see how many times that interval occurs and if the ideal case is true and the message is time driven this number should equal the number of messages in the filtered list minus one. If this is the case the time interval t can be used as the period Π of the time driven message.

The ideal case is not very likely to occur, since even if a message m is continuous time driven, there will be small fluctuations in the time intervals. In example, if for a certain time driven message $\Pi = 60$ seconds it might happen that some of the messages are logged with a time interval of 59 seconds or with time intervals of 61 seconds. In this example there can be a deviation of 1 second, but it is very well possible that this can be more. To take fluctuation into account and still be able to determine that a list of messages is time driven, it is needed to take the sum over a small range of values of time intervals instead of looking at a single value. This range can be described as the interval $[t - \delta, t + \delta]$, see Figure 6.3, where δ is the deviation that occurs and t the interval value under interest. So instead of determining the most occurring interval after counting how many times each interval occurs, first the number of intervals that belongs to each possible range is counted using the *count_range* and *countr* functions below. When this is done the range in which the most intervals occur can be determined via applying the *moi* function on the result of the *count_range* function.

$$\begin{aligned}
\text{count_range} &: T \times [T \times \text{Int}] \rightarrow [T \times \text{Int}] \\
\text{count_range}(\delta, []) &= [] \\
\text{count_range}(\delta, (t, i) \triangleright xs) &= (t, \text{countr}(\delta, t, (t, i) \triangleright xs)) \triangleright \text{count_range}(\delta, xs)
\end{aligned}$$

$$\begin{aligned}
\text{countr} &: T \times T \times [T \times \text{Int}] \rightarrow \text{Int} \\
\text{countr}(\delta, t, []) &= 0 \\
\text{countr}(\delta, t, (t', i) \triangleright xs) &= \begin{array}{l} \text{if } t - \delta \leq t' \leq t + \delta \rightarrow i + \text{countr}(\delta, t, xs) \\ \quad \square \quad t - \delta > t' \rightarrow \text{countr}(\delta, t, xs) \\ \quad \square \quad t + \delta < t' \rightarrow \text{countr}(\delta, t, xs) \\ \text{fi} \end{array}
\end{aligned}$$

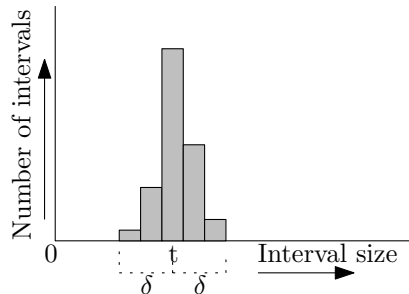


Figure 6.3: Example of the range $[t - \delta, t + \delta]$.

Because every interval length will be counted more than once, the number of intervals of

size $t = 60$ will be counted for all interval values between $60 - \delta$ and $60 + \delta$, it is needed to take the maximum range, the t for which the range $[t - \delta, t + \delta]$ contains the most intervals. For a continuous time driven message this will be the range where all intervals are contained in. The function that can compute the maximum range is the *moi* function again and the resulting t can be chosen as the period for the timer.

Another reason why the ideal case is not always true is that it might happen that a line is missed in the logging process and then one of the time intervals gets twice as large, or three times if two lines are missed, compared with the others. Although this seems to point at discontinuous time driven events, as will be discussed in Section 6.3, this would be a strange conclusion if 1 out of 1000 time intervals is twice as big as the others. To solve this problem the probability that a certain message type is time driven is considered in a quite straight forward way. The probability, P , that a certain message is time driven is calculated by taking the number of intervals that occur in the maximum range, selected by the *moi* function on the output of the *count_range* function, and divide it by the number of time intervals that exist for the message m , which is the size of the filtered list minus one, in formula:

$$\begin{aligned}
 P : T \times Int \times L &\rightarrow Real \\
 P(t, i, []) &= 0 \\
 P(t, i, l \triangleright xs) &= i / (length(l \triangleright xs))
 \end{aligned}$$

This probability only gives a chance that a message is time driven, but it does not say anything about the message being continuous or discontinuous time driven as shown in Figure 6.2, where this probability forms the y-axis of the figure. To solve this also the maximum and the mean value of the time intervals are taken into account. If just two or three messages are missing in the log file and the messages is continuous time driven, dividing the maximum time interval by the mean of the time intervals gives a factor ω that is smaller than 3. Note that the mean value will get a bit larger too if messages are missing, but for deciding if a message is continuous or discontinuous time driven this does not matter since the number of messages should be sufficient to be able to decide if a message is time driven, meaning that the mean value will not become too much larger. Also it will be most likely the case that there is a big gap in this factor between continuous time driven messages and discontinuous time driven messages. Since if a timer really is turned off, it is probably not turned off for just two or three times the mean value, but for more than 20 times the mean value as shown in Section 6.3. Formally this factor ω can be calculated by taking the maximum over the list of time intervals, and divide by the mean value of the same list. In formula form:

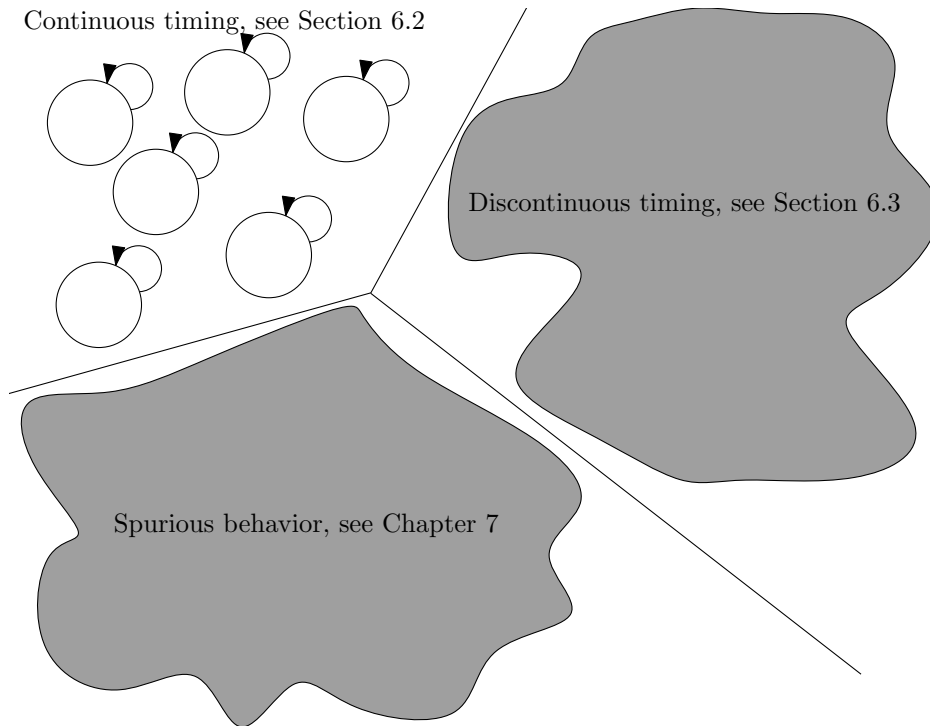


Figure 6.5: Part of the model that can be constructed after continuous timing.

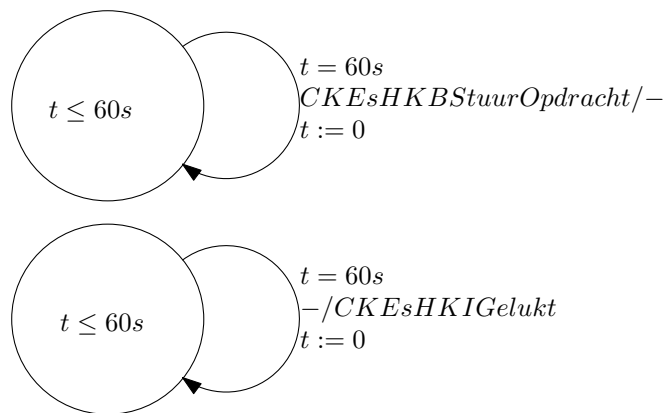


Figure 6.6: The continuous time driven part for one process from the BOS system.

6.3 Discontinuous timing

In a real-world system as the BOS is, it is very well possible that there exist timers that are switched on and off from time to time. In example for BOS it could be that there is a timer that checks weather conditions only when the sea level is above a certain level or a timer that is only turned on when a certain action is in progress, like closing the barriers, to check the status of them. In case this happens it is important that this can be detected, since this kind of timers will give a different model then continuous timers.

In Section 6.2 it is explained how the probability can be calculated that a message is time

driven. For discontinuous time driven messages, which are time driven messages for which the timer can be turned off for a while, this calculation still holds. Since only a small part of the time intervals will be large and most of them will be around the period of the timer, the probability that it is a time driven message as defined in Section 6.2 is high for discontinuous time driven messages. Also the factor ω , as explained in Section 6.2, can be computed, but since for discontinuous time driven messages the maximum time interval is much larger than the mean value of the time intervals, ω will be much larger than 3 as given for continuous time driven messages. So the probability P gives a probability that a messages is time driven and the factor ω ensures that the difference between continuous time driven messages and discontinuous time driven messages can be detected.

Now, for the discontinuous time driven messages it is not only important to be able to detect that it is a discontinuous time driven message, but for the purpose of creating a model it is desirable to detect the message that enables or disables the timer. Since the system is an event driven system this message should exist, otherwise a process would decide on its own to enable or disable a timer. To detect the message that enables and disables the timer it is not only important to be able to decide that the message is discontinuous time driven, but it is also needed to be able to split the list of messages into continuous time driven blocks, see Figure 6.7. This means that the goal is to split the list of filtered messages into blocks, of which the time intervals are like the ones of continuous time driven messages as defined in Section 6.2. This can be done by a divide and conquer approach by splitting on the maximum time interval until the factor ω is, as for continuous time driven messages, smaller than 3. The formula that splits the list of filtered messages on the maximum time interval t is the following:

$$\begin{aligned}
 split : T \times L &\rightarrow L \times L \\
 split(t, []) &= ([], []) \\
 split(t, (t', m') \triangleright []) &= ([(t', m')], []) \\
 split(t, (t', m') \triangleright (t'', m'') \triangleright xs) &= \begin{cases} [(t', m')], (t'', m'') \triangleright xs & \text{if } t = (t'' - t') \\ (t', m') \triangleright Lx, Ly & \text{if } t \neq (t'' - t') \end{cases}
 \end{aligned}$$

where

$$(Lx, Ly) = split(t, (t'', m'') \triangleright xs)$$

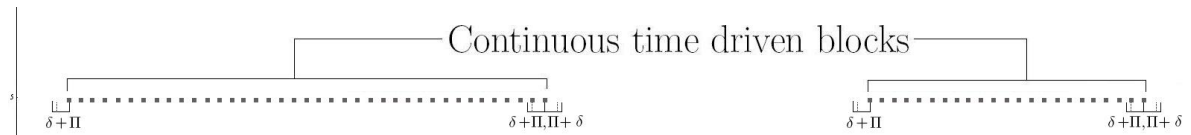


Figure 6.7: Example of continuous time driven blocks.

When the discontinuous time driven messages are split into continuous time driven blocks, it is possible to determine the set of messages that is one period $\Pi + \delta$, see Figure 6.7, before the first message of the block. In this case Π is the period of the timer and δ is the same delta as used to determine if a message is time drive. The δ is used for the same reason as earlier, namely there might be a fluctuation in the logging time. If there are enough of these continuous time driven blocks, for a certain type of discontinuous time driven message, it would be possible to take the intersection over the sets of messages that are in front of the

first message of the blocks, then take out all the messages that are continuous time driven, since these can not influence the timer because they occur through the complete log file, and in the ideal case one type of message is left that triggers the starting of the discontinuous time driven message. Since the ideal case might not occur, it is possible that one of the messages that triggers the timer is missing in the log file and an empty set is the result, it is better to count in how many sets each type of message occurs and then take the message with the maximum occurrences as the start trigger for the discontinuous time driven message, provided that this maximum occurs in at least ninety percent of the blocks. For the message that causes the discontinuous message to stop more or less the same can be done, but instead of taking only the messages that occur in a period of $\Pi + \delta$ after the last message of a block, also the messages between the second last and last message are taken into account, see Figure 6.7. This is done because a timer can be stopped in two ways, namely by stopping the timer directly or by saying that it may not start again after it runs down. Both options should be dealt with, since it is not clear from the log file how the timers in the system under study work. Note that for this method it is assumed that the period of the timer is the same for all continuous time driven blocks, it might be reasonable that in some cases these periods differ, but this will be part of the future work.

The method just described only works if there are enough of these blocks. Since for this research project the size of the log data was not sufficient to provide enough blocks, to be able to handle discontinuous timers that are put on and off very rarely it is possible to add a start and stop message. This means that the first message of a block gets substituted by a start message and the last message by a stop message. The name of the message can be chosen, e.g. “Start” and “Stop” followed by the message type would be enough. These messages are added to be able to link the discontinuous time driven messages to the spurious behavioral model of the system. In case the start and stop messages could be found, these message would form this link. Note that by adding this message, the real start and stop message does not disappear, meaning that a certain message gets duplicated with another name, it is just not known which message it is.

So a discontinuous time driven message can be detected by checking if the probability of the message being time driven is high and the factor ω is high too. When this is the case there is a need to detect the message that starts and stops the timer. If this is not possible such a message can be added. For the period of the timer the value of Π can be used just as for the continuous time driven messages in Section 6.2 and then a model template would look like as in Figure 6.8. Where the start and stop messages are the state changes between two states of which one state has a loop that is triggered when the timer runs out after each period Π .

Figure 6.9 shows the part of the model that can be constructed till here. The continuous timing has been explained in the previous section and the discontinuous timing is explained in this section. Spurious behavior will come back in the next chapter.

Figure 6.10 shows the results of this section for one of the processes in the BOS system. Again these results were correct according to the system expert, with again the note that the “CKEsKEROpdrachtStatus” message was a reply on the “CKEsHKBVraagOpdrStatus” message.

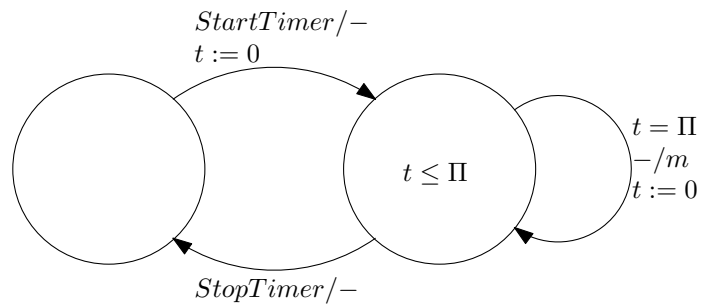


Figure 6.8: Example of a discontinuous time driven message m with period Π .

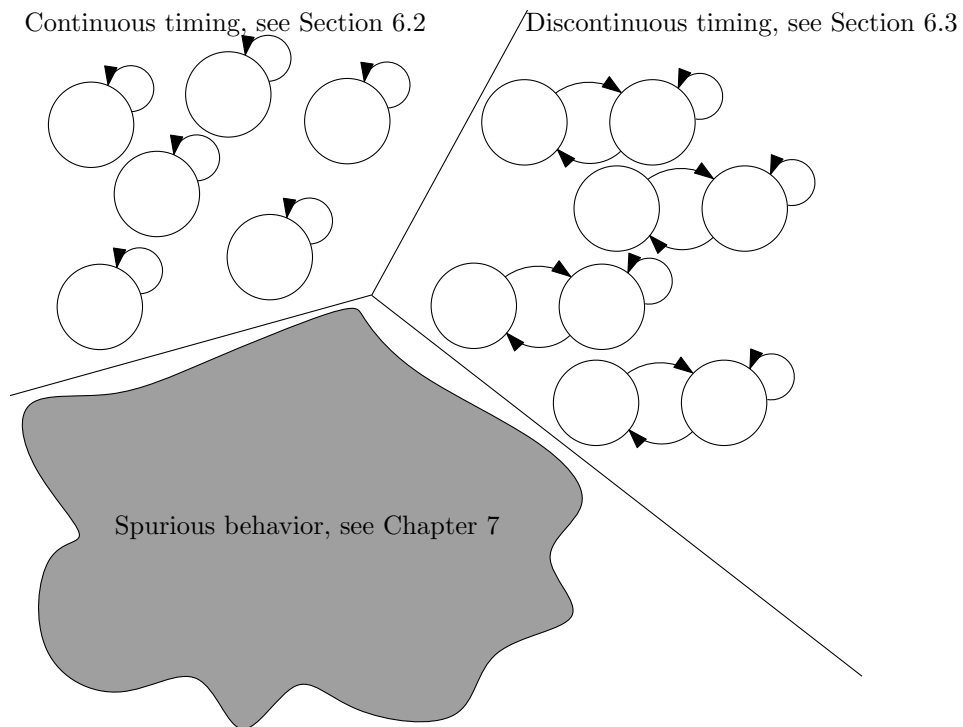


Figure 6.9: Part of the model that can be constructed after continuous timing.

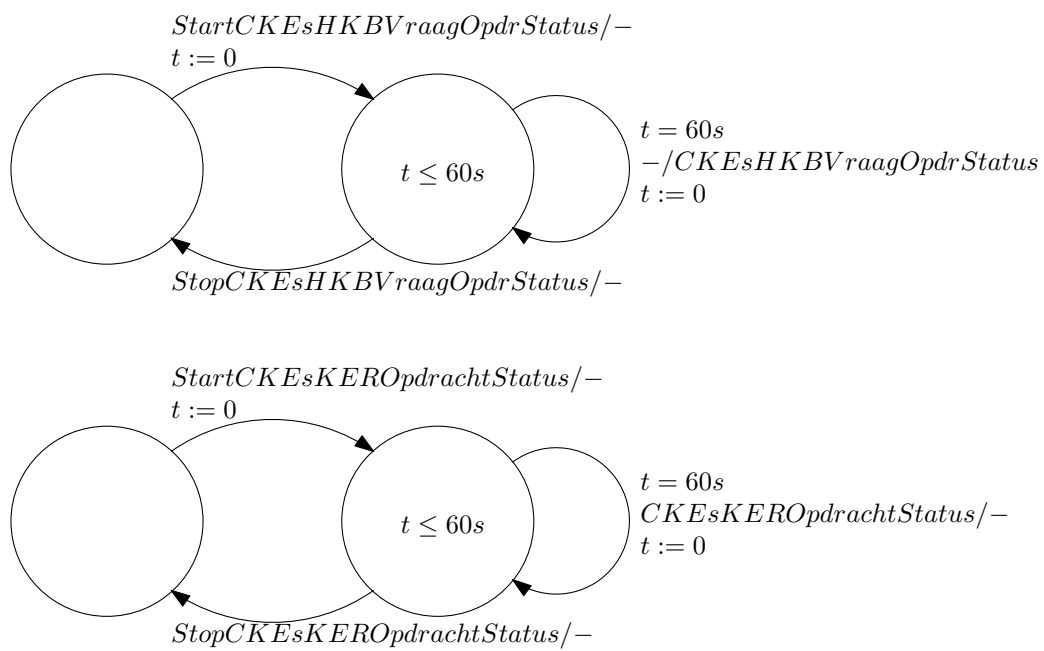


Figure 6.10: The discontinuous time driven part for one process from the BOS system

Chapter 7

Spurious behavior

7.1 Introduction

After filtering out the time driven messages, the spurious messages, and possibly the added messages for starting and stopping the timer as described in Section 6.3, are left over in the log file. These spurious messages describe the behavior of the system that is not time driven. For BOS most of the spurious behavior will be the safety critical behavior that is only executed incidentally, e.g. in extreme bad weather conditions. In this chapter a way to extract a model describing this spurious behavior, an example of how the model can look like and the actual problems and doubts that exist with this method are given.

To complete the behavioral model of the system with respect to the log files the goal will be to find the spurious behavior and to be able to construct a model from this. This goal of finding a model of the spurious behavior can roughly be split in three subgoals, namely:

- The detection of different states and transitions between them. It is needed to be able to detect how many states the system has and how the system goes from one state to another.
- The detection of a stable state, a state in which the system keeps returning in a finite number of steps as some kind of “rest” state. This state can be seen as the start and end point of traces from the system.
- The detection of loops, transitions back to states that are already detected. This is important since it keeps the model small and increases the predictable value of the system since it makes it possible to combine different parts of traces.

Since the assumption is made that the system is an event driven system it is only possible to get output from the system after there was some input to it. This means that it is not possible to get output that is not linked to a certain input. Since it is in general not possible to know to which input a certain output is linked it is assumed that it is linked to the last input that appeared in the log file. This last assumption is reasonable since all the processes are sequential processes and since the assumption is made that when reacting to a certain input a process will not be able to handle the next input until his job is done, which is reasonable since inputs are stored in buffers.

The assumption that the system is event driven has the advantage that it is assumed that all communication takes place via the message (the events) and not via shared memory

or something like that. This is important since communication via shared memory can not be logged and changes within a process can not be detected. Furthermore assuming that it is event driven has the advantage as just mentioned that it is not possible to get any output that is not linked to an input, so it is not possible that the system starts spitting out messages without an input. Finally it has the advantage that each process really can be modelled separately, since the only way processes are coupled is through the events. The main disadvantage of the assumption of being event driven is that if there is communication or exchange of data due to shared variables this will be missed and can not be noticed.

7.2 Extracting a model

From the log file without time driven messages a model can be extracted under the assumption that the system is event driven and that every output is linked to the last input that preceded it. Of course it is possible that there are inputs that do not lead to an output from the process at all, or that there is more than one output to a certain input. Note that for understanding messages are simplified, content and time stamps are omitted from the presentation unless mentioned otherwise, but the order in which they appear is important. Question marks will point to incoming messages and exclamation marks will point to outgoing messages. To create a model from the messages left over in the log file, the trace is first split in input/output subtraces consisting of a single input and the following outputs till the next input. After that the following inductive approach is taken:

By starting with one state from which all possible input and output combinations are assumed to be possible, a starting point is given. Then there is only a state change to a new state when there is a different reaction to a certain input as before. In example if the input a first one or more times leads to an output b and then c appears as an output on the input a , this clearly means that the process is in a different state as when it led to the output b . So whenever this happens a state change is assumed, the transition for this state change is labeled with the message that had a different output. Figure 7.1 visualizes this for the trace $a?b!a?c!a?c!a?d!a?d!$.

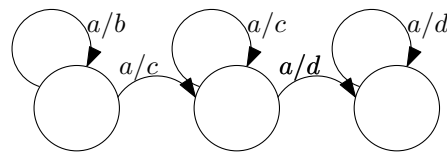


Figure 7.1: Example of a situation where a state change is applied.

From the new state the process starts over again, meaning that the input/output combinations that have been seen in the first state are not considered for the second state. So if the input a had two different outputs and led to a state change and then an input f while being in the second state has a different output then before in the first state, this does not mean that there is a state change since the different output can be caused by being in different states already. Figure 7.2 visualizes this for the following trace, $a?b!f?g!a?c!f?h!a?c!a?d!a?d!$.

When it happens that an output does not come within a time interval of a few seconds, 2 or 3 in example, after the last input, but there is a gap in the time line before the output appears that is much larger, this probably means that it took some time for the process to execute the message. This is modelled by an extra state that represents the passing of time it

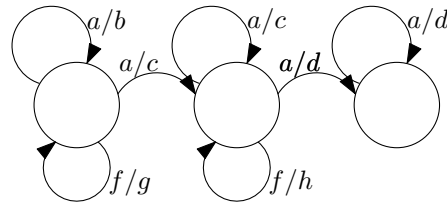


Figure 7.2: Example of a situation where no further state change is applied.

takes for the process to perform the action. Note that for this state not a timing requirement is added since it is not clear if it depends on the amount of time passed before the action is completed or not. It can be that it just takes some time before the process is finished doing the action and is able to perform the output, which means that the amount of time the process needs for this might differ for each instance. Figure 7.3(a) shows two possible situations that may occur, first there is a situation where the input d is answered by the output e and after some time with the output f and second there is the situation where the input g is after some time answered with the output h . That a separate state is added is a choice of the system expert, another way to do it would be by creating self loops at the state with an input maybe without an output and one self loop with only one or more outputs but no inputs, see Figure 7.3(b). This last option makes more traces possible with outputs that are not linked to an input. Note that in Figure 7.3(a) there are also outputs without inputs, but these are always preceded by the same input that caused the state change.

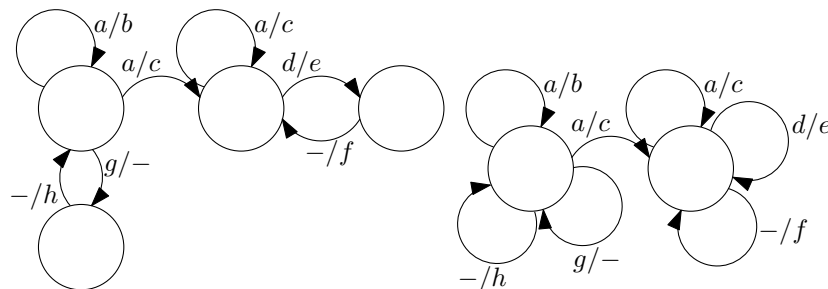


Figure 7.3: Examples of a situation where the output comes after a large period of time.

By this method it is possible to find different states from the process under interest and the transitions can be labeled in the way described above. Although this seems to be a reasonable way to label the transitions it is not clear if it is correct in the sense that this is the message that triggered the state change. In general it is not possible to know if the correct message is taken.

For this research project it was not possible to check if this heuristic works correctly since the amount of data was too small. For the same reason it was not possible to detect a stable state and loops in the model, the lengths of the traces were too small and the number of traces was not enough to find loops and a stable state. To create loops in the model an idea is to go back to a state if the message occurs again that triggered the state change. In example, if the input a first leads a couple of times to an output b and later on to an output c , following the method described above there is a state change to another state. If then later on it occurs that on the input a there is an output b again this could be seen as a transition to the first

state again, see Figure 7.4. The same holds when a couple of times input a with output d appeared, if now on the input a output b comes this can be seen as transition to the first state, while if the output would be c , it would be a transition back to the second state.

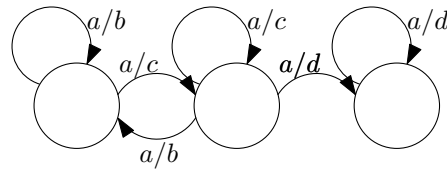


Figure 7.4: Example of how a loop can be constructed.

7.3 Procedure by example

The last section defines the different possibilities that exist within this method. By applying this on a trace a model can be extracted. In example, the following trace is given, $a?b!d?e!a?c!a?c!d?f!a?c!g?..<time\ break>..h!d?f!$. The model can be constructed in the following way:

Starting with one state from which it is assumed that all transitions are possible until problems with this assumption arise. So from the trace for first input/output combination $a?b!$ a self loop can be added without problems. For the second input/output combination $d?e!$ the same holds, since it is not contradicting with what is seen before.

Then the next input/output combination is $a?c!$. This is contradicting with what is seen before, since the input a earlier led to the output b and now to the output c . This means a new state is added and the transition between the states is labeled by the input/output combination $a?c!$. From this state the process starts over again. The input/output combinations $a?c!$ and $d?f!$ are not contradicting with anything seen before in this state so self loops are added. Note that $d?f!$ is not contradicting since this is a new state and the $d?e!$ combination that appeared before appeared in a different state of the model.

Next there is another $a?c!$ input/output combination for which nothing has to be added since the self loop already exists in this state. This is followed by the input $g?$ then there is a break in the time line and after some time there is an output $h!$. For this is an extra state added the transition towards this state is labeled by the input $g?$ and the transition from this state back to the previous state is labeled by the output $h!$. Finally there is another $d?f!$ combination in the trace, for which nothing has to be added since a self loop labeled by this combination already existed in this state. This process is visualized in a few steps in Figure 7.5.

The procedure as described in words in this section can also be described in a more formal and algorithmic way. To do this a few notes have to be made in the algorithm given here, $\lambda, S, current$ and ST are variables. λ is the postfix of the traces that needs to be handled, split into blocks of a single input and the following outputs till the next input. S is the set of states found till there, $current$ is the current state and ST is the set of transitions found till there. Then $(i, o) = \lambda.first$ means taking the first block consisting of a single input and the following outputs and removing it from λ and assigning it to (i, o) . $temp = NewState$ means creating a new state and assigning it to $temp$ and finally text between $< --$ and $-- >$ is just comment for understanding. The algorithm is shown here:

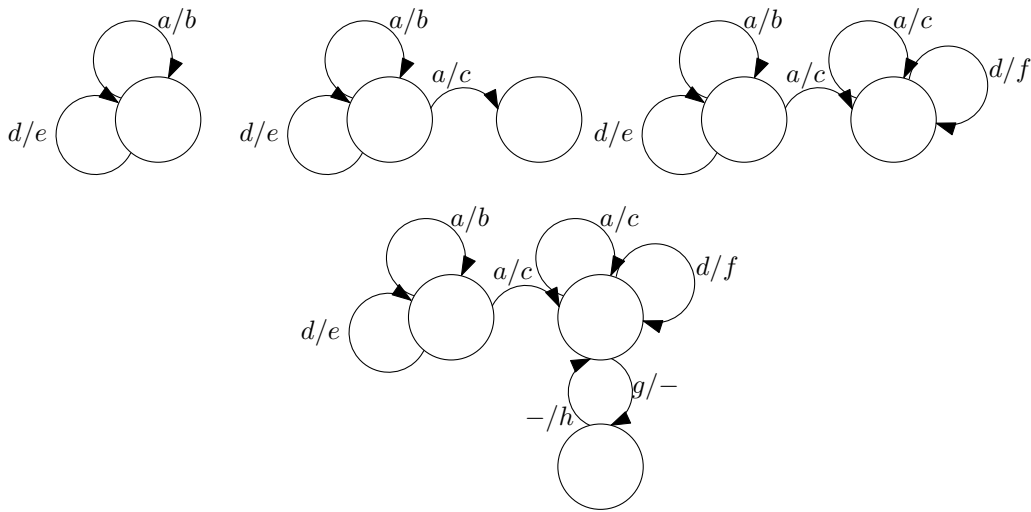


Figure 7.5: Visualization for the procedure in a few steps.

```

 $\lambda \in \Sigma^*$ 
 $S = \{s_0\}$ 
 $current = s_0$ 
 $ST = \emptyset$ 
while  $\lambda \neq \emptyset$  do
   $\leftarrow$  As long as  $\lambda$  is not empty do  $\rightarrow$ 
   $(i, o) = \lambda.first$ 
  if  $o$  contains time break then
     $\leftarrow$  First check if there are time breaks in the block considered at the moment  $\rightarrow$ 
    Split  $o$  in  $o'$  and  $o''$  on first time break
     $temp = NewState$ 
     $S = S \cup \{temp\}$ 
     $ST = ST \cup \{current \xrightarrow{i/o'} temp\}$ 
    while  $o''$  contains time breaks do
       $\leftarrow$  Check for more time breaks  $\rightarrow$ 
      Split  $o''$  in  $o'$  and  $o'''$  on first time break
       $temp2 = NewState$ 
       $S = S \cup \{temp2\}$ 
       $ST = ST \cup \{temp \xrightarrow{-/o'} temp2\}$ 
       $temp = temp2$ 
       $o'' = o'''$ 
     $ST = ST \cup \{temp \xrightarrow{-/o''} current\}$ 
  else
    if  $(\exists_{s_2} : current \xrightarrow{i/o} s_2 \in ST)$  then
       $\leftarrow$  Check if a transition labeled by this input/output combination exists  $\rightarrow$ 
      Pick a  $s_2$  such that  $(current \xrightarrow{i/o} s_2 \in ST)$ 
       $ST = ST \cup \{current \xrightarrow{i/o} s_2\}$ 

```

```

    current = s2
  else if ( $\exists p : current \xrightarrow{i/p} current \in ST$ ) then
    <- Check if this input/output combination is not contradicting ->
    temp = NewState
     $ST = ST \cup \{current \xrightarrow{i/o} temp\}$ 
    current = temp
  else
    <- And if that all is not the case add a self loop ->
     $ST = ST \cup \{current \xrightarrow{i/o} current\}$ 

```

This gives an overview of how the procedure to extract a model would work for a trace as given above. Figure 7.6 shows that all three parts of the model that can be modelled by timed I/O-automata are explained now, continuous time driven events in Section 6.2, discontinuous time driven events in Section 6.3 and spurious behavior in this chapter.

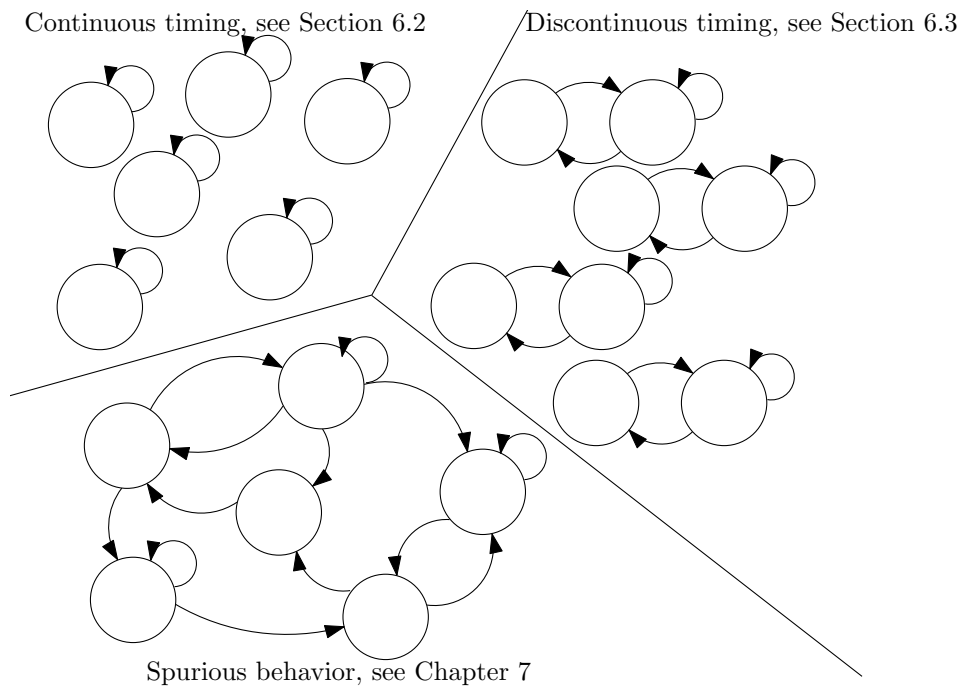


Figure 7.6: Part of the model that can be constructed after continuous timing.

7.4 Content of a message

When applying the method described in the previous sections to extract a model for the spurious behavior of the system the question rises again what content should be taken into account to distinguish messages. In example, when in the log file messages $a[x]$ and $a[y]$ exists as input messages and $a[x]$ leads to the output $b[]$ and $a[y]$ leads to the output $c[]$. In case no content is taken into account $a[x]$ and $a[y]$ are considered to be the same input message a and in that case the first time it will lead to an output $b[]$ and later on to an output $c[]$,

which means there should be a state change. While if the content of message a is taken into account then there will be no state change caused by this since $a[x]$ and $a[y]$ will be seen as different input messages. Figure 7.7 visualizes this for the trace $a[x]?b[]!a[y]?c[]!a[y]?c[]!$.

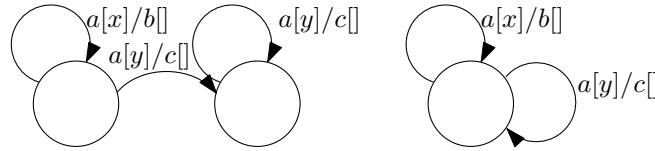


Figure 7.7: Example of how content has influence on the model, in the first case without content and the second case with content taken into account.

Figures 7.8 and 7.9 show two different models of the same log file. For the first model, Figure 7.8, no content is taken into account, so more messages will be considered the same, and for the second model Figure 7.9 all content of the messages is taken into account, so more messages will differ from each other. It might seem to be surprising that the more content is taken into account the less different states are found, but the more content is taken into account the less times the “same” input message is discovered and the less times a “different” output is found for the “same” input.

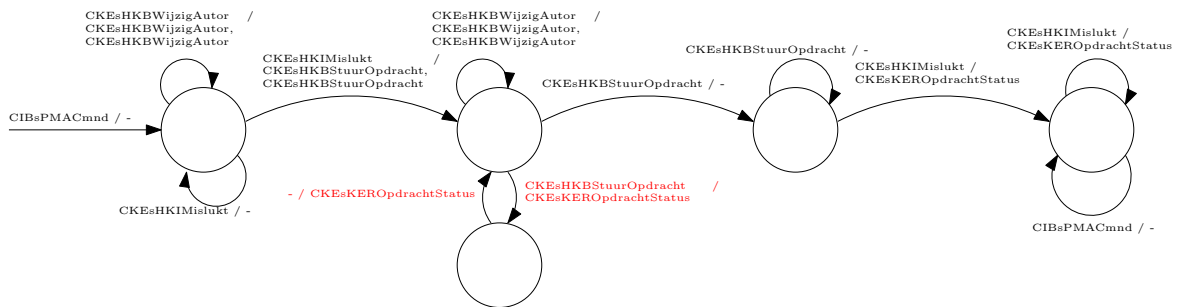


Figure 7.8: Example of a spurious model from a log file without taking content into account.

The decision about what content should be taken into account can not be made automatically and has great influence on the model that can be extracted. For this reason the knowledge of someone who can decide which content is important and which is not is needed. For some parts of the content this can be decided by just taking a close look at it. E.g. when a part of the content exists of the unix time stamp, which are 10 digits that increases in time, it is not needed to take it into account. If that is still the case with a part of the content that exists of 4 digits and might be something like water heights is not clear and needs the review of some system expert. In this case the expert decided that Figure 7.8 would be the best model for the process.

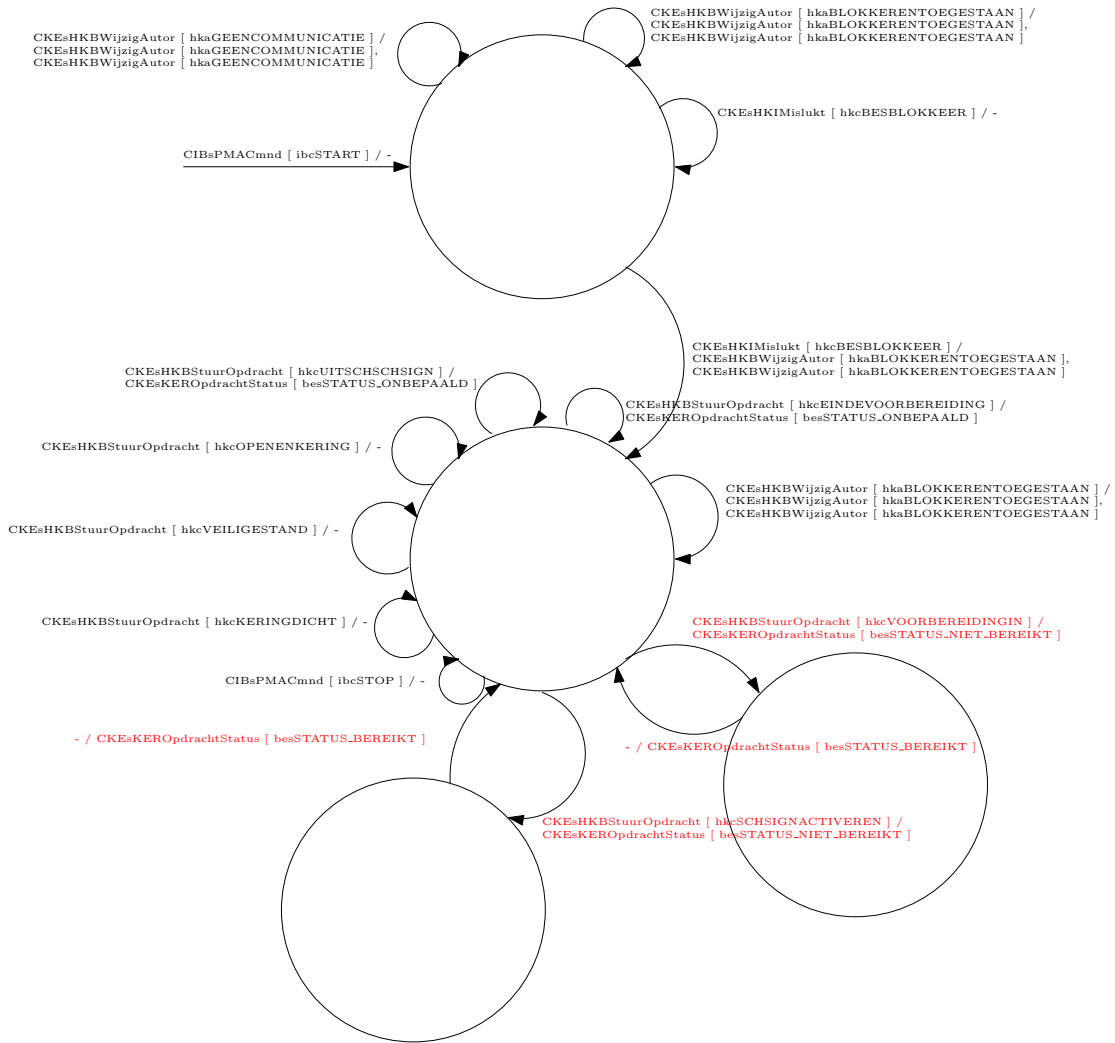


Figure 7.9: Example of a spurious model from a log file with taking content into account.

Chapter 8

Conclusion and future work

8.1 Conclusion

In this paper a method is presented to extract a model from log data and more specific log data from the BOS system. Many assumptions have been made about this BOS system, that all have influence on the model that can be extracted from the data. This means that it has become clear that finding the original model, as made in the design phase of the BOS system, will be very hard if not impossible. Even finding a good model that still has some predictive value, a model that describes the same behavior as possible in the system, is hard. Nevertheless it is shown that the timed behavior of the system can be modelled from the log data and that this might be possible for the spurious behavior, even though it requires the knowledge from a system expert at some points. The model extracted in this paper shows that the timed behavior of the system can be found and modelled in a correct way, but that for the spurious behavior more data is needed and that there are some open questions for this part.

As explained it is important to start with finding the time driven messages. This shrinks the size of the log file with about 99 percent which makes it possible to handle large log files that contain ten thousands of messages that were sent over a large period of time, for example a couple of days. From these time driven messages some elements of the model, as shown in Figures 6.5 and 6.9, can be constructed that define the timer and its period separately from the spurious behavior. Also a method is given to detect the start and stop messages for the timers of discontinuous time driven messages. In case there are not enough starts and stops, so the number of continuous blocks for discontinuous time driven messages is too low to determine the message that starts and stops the time driven messages, adding a start and stop message in the log file gives a solution to at least be able to handle this situation and to model the starting and stopping of the timer in the spurious behavior.

For the detection of the spurious behavior a first step of how this behavior can be found and how it can be modelled is given. It is shown how state transitions can be found and modelled, as well as how these transitions can be labeled. It is also explained how time consuming actions can be modelled. Further an idea, although incomplete, of how loops can be detected is given, which needs more research to be useful. The modelling of the spurious behavior is incomplete since important parts of the spurious behavior, such as finding a stable state, are left out of consideration.

8.2 Future work

Testing on large amounts of data

The methods presented in this paper need to be tested on a larger scale for a large amount of log data. At this moment there was only enough data available to test the continuous time driven messages in a good way and to detect the discontinuous time driven messages. But to test the methods of detecting start and stop messages for discontinuous time driven messages and to test the methods presented in the chapter about spurious behavior, Chapter 7, more log data is needed that contains this information. The log data needed to test these methods can come from the BOS system. This data is not available at this moment and may be impossible to obtain. It is also possible to generate log files from an already existing model to test the methods of model extraction described in this paper provided that the log files and the system they describe meet the requirements mentioned in Chapter 4.

Content

As shortly discussed in Section 5.2, it is not clear which content should be taken into account to distinguish messages and which not. The choices that are made to do this have great influence on the model that can be extracted from the log data. For this reason it would be advisable to do more research into this subject.

Finding start/stop messages

In Chapter 6 a method is given to extract the time driven behavior from the log files. To complete this method a few points of discussion and a few points that need to be tested are left over. The biggest point of discussion will be the detection of the start and stop messages for discontinuous timing, see Section 6.3. The method that is given to detect these messages might very well work, but the data that was available was not sufficient to test this method since only 2 or 3 blocks could be found. This means that the method to find these start and stop messages needs to be checked on a larger scale and maybe it needs to be adjusted to make it work correctly.

More timers

Then there can be a situation where there are two timers in one process that sent the same message, which actually occurs in the BOS system. Such a situation leads to strange results in the probability of messages being time driven, probabilities between 0.35 and 0.65. This happens for example with two timers t_1 and t_2 that both sent the same message m . The first timer has a period Π_1 of 30 seconds which starts at time 0 and the second timer has a period Π_2 of 60 seconds which starts at time 37, see Figure 8.1. Then time intervals between two messages of the same type, m in this case, get like 30, 7, 23, 30, 7, 23, 30, 7, 22, 31, 7, etc... seconds. For these situations a method needs to be found to detect this, which might be through probabilities of being time driven, and to detect the separate processes with periods etc.

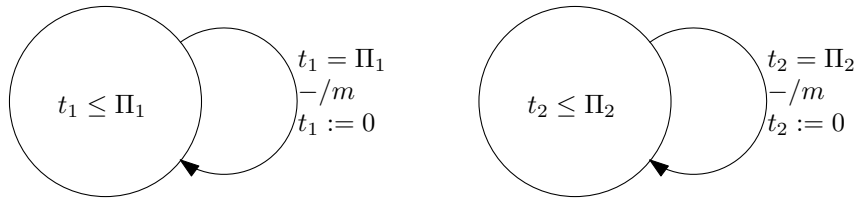


Figure 8.1: Example of two timer sending the same message m with different periods.

Changing timer period

Another situation that may occur is that the period of a timer changes after a certain message. For BOS in example this could be after the water height reaches a certain value, so below the value it checks the water height once an hour and above it it checks the water height every 10 minutes. It is very well plausible that such a situation occurs so it is necessary to think about how this can be detected and modelled. Possible decisions that can be made are to model it like a discontinuous time driven message or to change the timer period. In the first case there are two timers and the start message for one timer is the stop message for another and visa versa, resulting in a model like Figure 8.2, and in the second case there is a self loop that changes the value of Π . How this can be detected and which solution is best, or maybe none, needs more research and stays as an open question.

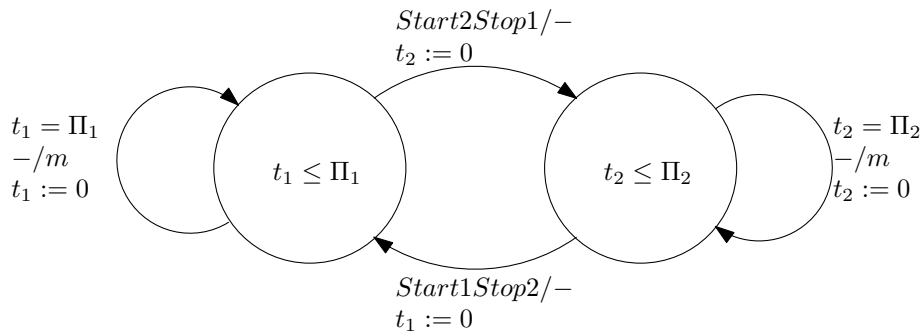


Figure 8.2: Example of first case where the period of the timer changes.

Replies on time driven messages

Then there is the situation that a time driven message needs a reaction, e.g. within BOS it would be possible that a message asks water heights resulting in an answer giving the water height at that moment. In the method described in Chapter 6 there will be no distinguishing between these two, meaning that they both will be assumed to be time driven while the first one is triggered by a timer and the second one by the first message. So it would be useful to distinguish time driven messages from answers to time driven messages. This is useful since the method used now will detect them both as separate timers, which makes other traces possible. E.g. when a time driven message $a?$ always gets a reaction $b!$, at this moment they both will be seen as time driven message of which the model will allow the trace $b!b!b!b!b!b!...$ too, while coupling them together will lead to traces that allow pairs of $a?b!$ and excludes traces containing $a?$ or $b!$ more than once after each other. To do this the method, from

Workflow mining [27], of counting how many times message a is followed by b maybe useful to take a closer look at. See Figure 8.3 as an example of how this situation can be modelled.

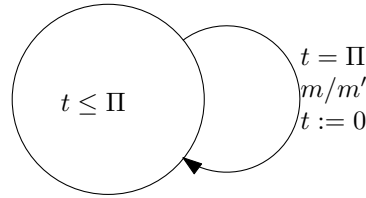


Figure 8.3: Example of how a time driven message with a reply can be modelled.

Continuous time driven blocks

There are situations in the log file where discontinuous messages are split into blocks and there are one or more blocks existing of plenty of messages with more or less the same intervals and there are a few blocks that just consist of a single message. How to handle this situation needs a closer look, since this single message can also be seen as spurious behavior or as a very small block of the discontinuous message. An advantage of seeing them all as time driven messages is that the result will be a more general and less complex model, with more continuous blocks. The disadvantage will be that it might be possible to put timers on and off in the model where it is not possible in the system and that it is hard to find the start stop messages for the timers if the single message is not time driven, since they will not exist in this case. It might be that better conclusions can be obtained when there is a good way to determine the start and stop messages for the timers of discontinuous messages. In example if the start and stop messages can be determined without these single messages it would be possible to see the single message as a continuous time driven block if the message is surrounded by a start and stop message.

Detecting loops

At this moment for the spurious behavior a method is given that makes it possible to detect different states and transitions between these states. Also it is shortly mentioned how to detect loops in the model, but this could not be tested due to data shortage. This means that there is much work that can be done in the future in this part of the model extraction. The method presented to detect loops and the points of discussion that are mentioned with it in Chapter 7 can be tested and adapted in such a way that the best solution can be chosen.

Stable state detection

Another part of the future work on the spurious behavior is the detection of a stable state. A stable state is a state in which the system keeps returning, some kind of a “rest” state. It is important to detect such a state since it gives the possibility to determine the start and end of a trace. A trace will start from that state and will end when it returns to the state and a new trace can start again. It is not clear how such a state can be detected, maybe it will just reveal itself when there is more data available, since the system keeps returning to one state, or maybe this state needs to be detected first. Although described shortly, much of the

future work will be in this subject, since it is important to make the model “complete”. A stable state makes the model complete since it defines a starting and ending point for traces that should be accepted by the system, which is not possible without a stable state.

Data conversion

Finally a subject that also will require much of the future work will be the data conversion. Within this paper the detection of data conversion is not covered at all, but is an important aspect of extracting a model from the log data. With data conversion it is meant that the data, e.g. variables, within the process are changed and mainly how they are changed, which can be the cause of a message a in the first case leading to an answer b and in a latter situation to an answer c .

So the main goals for this subject will be the detection of the internal variables of a process, the detection of messages that change these variables and how these variables are changed by these messages. Basically this is the part of the BOS system that is modelled by Z-schemes [22], which means that it is modelled different from the other behavior of the system and probably needs to be modelled different from the Timed I/O automata used in this paper.

Bibliography

- [1] R. Alur, C. Courcoubetis, and D. L. Dill. Mode-checking for real-time systems. In *Proceedings of the 5th Annual Symposium on Logic in Computer Science*, pages 414–425, Philadelphia, PA, June 1990. IEEE Computer Society Press.
- [2] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, September 1989.
- [3] D. Angluin. On the complexity of minimum inference of regular sets. In *Information and Control*, volume 39, pages 337–350, 1978.
- [4] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87–106, November 1987.
- [5] M. A. Arbib and E. G. Manes. Machines in a category: An expository introduction. In *SIAM Review*, volume 16, pages 163–192, April 1974.
- [6] T. Berg, B. Jonsson, M. Leucker, and M. Saksena. Insights to Angluin’s learning. Technical Report 2003-039, Department of Information Technology, Uppsala University, August 2003.
- [7] M. R. V. Chaudron, J. Tretmans, and K. Wijbrans. Lessons from the application of formal methods to the design of a storm surge barrier control system. In *FM ’99: Proceedings of the World Congress on Formal Methods in the Development of Computing Systems-Volume II*, volume Volume 1709/1999, pages 1511–1526, London, UK, 1999. Springer-Verlag.
- [8] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, MA, USA, 1999.
- [9] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: extracting finite-state models from java source code. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 439–448, 2000.
- [10] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, volume 407/1990 of *Lecture Notes in Computer Science*, pages 197–212, New York, NY, USA, 1989. Springer-Verlag New York, Inc.

- [11] W. Geurts, K. C. J. Wijbrans, and G. J. Tretmans. Testing and formal methods - BOS project case study. In *EuroSTAR'98: 6th European Int. Conference on Software Testing, Analysis & Review, Munich, Germany*, pages 215–229, Mervue, Galway, Ireland, 1998. Aimware.
- [12] E. M. Gold. System identification via state characterization. In *Automatica*, volume 8, pages 621–636. Pergamon Press, 1972.
- [13] E. M. Gold. Complexity of automaton identification from given data. In *Information and Control*, volume 37, pages 302–320, 1978.
- [14] A. Groce, D. Peled, and M. Yannakakis. Adaptive model checking. *Logic Journal of the IGPL*, 14(5):729–744, October 2006.
- [15] A. Hagerer, H. Hungar, O. Niese, and B. Steffen. Model generation by moderated regular extrapolation. In *FASE '02: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering*, volume Volume 2306/2002 of *Lecture Notes in Computer Science*, pages 80–95, London, UK, 2002. Springer-Verlag.
- [16] A. Hessel and P. Pettersson. A test case generation algorithm for real-time systems. In *Proceedings of the Fourth International Conference on Quality Software*, 2004.
- [17] G. J. Holzmann. *Design and validation of computer protocols*. Prentice-Hall, Upper Saddle River, NJ, USA, 1991.
- [18] G. J. Holzmann. Logic verification of ansi-C code with SPIN. In *Proc. of the 7th International SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.
- [19] IEC. Functional safety: Safety related systems, international standard IEC 61508, International Electrotechnical Commission, Geneva, Switzerland, 1996.
- [20] V. Kulkarni, R. Venkatesh, and S. Reddy. Generating enterprise applications from models. In *OOIS '02: Proceedings of the Workshops on Advances in Object-Oriented Information Systems*, volume Volume 2426/2002 of *Lecture Notes in Computer Science*, pages 270–279, London, UK, 2002. Springer-Verlag.
- [21] A. Nerode. Linear automaton transformations. In *Proceedings of the American Mathematical Society*, volume 9, pages 541–544. American Mathematical Society, August 1958.
- [22] J. M. Spivey. *The Z notation: a reference manual*. Prentice-Hall, Upper Saddle River, NJ, USA, 1989.
- [23] M.-L. ten Horn van Nispen. Johan van Veen. *Tijdschrift voor Waterstaatsgeschiedenis*, 10:16–20, 2001.
- [24] M.-L. ten Horn van Nispen. Veen, Johan van (1893-1959). *Biografisch Woordenboek van Nederland*, 5, 2002.
- [25] B. A. Trakhtenbrot and Y. M. Barzdin. *Finite automata: behavior and synthesis*, volume 1 of *Fundamental studies in computer science*. North-Holland, 1973.

-
- [26] G. J. Tretmans, K. C. J. Wijbrans, and M. Chaudron. Software engineering with formal methods: The development of a storm surge barrier control system - revisiting seven myths of formal methods. *Formal Methods in System Design*, 19(2):195–215, 2001.
- [27] W. M. P. van der Aalst, B. F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A. J. M. M. Weijters. Workflow mining: a survey of issues and approaches. *Data Knowl. Eng.*, 47(2):237–267, November 2003.
- [28] T. Weijters and W. van der Aalst. Process mining: Discovering workflow models from event-based data. In *Proceedings of the ECAI Workshop on Knowledge Discovery and Spatial Data*, pages 283–290, 2001.
- [29] T. Weijters and W. van der Aalst. Rediscovering workflow models from event-based data using little thumb. *Integrated Computer-Aided Engineering*, 10:151–162, 2003.
- [30] K. Wijbrans, F. Buve, R. Rijkers, and W. Geurts. Software engineering with formal methods: Experiences with the development of a storm surge barrier control system. In *Formal Methods*, volume Volume 5014/2008, pages 419–424. Springer Berlin / Heidelberg, 2008.