RWTH Aachen

Department of Computer Science
Technical Report

Time-Continuous Behaviour Comparison Based on Abstract Models

Jacob Palczynski

ISSN 0935–3232 · Aachener Informatik-Berichte · AIB-2013-20

RWTH Aachen · Department of Computer Science · December 2013

The publications of the Department of Computer Science of *RWTH Aachen University* are in general accessible through the World Wide Web.

http://aib.informatik.rwth-aachen.de/

Time-Continuous Behaviour Comparison Based on Abstract Models

Detecting Similar Signal Behaviours in Embedded Systems Test Evaluation

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der
RWTH Aachen University zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker
Jacob Palczynski
aus Zabrze (Polen)

Berichter: Professor Dr.-Ing. Stefan Kowalewski
Professor Dr. Thomas Rose

Tag der mündlichen Prüfung: 16. September 2013

Diese Dissertation ist auf den Internetseiten der Hochschulbibliothek online verfügbar.

Jacob Palczynski
The MathWorks GmbH
`jacob.palczynski@mathworks.de`

Aachener Informatik Bericht AIB-2013-20

Herausgeber: Fachgruppe Informatik
RWTH Aachen University
Ahornstr. 55
52074 Aachen
GERMANY

ISSN 0935-3232

Abstract

Model-based approaches in the development of embedded software become more and more widely spread. In these approaches, different models of one target system are the central elements. Recently, a growing number of algorithms and functionality of embedded software are designed using such model-based approaches. In controller design, Rapid Control Prototyping is a prominent example. Its main advantage is the possibility to develop functionality of both the controlling embedded software and the controlled system in one modelling environment. Additionally, the developer can simulate the whole system and improve its performance, debug algorithms, etc. in early development stage. Important test phases are Software in the Loop and Hardware in the Loop. In this work, we discuss two problems that may occur during these phases.

The first question is: Taking into account the continuous characteristics of system variables, how can we safeguard a consistent evolution of the development artefacts? To deal with this problem, data resulting from testing these artefacts under the same stimuli is compared. Since different artefacts were tested at different stages of the development under different conditions, we cannot expect that the results of each test case to be exactly equal, but at most similar. This similarity check of continuous systems is addressed in this work.

The methodology presented here is based on behavioural black-box models of our system in the time domain on different levels of abstraction. A behaviour is represented by the input and output signals of a system and their interrelationship. On each level of abstraction, a system's model is put up by a set of behaviours, each of which consists of input signals and the according output signals created by the system. The description of the signals themselves varies strongly, depending on the level of abstraction.

For the comparison of two systems or artefacts, we have to introduce a similarity relation with respect to an abstract model. Two systems are similar with respect to an abstract model A, when their behaviours conform to this abstract model A. The central question is how we can find properties in measured signal data. To find a characteristic property in a set of measured signal data, we compute the cross correlation of the interpolated measured data and a template signal. By this, we find on which time interval of the measured data one property potentially occurs. Repeating this for all properties yields all occurrences of the properties in the system's abstract behaviour model. In the end, we know that these systems conform to the abstract model and therefore are similar with respect to the abstract model.

The second problem we address is: Given real time test devices that are less expensive but have clocks with unknown precision, how can we make sure that the test results obtained from these devices are valid? The motivation here are possible differences in sample timing due to not exactly working hardware timers. Due to this, a drift occurs in the sampled data which leads to distorted signals and seemingly wrong timings of events. We compare such a signal to one obtained with a certified device by searching for certain properties, and obtaining the points in time of the occurrences. Using these time information, we can estimate the difference of the sample times, i.e. the drift. For the search for properties we use the cross correlation approach already used for regression tests.

Acknowledgements

First, I would like to express my gratitude to my advisor, Professor Dr.-Ing. Stefan Kowalewski for giving me the opportunity to join the Embedded Software Laboratory. During the development of this thesis, he provided me advice, guidance and support, and assisted me making important decisions.

During the ZAMOMO project, I had the pleasure to collaborate with, amongst others, Professor Dr. Thomas Rose. Since this work is based on my results in this project, I am glad that he is willing to be the co-referee of my doctorate thesis.

I sincerely thank the project partners for the fruitful discussions, and good collaboration: Dr. Michael Reke, Dr.-Ing. Martin Düsterhöft, Prof. Dr.-Ing. Dirk Abel, Dr.-Ing. Frank-Josef Heßeler, Dr.-Ing. Peter Drews, Dr.-Ing. Ralf Beck, Dr. Dominik Schmitz, and Harald Nonn.

Let me express my graditude towards Andreas Polzer. We started working together when ZAMOMO started, and shared an office for five years. I also thank my other colleagues at the Chair of Computer Science 11, for their constant support and good collaboration, especially Dr. Carsten Weise, André Stollenwerk, Daniel Merschen and Hilal Diab. Sebastian Moj helped to implement several parts of the prototypes, for which I am grateful.

Last but not least I thank my wife Irene and my parents Lucyna and Marcus for repeatedly pushing me towards finishing this work. Without these pushes I might have given up.

Contents

Abstract i

Acknowledgements iii

I. Continuous Data in a Discrete World 1

1. Introduction 3

2. Motivating Problems 9
- 2.1. Validation of Development Artefacts Using Back-to-back Tests . . . 11
- 2.2. Clock Drift Estimation of Real Time Testing Hardware 14

3. State of the Art 17
- 3.1. Mathematical foundations 17
  - 3.1.1. Automata and Hybrid Automata 17
  - 3.1.2. Similarity of Signals and Sets 19
- 3.2. Embedded Software Development 22
  - 3.2.1. V-model and Rapid Control Prototyping 22
  - 3.2.2. Model-based Design with MATLAB/Simulink 25
  - 3.2.3. Testing Embedded Software 28

II. Formal Ground 31

4. Abstract Behaviour Modelling 33
- 4.1. Syntax and Semantics 34
- 4.2. Association to Automata 42

5. Conformance and Similarity 45

III. Implementation – The Approach in Practice 51

6. Implementing Abstract Behavioural Models 53

7. Conformance Checking 59
- 7.1. Generation of Reference Signals . . . 61
- 7.2. Property Identification in Signal . . . 62
  - 7.2.1. Fitting and Interpolating Sampled Signals . . . 65
  - 7.2.2. Analytically Deriving Cross Correlation . . . 67
  - 7.2.3. Checking the Occurrences . . . 73
- 7.3. Validation of Order and Timing . . . 74

IV. Application 77

8. Back-to-Back Testing of Models and Autogenerated Code 79
- 8.1. Approach to Validation of Two Development Artefacts . . . 79
- 8.2. Generalized Approach to Conformance Validation . . . 79
- 8.3. Results . . . 81

9. Clock Drift Estimation 89
- 9.1. Approach . . . 89
- 9.2. Results . . . 92

V. Discussion & Outlook 97

10. Related Work 99

11. Summary 103

12. Future Work 107
- 12.1. Reference Signal Snippets . . . 107
- 12.2. Test Case Coverage . . . 108

VI. Appendix 111

A. Computation of Integrals I_l 113

List of Tables

4.1. Overview on the elements of behavioural models. 37

7.1. Characteristics of a choice of fitting and interpolation methods. . . . 66

List of Figures

2.1. Back-to-back testing. 12
2.2. Effect of a clock's drift. 15

3.1. V-model-like development process for control systems. 23
3.2. Model-based design. 26

4.1. Abstraction levels and their associations. 38
4.2. Semantics of a signal on PFL, TML, and SFL. 40

6.1. Simulink model on TML. 56
6.2. UML class diagram of abstract behavioural models. 57

7.1. Work flow of conformance checking. 60
7.2. Work flow of property identification. 63
7.3. Comparison of interpolation methods. 66
7.4. Example: Shifted PT_2 behaviour. 70
7.5. Cross correlation with `axcorr`. 71
7.6. Comparison between `xcorr`'s and `axcorr`'s results. 72

8.1. Similarity with respect to abstract model. 80
8.2. Testing using the conformance relation. 80
8.3. Measured disturbed signal of a vehicle's velocity. 81
8.4. Graphical representation of an example signal definition on PFL. . . 82
8.5. Example: signal definition on TML and on SFL. 83
8.6. Comparison of the interpolated measured signal and SFL model. . . 84
8.7. Reference signal for the example phase. 85
8.8. Analytically computed cross correlation of example test signal and reference signal. 86
8.9. Reference signal shifted against test output. 87

9.1. Effects of a clock's drift. 90
9.2. Using conformance checking for drift estimation. 91
9.3. Simulink model implementing a correctly working and a drifting clock. 94
9.4. Reference snippet. 95
9.5. Potential occurrences found with `axcorr`. 95

9.6. Mean square error for potential occurrences. 96

List of Definitions, Theorems, etc.

4.1. Definition (System) 35
4.2. Definition (Behaviour) 35
4.3. Definition (Signal) 35
4.4. Definition (Interdependency) 38
4.5. Definition (Family of functions) 39
4.6. Definition (Semantics) 39

5.1. Definition (Conformance) 45
5.2. Definition (Similarity) 46
5.3. Theorem 47
5.4. Lemma 47
5.5. Lemma 48
5.6. Corollary 49
5.7. Corollary 49
5.8. Lemma 50

Part I.

Continuous Data in a Discrete World

1. Introduction

When developing software for embedded systems, one has not only to take into account the behaviour of the software and its interaction with a human user, but also with the physical surroundings. This physical surroundings comprise in particular the physical system that is meant to be influenced or controlled by the embedded system and its software. Physical variables, e.g. time, velocity, pressure, angular acceleration, temperature, voltage, etc., are measured with help of sensors. Using these measurements, the control algorithm controls the actuators influencing the physical system's variables. While it is clear that the embedded software belongs to the discrete part of the system, one cannot assume the same for the embedding and controlled system, i.e. the plant. For macroscopic systems it is generally accepted to assume, that physical variables are continuous and can be expressed with real or complex numbers.

It is a known fact, that real numbers differ from integers when it comes to countability. Real numbers $\mathbb{R}$ form a continuous domain, and therefore are innumerable; in contrast to this, the domains of integers $\mathbb{N}$, and rationals $\mathbb{Q}$ are numerable. Although there are infinitely many integers, we still are able to count and numerate them; since rationals are fractions of integers, the same results holds for them. A known result of set theory concludes the opposite for reals, thus we cannot count them. In particular there is no injective mapping of the set of real numbers to the one of rationals, and on the other hand, it is not possible to map integers on reals surjectively.

What effect do these facts have on embedded software development? The most obvious one is that we are talking about hybrid systems, i.e. systems that incorporate both discrete and continuous behaviour. Here, the software part is discrete, while the embedding one in general behaves continuously. Sensors and actuators serve as interface between both parts and have to map the domains of the software part to reals, and vice versa. Still, when developing embedded software, the developer has to take into account the peculiarities of the continuous domain's real numbers.

In the development of controller software the integration of continuous and discrete behaviour is usually approached by Model Based Development (MBD) and Rapid Control Prototyping (RCP). In MBD models stand in the center of the development process, and are used for requirement engineering, early behaviour modelling, code generation, and testing. A subset of a collection of textual requirements is aggregated to an initial model which clarifies ambiguities, gives an outlook on the consequences of the requirements, and makes those executable via simulation. Such a model can consist of several components, for instance a controller and the plant. In the following development steps, this model is modified to meet additional requirements and to deal with limitations posed by the controller hardware, actuators etc. During this advanced development step, the introduced concepts can be evaluated, again making use of simulation, now for testing and tuning purposes. Finally, we can use the model as a blueprint for the controller's source code. This code can be written by hand, or generated automatically; combinations of both methods are possible, too.

The model-based approach in controller design incorporates mainly mathematical models of both plant and controller to capture the characteristics and behaviour of both elements. Modelling using block diagrams is a far spread method applied by control engineers and system designers. Such diagrams are composed of blocks and signals in between, thus a block diagram is a directed graph with the blocks being the vertices and the signals being the edges. The blocks represent mathematical functions that are applied to the input signals and produce the output signals.

A common tool used for modelling and simulating such block diagrams is MathWorks Simulink, which will be introduced later on. The simulation results represent early tests by which the general controller design can be validated, although no code for the embedded system was created at this point in time. After several repetitions of refinement and validation steps the controller design model becomes more mature. Based on these controllers models, code for controllers is written, usually in C, a process that can happen both manually or automatically with help of code generators. The code is compiled for and deployed to the target system afterwards, the hardware used as target system depends on the stage of development. The personal computer system already used for development is sufficient for serving as an early evaluation environment. Later on, the software is downloaded on evaluation boards carrying the target processor. The controller is tested iteratively, and the obtained test results are used to improve both model and code. This process is called RCP.

Although in MBD the model stands in the center of the development, different other artefacts may be created during the development process. Amongst others, C code produced on base of the model is an important one. The behaviour of an executable artefact can be expressed by the input and output signals. As is pointed out by Pehinschi, et al. in [41], the behaviour of model and compiled code can differ. Depending on the requirements, the differences might be tolerable. Still, the need for validation and safeguarding of embedded code developed in a model-based way exists.

There are several possible reasons for the differences in the behaviour artefacts originating from different development phases. To achieve accuracy, for instance, models can be simulated with variable sample rates, thus achieving an almost continuous behaviour. In later development phases, e.g. in hardware in the loop tests, control software is executed periodically in fixed time steps. Due to this, accuracy may be lost, and the system's behaviour changes. Another reason for varying behaviours lies in the use of different sets of numbers and arithmetics. While in simulation one can make use of reals of double precision (64 bits), in embedded systems this is often not possible due to hardware limitations. In this case, integer arithmetics is usually applied, or, for better accuracy, fixed point arithmetics.

Even if real numbers would be used in an embedded system, differences between test runs with simulation and test were possible. Take, for example, the testing of a road vehicle by a human test driver in real environment. Here we have parameters, inputs and disturbances which cannot be defined and reproduced precisely. This might result in data obtained in the test case, which is similar but not equal to the requirements. Here, the whole vehicle – including the test driver – can be treated as one of the development artefacts.

These problems raise the question, how one can tell, whether two development artefacts behave in the same or at least in a similar way. There are several ways to check such characteristics, all of them incorporate *conformance*, a notion quite well understood in the software world. For conformance testing, there are established standards from IEEE, ETSI or W3C (see [19, 29, 55]). While these testing standards are driven from a practical point of view, conformance checking also has a firm theoretical ground (see e.g. [11, 53, 54]). This is not only true for pure software systems, but also for real-time systems where we have the notion *timed conformance*. Here *timed automata* (see [6]) and relations like *timed ioco* (see [45]) play an important role.

In the domain of embedded software, safeguarding automatically generated code is of utmost importance. By safeguarding code, one validates that the code conforms to the requirements. Several work in this field has been published by Stürmer, et al., discussing different approaches to safeguarding automatically generated code. In [51], a comprehensive overview on such methods is given where the authors discuss the validation of both code generators as well as of generated code. While formal verification of code generators is considered as not practically usable in an industrial environment, they stress the importance of testing (see also [12, 50]). One method the authors discuss, is the back-to-back testing of generated code and original model. Taking into account that in model-based design the models stands in the center of the development, we can assume that at a certain point of time the model meets a subset of its requirements. Ideally, testing model and code behaviour in a back-to-back method is equivalent to testing the code behaviour with respect to the requirements. To achieve equivalence, the model has to comply with each requirement.

In order to compare a development artefact with the model or in general two artefacts with each other, we will introduce a relation of artefact behaviour with respect to abstract behavioural models. By comparing the signals obtained in test or simulation with the abstract behavioural model, we can tell whether the analysed artefact behaves in the way specified in the model. We can extend this process in order to compare two artefact's behaviours, e.g. a model and code derived from it, or eventually to compare two systems in general. For this, we check whether both systems behave as specified in the model. If both do, we know that the systems' behaviour is similar with respect to (w.r.t.) the abstract behavioural model and do not have to check their behaviour for equality. In this way we can relate similarly, not equally behaving systems.

As pointed out before, due to the presence of the continuous signals when modelling a hybrid system software development is far more complex than in the pure digital case. First, it is not obvious what is the best way to specify continuous behaviour. Several approaches exist in literature, e.g. SysML, and Qualitative Simulation [20, 23, 31, 36], often focussing either on theoretical or on practical aspects. Further, depending on the formulation of a specification, it might not be easy to check if a continuous signal matches the specification, e.g. when the specification can not be mapped to an envelope. This is especially true when signals are not just piecewise linear functions, as e.g. in the theory of hybrid automata [25].

We will introduce a method for modelling software-intense embedded systems, called *abstract behavioural modelling*. The term *abstract* refers to the abstraction from the exact behaviour of the signals. Instead, it defines signals by the properties required by the system. Originally this modelling methodology has been motivated by the work in the project ZAMOMO (Verzahnung von modellbasierter Softwareentwicklung und modellbasiertem Reglerentwurf) which was funded by the German Federal Ministry of Education and Research [48, 49]. The goal of ZAMOMO's participants from industry, universities and research institutes was dovetailing model-based software development and model-based controller development. One means to tackle that problem was a modelling methodology for control plants, incorporating hierarchical abstractions that would match better with the modelling of software. In the respective working package, we decided to concentrate on a behavioural approach abstracting from a signal's exact values and timing.

This method is based on the work of Wiesbrock, et al. [20, 23] and has been presented in [37, 40]. The abstract model is needed to capture the fact that signals in real application do not match exactly their specification: a rectangular signal is never an ideal rectangular signal in practice. By using abstract models, we can capture this inaccuracy by specifying properties instead of defining exact signals: the rectangular signal could be described by the property that it changes from an interval around 0 to one around 1 within a time interval between three and four seconds after system start.

Given an abstract model, we need to to find out if a measured signal lies within the semantics of the model, i.e. is one of the signals allowed by the model. This leads to the definition of the *conformity* of a signal to an abstract model [38, 40].

Based on this conformance relation between signals and models, we generalise our notion and define similarity of two systems showing continuous behaviour with respect to an abstract model. A major point in our definition of similarity is that similarity of two systems does not only depend on the systems' signals themselves, but that the abstract model has to be taken into account, i.e. two signals can only be related with respect to the abstract model.

As an example, take two rectangular signals with a rising edge at time eleven and fifteen seconds. Under an abstract model with a rising edge between ten and twenty seconds, these two are similar to each other, while they are not under an abstract model with a rising edge between eight and twelve seconds. The suggested check whether a sampled continuous signal conforms to an abstract model consists of three steps.

1. We start with choosing accurate reference signals representing the abstract model.

2. In the second step, called *property identification*, we use cross correlation and mean square error to identify the fragments of the measured signal for the different properties in the abstract model.

3. The order and the timing of the measured signal is checked in the last step.

Note that our method is a semi-decision procedure. If we find that a signal conforms to an abstract model, then this is actually the case. However, even if the last step fails, we cannot conclude that the signal does not conform to the abstract model. Instead, the reference signal might have been badly chosen, and the fragmentation for the property mapping is not the correct choice for the given signal. We will discuss some approaches to this problem in the outlook. In general, we assume that the reference signal is well-chosen by a domain engineer, and that his expertise is helpful in finding reference signals for properties.

One might argue that by using several signals one could try to find a decent coverage of the abstract model's behaviour, i.e. covering the most probable or important behaviours. This way, it might even be possible to prove that a given measured signal does not conform to the abstract model. However this is not always possible. First, complete coverage is usually hard to find for arbitrary models; second, coverage will depend heavily on the application domain. General approaches were undertaken e.g. by Dang and Nahhal [14, 35], or Girard, Julius and Pappas [21, 22, 30]. We will discuss some aspect of their work when talking about related and future work.

In [37], we used the abstract behavioural models to provide a modelling technique for control plant requirements. In this work, we will take a closer look at that application as reference for the similarity comparison of test signals (cf. [38, 40]).

Recently its application range was extended to a property based estimation of clock drifts [39]. We will present the later two with help of examples in this work.

In the following two chapters we present the two example problems, which also serve for evaluation. Additionally the setting and basis of this work, namely the mathematical foundations and model-based development of embedded software, will be covered. In the chapters of the following part, we introduce the abstract behavioural modelling and the similarity relation formally, before we turn on the implementation of both concepts. As indicated before, these concepts are evaluated, which results are presented in Chapter 8 and 9. Eventually we discuss the results of this work and give an outlook on possible enhancements.

2. Motivating Problems

This work is motivated by several problems in the phases of an embedded software life cycle, e.g. in the requirements analysis, software design, implementation, validation, deployment, and utilisation. During utilisation, the software of an embedded system processes data originating from the outside which usually is of continuous nature, i.e. real numbers. In contrast to this, hardware limits embedded systems to computing with discrete data, and even data of type `double` is only quasi-continuous. Often floating point operations are too slow to meet real-time requirements, so that fixed point arithmetics is used instead, leading to even coarser representation of real number variables. These problems have to be considered while developing software, accompanied by several additional questions. For instance, the set of real numbers is innumerable, thus there exists no method to test all instantiations of one real variable. The problem gets even worse when taking into account that this variable's value may change over time, which is due to time being also continuous.

A good example for such embedded systems are control systems consisting of a physical system, called plant, controlled by an electronic device, the controller. The physical system's behaviour is characterised by its continuous state variables' development which depends on the system states' current values and the system's inputs. In order to describe such systems we put up differential equations of the form $\frac{d}{dt}x\left(t\right) = f\left(x\left(t\right), u\left(t\right)\right)$, x and u being vectors of state variables and input variables, resp. For example, the change of a car's current velocity does not only depend on the torque applied by the engine, but also on the current speed, that influences the drag working against acceleration caused by the engine. In this example, both torque and drag serve as inputs, while velocity is the output that we observe. If we want to control the current speed automatically, we have to take into account this structure. The engine's torque is the input variable, which can be influenced directly, while the drag acts as disturbance. As mentioned above, we observe the current speed and our aim is to set this speed to a desired value.

Generally speaking, there exist two possible structures to control a physical system, open loop and closed loop control. In the first mentioned control structure, the controller does not observe the plant constantly. Thus, there is no feedback of the plant's current state. In contrast to this, a closed loop controller relies on the constant or sampled measurement of state variables and calculating the control error. By reducing this error, the controller tries to drive the plant to the desired state. In our example, such a closed loop controller would measure the current value of velocity, compare it to the one set by the driver and change the torque produced by the engine.

Nowadays, controllers are implemented as embedded systems, incorporating specific computer hardware and software. When developing such Electronic Control Units (ECUs) one has to take into account the differences of 'the arithmetics in the physical world' and those of computer systems. While physics 'uses' real and complex numbers, i.e. innumerable sets, computers use discrete numbers, e.g. integers or rationals. When converting continuous numbers to discrete ones, inaccuracies occur and have to be dealt with. Additionally, ECUs do not work continuously, but perform input and output operations at discrete points of time. This short introduction to control systems only covers the most important topics with regard to the following work; more details can be found in literature, e.g. [33].

Testing is an integral part of the software development process, and is, generally speaking, the main method to validate certain properties of the developed software. The process of testing involves two main actors, a system under test (SuT) and an oracle. The SuT is the development artefact whose properties are to be validated, e.g. a program, a software module, etc. The oracle serves as generator for test cases and contains the information about the expected results of the tests. The test cases' design depends on the properties that one wants to validate. For example, testing the communication of distributed software requires different tests than those for validating that every line of code of a software component is reachable from the initial state. Other examples for testing goals are

- conformance to safety standard,
- state, path coverage,
- functional test, etc.

When testing embedded software and systems, we have to take into account the physical system and therefore, test of digital controllers involve testing methods used in control engineering. Let us take a brief look on automotive software development as one example. According to Schäuffele and Zurawka [42] three groups of test objects stand out:

Functional testing is used for controller tasks, but also for monitoring and diagnosis,

Component testing relates to real-time operating systems, communication and network management,

Behavioural testing is used to validate software and system in standard, extreme and error situations.

Especially behavioural testing has an important role in control design, since it checks whether a control algorithm complies to its requirements regarding controller performance. Those tests can be executed at different stages of the development process, e.g. in RCP behavioural testing is not only used for parameterisation and controller design, but also for integration [5] (cf. Sec. 3.2.3).

Since development artefacts in RCP are tested at several different steps of the development, those tests have the nature of *regression tests*. During regression tests the same test cases are applied to one component's artefacts originating from two different stages of the development process. Here, the older one serves as the oracle, while the newer one is the SuT. The similarity relation we will introduce here helps in the analysis of such regression test by facilitating the comparison and assessment of slightly different but similar behaviour.

In the following two section we will focus on two aspects in embedded systems development, back-to-back testing of systems with continuous variables and clock drift estimation of real-time testing hardware. Back-to-back testing compares the behaviour of two systems, e.g. two development artifacts; the aim is to show that both behave in an equal or at least similar way. Clock drift can occur in measurement devices made for real-time testing used in Hardware in the Loop (HiL) testing, which may lead to wrong sampling times and distorted measurements over time.

2.1. Validation of Development Artefacts Using Back-to-back Tests

Back-to-back testing is a testing methodology originally used to compare two revisions of a system, usually as regression tests. The same test cases are run on both revisions, yielding data on the corresponding behaviours of both systems. As mentioned before, back-to-back testing also can be used to safeguard autogenerated code [12, 51], treating the two different artefacts as a system's two revisions. Here the original model's behaviour can serve as the oracle, while the generated code is considered to be the SuT. Several metrics can be applied to such tests, in our case we are interested in the observable behaviour of both systems. In the case of discrete data normally found in the software engineering world, it is a sensible approach to compare the obtained data for equality. However, it has disadvantages when it comes to continuous data.

Fig. 2.1 depicts the concept of the black box back-to-back testing, where two systems are stimulated in the same way and their reactions are compared. In our case, one of the systems is the oracle, the other one the SuT. Both, the SuT and the reference system are treated as black boxes, i.e. the internal structures are not considered. In particular, it is not necessary to monitor internal variables and states. In order to observe a black box' behaviour, one looks at the interdependency between

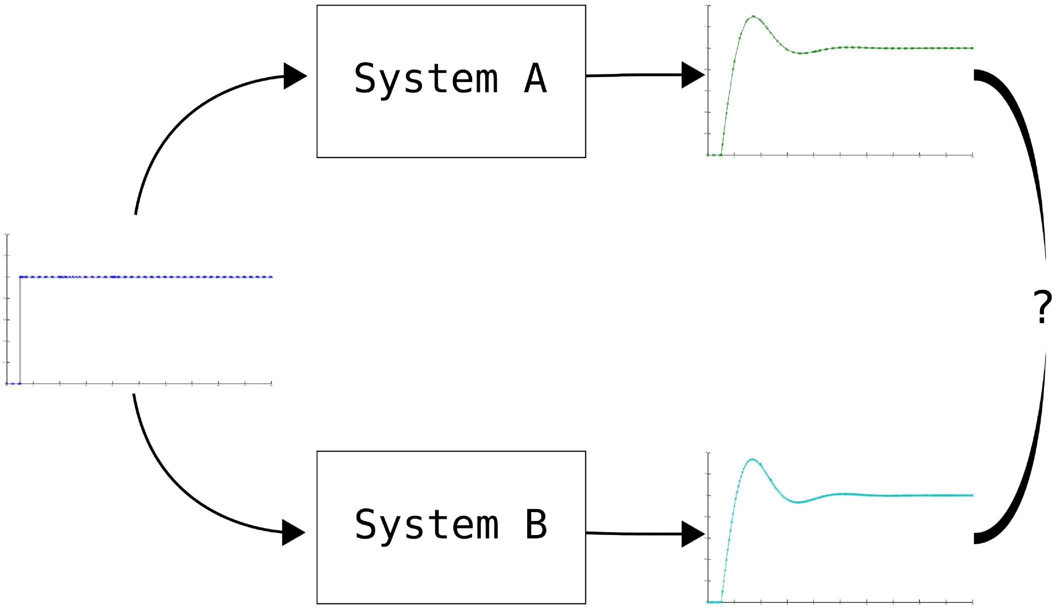
System A
System B
?

Figure 2.1.: Back-to-back testing: two systems are stimulated with the same input runs or signals; the outputs of both are compared.

the input and the output variables. As the SuT usually is a dynamic system, it is not sufficient to measure all variables' values at one time. Rather, the focus has to be put on the evolution over time.

It has to be emphasized, that black box back-to-back testing gives different answers than white-box test methods that, e.g., examine path coverage, etc. The focus of black box testing lies on the outside behaviour of the SuT. In fact, SuT and reference system can even have completely different inner structures. In such cases, metrics like path coverage do not make sense. Using the original model as the reference system, code automatically generated from this model can be validated by back-to-back testing. It has to be pointed out that according to [51] this is a good method to validate code generators with the help of appropriate test suites.

The behavioural back-to-back testing method has both advantages and disadvantages, and its usability depends on the characteristic of the SuT one wants to test. As long as one does not define output variables that feed the SuT inner state to the outside, one cannot say anything about the state, transition or path coverage. On the other hand, white-box testing hardly makes statements about the outside behaviour of a system, and metrics, such as path coverage, are not applicable. Moreover, the question arises when a continuous state was visited, since – as mentioned before – equality of real numbers can hardly be achieved. On the other side, behavioural back-to-back testing offers several important advantages. By observing the relation between the input stimuli and output variables, we are able to focus the comparison of different systems on their outside behaviours.

This comparison is useful when dealing with control systems, where the software system is meant to influence a physical system. In such cases we are interested whether the control software reacts on inputs from outside as desired and i.e. sets output variables in the required way. Thus, the inner structure of the program is not of utmost importance. The test cases defined by the relation between the input stimuli and the output variables developing in time can be used to validate the behaviour of both models and code. Here the question arises, how these test cases can be described adequately, catch the exactness needed in control systems, and cope with the difficulty to check equality of real numbers.

We will return to this problem in Chapter 8, where we show how one can apply abstract modelling for an evaluation of back-to-back test results. Abstract modelling allows a decent way of dealing with similar, yet not equal signal runs; the inequalities can occur in both value and time. This method can be used generally for each two systems that are meant to behave in a same or similar way. Based on this modelling technique, we define a similarity relation w.r.t. abstract model. With this relation one can validate that two development artefact behave similar without the need achieving behavioural equality of both systems or equal measurements of those behaviours.

2.2. Clock Drift Estimation of Real Time Testing Hardware

HiL testing is an intensively and widely used method to validate software-intense embedded systems. In a HiL test an embedded system's interaction with a simulation of the controlled system (plant) is tested. The plant simulation usually runs on special hardware that emulates electrical interfaces to simulated sensors and actuators. Also the hardware has to meet real-time requirements, especially when testing feedback controllers.

The application of HiL simulations covers a wide range of domains, e.g. in automotive industry the vehicle dynamics, engines, breaks, and road characteristics are simulated. In real life, this data is continuous, but in simulation it is quasi-continuous because of the discrete nature of digital computers, as discussed earlier. With floating point numbers software can come close to continuous values. Furthermore, the simulation and execution of the embedded software happens at discrete points in time, i.e. the variables' dynamics become sampled.

Thus, correct timing and sampling is an important issue, especially the clocks of measuring devices have to be accurate. Although great effort is put into design and implementation of these devices, they can exhibit deviances from the specified behaviour. In practice we observe clocks that do run slightly faster or slower than expected. This leads to the problem that the resulting sampling rate is higher or lower, resp. than expected, we call this effect *drift*. When mapped on the time scale with the expected sampling rate, the longer the measurement takes the more distorted the measured data gets. Assume for example, a clock is running faster and therefore leads to a higher sampling rate. When we map the measured values on the expected time scale, the signal appears to be longer, and the expected properties seem to occur later as they really did.

This leads to a growing discrepancy over time, as depicted in Fig. 2.2. As we can see, both measurements look similar, exhibit the same signal properties in the same order, but values, change rates, and timing differ slightly. If the real signal did meet the requirement regarding time, the measured signal might not do, and even worse, a signal not fulfilling the real-time requirements might seem acceptable.

In order to assess the test's results correctly, the drift has to be estimated as exactly as possible. Currently this happens by running the same tests on one device with a certified clock and on the measurement device used for the HiL test. It is a valid assumption – backed by experience – that the stretch originates from the clock drift and not from different internal behaviour of the HiL devices. As discussed in [39], this process is not an easy one, e.g. it requires that such a certified measurement device is available. In Chapter 9 we will discuss a method to estimate the clock drift using abstract signal modelling.

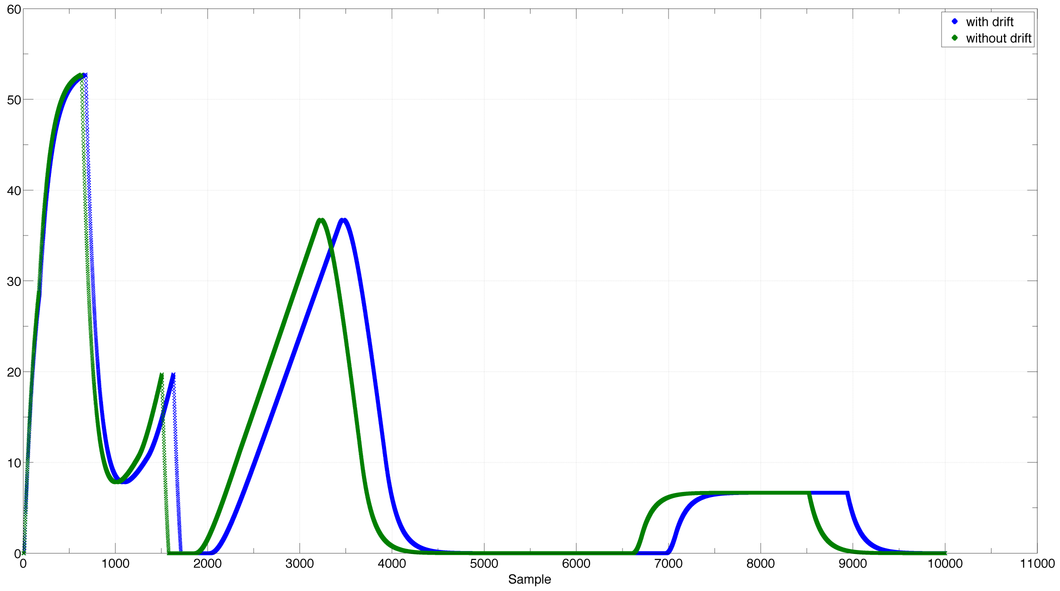

Figure 2.2.: The device with drift samples the signal more often, thus more values are obtained. When mapped on the same time scale along with the correctly working device, the signal seems to be stretched and longer.

3. State of the Art

In this chapter we introduce the fundamental concepts on which the abstract behavioural modelling and the similarity relation are based on. This includes both mathematical foundations as well as processes and tools used in embedded software development. We will start with the formal aspects of this work, including automata and set comparison, before we turn to model-based development of embedded software. There, we address the V-model, a widespread process in the development of embedded systems, and MATLAB/Simulink, a popular tool in this area.

3.1. Mathematical foundations

Let us first turn our attention to the mathematical aspects of abstract modelling and similarity relation. As discussed in Chapter 2, embedded systems, are physical or technical systems controlled by an electronic device, the controller. Nowadays controllers are small scale computers in which the functionality is implemented via software. To provide us with instruments to analyse the way such systems work, we have to introduce models that capture both physical variables and states, and discrete software behaviour. For this purpose automata related models, e.g. statecharts, hybrid automata, are widely accepted tools, since they incorporate the behaviour of software systems switching between discrete states, and the continuous behaviour of differential equations. We can represent signals by functions in time mapping reals to reals in the case of continuous signals or rational numbers to rational numbers in the case of sampled and discretized ones. In order to compare continuous and quasi-continuous signals we need also a notion of similarities of sets of real numbers and rational numbers, resp.

3.1.1. Automata and Hybrid Automata

This section briefly reviews some basics on automata, formal details are omitted and can be found in literature, e.g. [27]. The *Deterministic Finite Automaton (DFA)* bases on a transition systems, and consists of

1. a finite set of vertices,
2. a finite set of input symbols,

3. a transition function, represented by a finite set of labelled edges, mapping a state and an input symbol on a state,
4. a start state, and
5. a set of accepting states.

Nondeterminism is introduced by changing the transition function. While the deterministic one maps each vertex to exactly one other vertex for each input symbol, the nondeterministic one maps to a subset of the set of vertices and also allows transitions on the empty string. The resulting automaton is called Nondeterministic Finite Automaton (NFA). When we add an output component, we obtain Moore and Mealy automata, resp. The output of a Mealy automaton depends on both state and input symbol, i.e. the transition function maps both onto a state and an output symbol. In contrast to this, the output of a Moore automaton just depends on the state and not on the input symbol.

We call a sequence of an automaton's states respecting the transition function a *run* of this automaton. When a finite run of an automaton end in an accepting state, the sequence of input symbols causing this sequence is accepted. In embedded systems and in control systems in particular, operations often are not meant to terminate, but to run safely for an indefinite time interval. We can expand the notion of runs to infinite ones, in order to be able to accept infinite words. To this ends, we need a different notion of acceptance which leads to so called ω-automata [52]. These automata are still finite, since the number of states and input symbols is, but they operate on words of infinite length.

Until now, we only covered discrete automata exhibiting discrete state changes. If we want take into account real-time requirements and properties of physical systems, we need to introduce additional elements to the automata models. Modelling the time aspect in automata requires a notion of time, and a method to capture conditions related to time. Therefore, a finite set of continuous time variables called clocks is added to the discrete automaton, and by this we obtain *timed automata* [6]. The behaviour of each of these variables is represented by a function c that has the derivative $\dot{c} = 1$ in each state, and can be reset to $c = 0$ in transitions. We assume here that time passes only during the states and transitions take place instantly. The transition function's parameters are expanded by conditions, each of them can involve exactly one clock. Notice that a timed automaton's state consists not only of the active vertex, called location in the context of timed and hybrid automata, but also of the values of all clocks. Therefore, the notion of a run of such an automaton must be adjusted accordingly.

Taking one step further, we discard the constraint $\dot{c} = 1$ of the change rate of continuous variables. Now, these variables do not measure the time, but represent physical variables, and consequently we do not call these variables clocks. By allowing arbitrary functions for the variables derivatives, including differential equations, we obtain *hybrid automata* [25]. With hybrid automata we are able to capture both continuous and discrete behaviour of a system, e.g. of an engine and an ECU. Similar as with timed automata, we have to keep in mind that a state consists of the location and the value of all of its continuous variables.

The automata presented until now differ particularly in the representation of states. Another possibility of extending the automata model is by introducing concepts such as hierarchy, history, etc. Capturing such concepts is possible with automata but it leads to even more complex models. In [24] Harel introduced *statecharts* which are finite automata with syntactic constructs to model these aspects directly. Statecharts enhance finite automata with additional concepts:

Hierarchy is achieved by subcharts; each of these can have a separate start state, and the states within a subchart can be accessed from other subcharts' states directly through transitions.

Composition of parallel states and charts facilitates modelling concurrent automata and subcharts.

Inter level transitions connect states of different hierarchy levels.

History connectors save the last active state of a subchart; this state becomes active when subchart is entered again.

Condition connectors are transitions ending in different states depending on disjoint conditions defined in the connectors.

Temporal logic can be used in transitions to express timeouts, etc.

Entry, Throughout, Exit actions in states are executed when a state is entered, while it is active (called *during* in Stateflow), and is left, resp.

3.1.2. Similarity of Signals and Sets

The motivating examples in Chapter 2 show the need to compare continuous signals. For this we have to make clear how a signal is defined. A signal $s : T \to D$ is a function mapping a set of points in time T to an domain D, e.g. $\mathbb{B} = \{0, 1\}$, $\mathbb{R}$, $\{\text{good}, \text{bad}, \text{ugly}\}$, etc. In our understanding the set of points in time T can be defined as one interval of $\mathbb{R}^+$, i.e. a continuous set of nonnegative real numbers. While we can assume that time is continuous, we usually evaluate it at discrete points, a process called *sampling*. Although the development of a signal is based on the continuity of time, we only observe 'snapshots' of its evolution. Still we have to keep in mind that these sampling points can be taken at any moment in time.

Besides the definition as a function, a signal can be understood as a set of value pairs $S \subset T \times D$. Thus, it is possible to access the problem of *signal comparison* in different ways. By the term signal comparison we address the question whether two signals are equal or at least similar in a certain way. One challenge lies in the continuous nature of time and real numbers. To reduce complexity, when measuring we always select a rational number instead of an irrational one. If the mathematical representations of two signals are known as formulas, it is quite easy to tell whether these are equal. Also, comparing two discrete sets for equality is trivial, thus comparing two measurements at the same points of time of a discrete signal is, too. The matter gets more complicated, when we take into account the continuous time in between the sampling points and deal with signal domains containing irrational numbers.

Here, again, we are forced to use sampled and discretized data representing the signal, thus our data is a subset of $\mathbb{Q} \times \mathbb{Q}$. The most straightforward method is to compare the signal's data set element by element for equality. But the sampling and discretization in the measurement process can lead to differences both in points of time and measured values. Additionally, measurement of the signals might be shifted in time, i.e. the sampling times of one signal lie in between those of the other. Thus the simple check for equality is not feasible in such cases.

One way to address this problem is to use a notion of *similarity* of sets. Given two arbitrary sets A and B to be compared, the general idea is to determine a *distance* in between. In a naive approach, we can compute the distances of all the pairs of the set $A \times B$, i.e. determine how distant each element in A is from each element of B. For these pairwise distances we can compute the means, the sums, or the minima, etc., and use the result as distance between A and B. In our case, the elements of the sets origin from $T \times D$, thus we would need to compute two distances, one distance on T, and one on D.

How can we determine the distance between two elements? There exist several metrics for distance comparison in discrete sets, e.g. Euclidean, Taxicab, or Chebyshev distance [7]. With these metrics, we are able to compute the distance between two elements of a set, and hence we can use them to compute also the distance within the set $A \times B$. Another possibility is to determine the centres of both sets and to compute their distance.

For continuous sets this task is more difficult to complete. The question of similarity of continuous data is important when it comes to determine the test case coverage. Here, one has to check whether the signal data corresponds to the test case definition, and equality check might be too strict. Some work was done recently by Dang and Nahhal on the field of test coverage of continuous and hybrid systems [14, 35]. Here they present several approaches to compute coverages and distances between continuous sets. Also, Girard, Julius et al. address similar topics in their work [21, 22, 30]. Dang and Nahhal focus on a metric called star discrepancy measuring how regular the distribution of a set is. Since for large sets the complexity of the computation of the star discrepancy grows exponentially, they present an estimation method (cf. Cha. 10 and Cha. 12).

Another possible approach to similarity of signals is to use their interpretation as mathematical functions. From the field of signal processing we know several methods to compare signal data, in the following we will focus on correlation and mean square error (MSE). Correlation $\rho(\tau)$ is the degree of how far two sets shifted by τ are linearly associated. Thus in signal processing this method is used to compare two signal functions $x(t)$ and $y(t)$. The general definition is

$$\rho(\tau) = K \int_{-\infty}^{\infty} x(t)\, y(t-\tau)\; dt \tag{3.1}$$

where K is a scaling factor. Cross correlation is a well-known method for comparing statistical series, analysing signals, etc., where it is used to evaluate the similarity of shifted data. In spite of that, there seems to be no application of cross correlation on comparison of simulation and test results of different length and sampling in the domain of control systems. The aim of using cross correlation is to find a shift, at which the signals would match. For each individual value of shift $\tau \in \mathbb{R}^+$ the product of the original signal and the shifted one is integrated over the signal's length T_F:

$$R_{xy}(\tau) = \lim_{T_F \to \infty} \frac{1}{T_F} \int_{-T_F/2}^{T_F/2} x^*(t)\, y(t-\tau)\; dt \tag{3.2}$$

where $x^*(t)$ is the conjugate complex of $x(t)$.

We will use cross correlation to compute the temporal location of a required mathematical property or characteristic originating from an oracle in the signal data obtained in a test. Each property is represented by one or more reference signals of a certain duration and sampling method. By using cross correlation it is possible to use reference and test signal data with different lengths and samplings. Values of $R_{xy}(\tau) \approx 1$ indicate shifts, at which the reference signal is strongly correlated with the signal data.

Additionally to the cross correlation we apply MSE to validate our results. This is necessary because cross correlation only gives us a relative measure how similar two functions are, and therefore does not say anything on the absolute differences of the compared functions. For the mean square error, we compute the differences between the values of two functions, square them, and compute the mean value. With this, we can validate whether a signal lies in certain predefined bonds around another one. Furthermore, we can also use maximum absolute and relative errors to estimate the differences of two functions.

3.2. Embedded Software Development

The development of embedded software differs from the development process for desktop software in certain aspects, especially when it comes to controller software. For instance, user interaction is not important, while interaction with physical variables, i.e. measurement and setting via sensors and actuators, is a central aspect. In automotive industry and control engineering, there are some established process models, e.g. V-model and rapid control prototyping, which will be introduced on the next pages. Some of these processes are also standard approaches in software development in general but inhibit certain peculiarities, others are specific to this field.

Another term used intensively in recent times are *model-based approaches* to software development. In control engineering model-based design arose from the traditional block diagrams. There, blocks represent mathematical functions which are applied to signals represented by directed edges. In model-based design, these block diagrams are used to model plant, environment and controller. Together, all components can be simulated, tuned and validated. In the end, from the control algorithm's model applications are generated automatically and deployed to evaluation or production hardware.

In the last section of this chapter we take a closer look on testing embedded software. Testing is a integral step in RCP and model-based design, and is applied after every design phase. Hence, diverse development artefacts are tested with different methods, in particular using in-the-loop tests.

3.2.1. V-model and Rapid Control Prototyping

The V-model is one standard development process in embedded software development and especially common in automotive industry [42]. Originally suggested by Boehm [8] in the late 1970s, this process model was developed by defence authorities in Germany and USA independently. In Germany, the German Federal Ministry of Defence was one of the driving forces behind the V-model, which later was applied for software projects of other federal authorities and became popular in automotive industry. The V-model illustrates the development process of a software system in both time and detail and consists of two branches forming a V (cf. Fig. 3.1).

The left branch represents the design and implementation of the software system, usually containing steps like requirement analysis, specification of system architecture, and software component design. The right branch comprises the development steps related to integration, test and validation of the software components as well as the whole system. The results of these steps flow back to the corresponding steps in the development branch in order to enhance the artefacts created there. Thus, while the development process progresses in time, its artefacts firstly become more

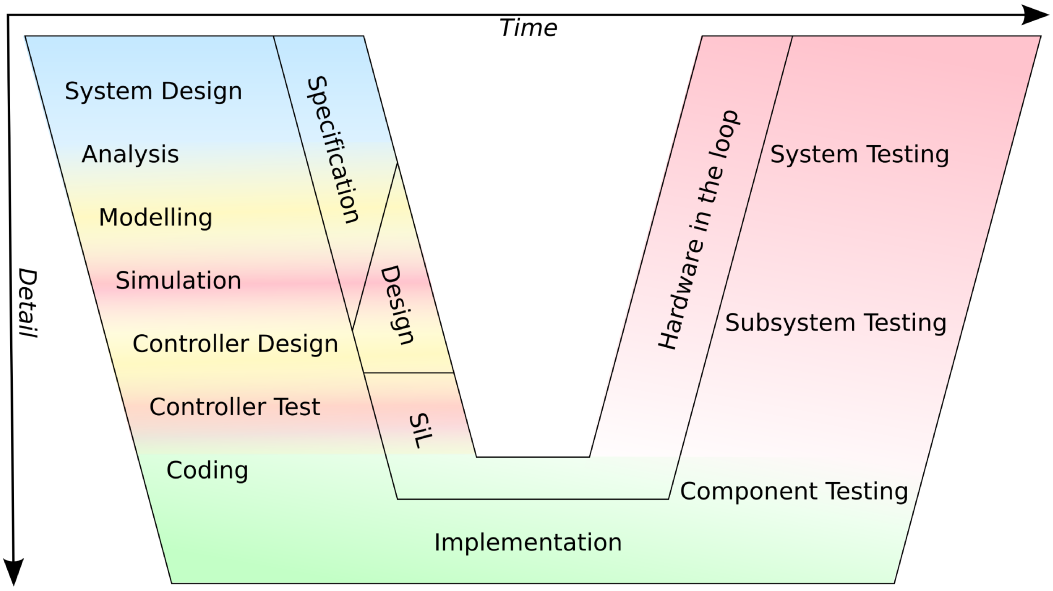
Time
Detail
System Design
Analysis
Modelling
Simulation
Controller Design
Controller Test
Coding
Specification
Design
SiL
Implementation
Hardware in the loop
System Testing
Subsystem Testing
Component Testing

Figure 3.1.: V-model-like development process for control systems (cf. [5]). Requirement-related tasks are coloured blue, design-related yellow, implementation-related green, and validation red (cf. Fig. 3.2).

detailed, before becoming subjects to testing. These two parts of the process lead to the aforementioned branches, the left one descending with respect to detail, the right one ascending.

Besides the general process, the V-model details out which development artefacts have to be passed in what form. Also, the documentation that is to pass to the next development step is established. Found errors in the right arm are fed back to the same level on the left, and after correcting the error, all steps have to be taken again as regression, including altering models, code, documentation, etc.

Fig. 3.1 does not illustrate the V-model generally known in software development. It is an process model adapted to controller development, and incorporates elements of *RCP* [5]. RCP introduces development steps for quick design and test of software-intense controllers. For this, a model of plant and controller stands in the center of the development, serving as executable requirement, design and test artefact, and base for automatically generated code. Crucial elements of methodology are a consistent tool chain consisting of both hardware and software, and repeating design-test-cycles. The software tools used in RCP have to support the control engineer in both modelling the plant and designing the controller. Therefore they must offer possibilities to mathematically represent physical properties of systems. Additionally, in order to accelerate the process it is recommended to generate the controller code automatically. The hardware used in RCP includes usual off-the-shelf PCs for early modelling steps, as well as evaluation boards and dedicated real-time simulation computers. Thus, the generated code has to be applicable on both evaluation and production hardware.

The first step in RCP is the design of the control system including an abstract view on both the plant and the controller. In an analysing step the engineer decides which variables serve as input and output, which states have to be controlled, which controller is suitable, etc. The analysis results are used to model the plant and to design the controller itself. As mentioned before, a good tool support is an important point here. After this step, we have a model of both the plant and the controller, and both can be simulated and tested at this early phase.

In order to use the developed control structure on an ECU, the controller model has to be transformed to an executable code. Usually, the controller model is translated to C code, which is then compiled to target-specific binary code. The translation from the model to intermediate C code can happen either manually or, and this has become more common recently, automatically with the so called code generators. These tools generate hardware specific code out of models according to user-specified parameters. Again, it is possible and recommended to test the obtained code in simulation with the modelled plant. The compiled binary is downloaded to an evaluation board for tests before it is used on the production hardware. Integration and further tests finish the development process. Testing is an important part of the RCP process, and we will focus on this in Sec. 3.2.3.

3.2.2. Model-based Design with MATLAB/Simulink

As described above, RCP implements a form of model-based development of embedded controller software. Here the model stands in the center of the development process and is used in four aspects [17]:

Requirements are transformed to models that can be executed in simulations; hence also dynamic behaviour specifications can be incorporated and validated in early development phases.

Design of algorithms refines the requirement models and leads to the models representing the controller that influences the plant model. Additionally, the algorithms are parameterised and the parameters fine-tuned.

Implementation and deployment of the designed algorithms more and more involves automatic generation of code which replaces handcoding and legacy code. Generating code is less error prone and allows to maintain coding standards and qualities. Additionally a found error does not have to be corrected in both model and code, but only in the model.

Validation and verification can be performed accordingly during each phase of the development process. Repeated simulations to test designed algorithms or generated code can show up errors in the design in early phases. Errors found can be corrected in the model, from where these corrections can be propagated easily to other development artefacts.

Fig. 3.2 depicts the central, interfacing role of models in a flexible model-based development process. The results of each step is implemented in or derived from the project models, leading to a quicker interchange between teams and a more flexible work flow.

There are several software tool chains supporting model-based design in embedded and control software engineering, e.g. ETAS ASCET, SciLab, National Instruments LabView, etc. A widely known and used development environment for embedded systems is MATLAB/Simulink by The MathWorks. MATLAB, an abbreviation for MATrix LABoratory, is originally a tool for numerical matrix operations. Based on these operations it now can be used in almost every domain from physics to finance. Domain specific functionality is provided via so called tool boxes. In control system theory MATLAB has been used since its early days and still is a quasi-standard in this field.

Simulink is a graphical modelling tool and simulator for block diagrams as e.g. used in control theory. It is based on MATLAB and the user can use MATLAB functionality within models as well as import data from and export to the MATLAB environment. A system in Simulink is composed of parametrisable blocks which

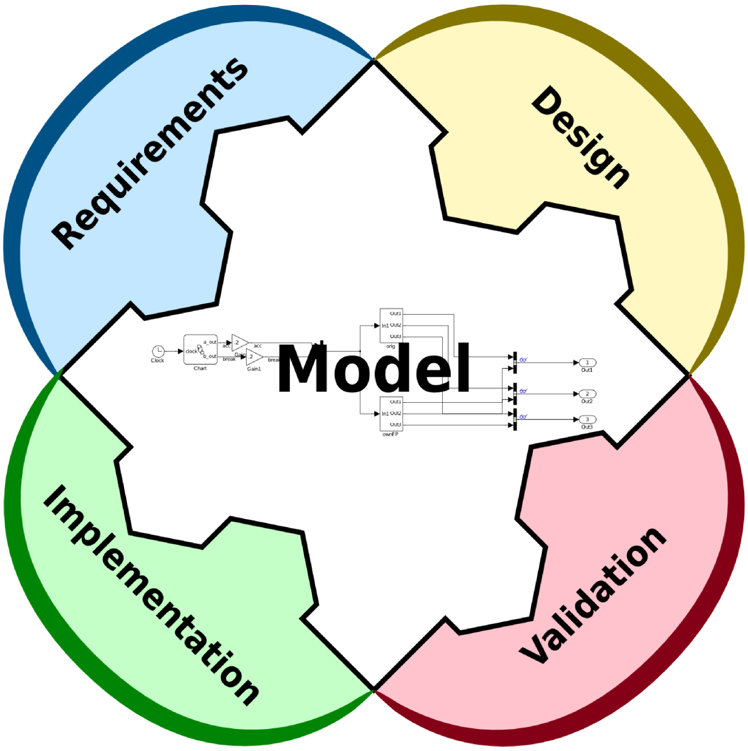

Figure 3.2.: The concept of model-based design. Models stand in the center of a flexible development process and serve as the interface between the requirements analysis, algorithm design, implementation, and validation.

represent mathematical functions, transfer functions, data type conversions, etc. Variables are exchanged via signals, i.e. directed connections. Thus, blocks and signals form a directed graph. As already indicated, signals have different data types, ranging from boolean scalars to arrays of doubles and even complex numbers.

A Simulink model's semantics is defined by its execution in a numerical simulation or by the generated code. A model is simulated at certain points in time, and the model's behaviour is observed via sampled signals. The sampling rate is determined by the so called solver depending on its internal, yet configurable rules w.r.t. the changes of signals and states. In every sampling time step a model's blocks are executed sequentially and in a deterministic way. Therefore, the individual blocks have to be ordered topologically, starting with blocks that have defined initial values, e.g. constant signals, integrators, etc. The solver determines the resulting values of signals and states by numerically solving ordinary differential equations, and chooses the next sample time.

Basic Simulink functionalities can be enhanced by the block sets, for applications in e.g. aerospace, automotive, etc. There are several block sets and features which are interesting for embedded software engineering, two of them will be introduced here: Stateflow and Simulink Coder. The Stateflow tool box consists of two additional Simulink blocks which allow to model logical decision mechanisms in form of Stateflow charts and truth tables, resp. Since Stateflow charts are a variant of statecharts (cf. Sec. 3.1.1), they facilitate the modeling of a wide range of transition systems. One enhancement of statecharts in Stateflow is the possibility of introducing continuous variable behaviour within states. This allows modelling of complex hybrid automata while benefiting from the statechart notation's advantages. Hybrid automata notations such as guards, invariants, etc. can easily be mapped to Stateflow elements using *continuous updating* of the statechart. For instance, condition connectors serve as guards and during actions can be used to express invariants. To describe the change of a continuous variable `var`, another variable `var_dot` is created, which is used to store the change rate of `var`. We will make use of these features when using abstract behavioural modelling in Simulink (cf. Cha. 6).

The MATLAB Coder and Simulink Coder[1] are used to generate C source code, which is then compiled with a third-party compiler. Hence, executables can be automatically generated out of Simulink models and MATLAB code. The Simulink Coder's output is meant to be used for simulation acceleration, rapid control prototyping and Hardware in the loop testing (HiL). In contrast to this, for generating production code for embedded systems MathWorks recommends Embedded Coder. Embedded Coder[2] is an enhanced code generator offering more possibilities to customize the resulting code. Generally speaking, both generators work as follows:

1. **Compile model:** After analysing the Simulink block diagram, an intermediate representation is created.
2. **Generate C code:** The intermediate representation is translated to C source code.
3. **Generate customised makefile:** A target specific makefile template is used to construct an appropriate makefile.
4. **Generate executable:** The user's system compiler is instructed to generate the executable program according to the makefile.

We will use Simulink in the later chapters to simulate behaviour of controller and plant, and generate behavioural test data.

---

[1] MATLAB Coder and Simulink Coder incorporate functionalities formerly included in Real-Time Workshop.

[2] Embedded Coder was known formerly as Real-Time Workshop Embedded Coder.

Since we will analyse the obtained data using algorithms provided or implemented in MATLAB, let us take a brief look at some of MATLAB's characteristics and features. MATLAB is a software package for technical computing including its own programming language of the same name. This language is an interpreted one, i.e. programs written with it do not have to be compiled, and it can also be used on the MATLAB's command line.

The main focus lies on manipulating and analysing large amounts of data organised in arrays. For this MATLAB facilitates working with arrays of arbitrary dimensionality in general, and matrices and vectors in particular. Most of the supplied functions are 'vectorized', meaning that they can be applied to an arbitrary array in one step, where the operation will be executed on each individual element. Thus there is no need for creating loops to work element-by-element on an array, and in fact the vectorised functions are by far faster then own looping constructs. Consequently, it is not recommended to use an divide-and-conquer approach, breaking up a big data array into small ones, but rather go the other way round. A good practice is to expand a small problem to a larger matrix and solve it in one step, thus trading in memory for speed.

For data analysis, basic MATLAB provides some predefined statistical functions, computing e.g. means, standard deviation, etc. Small programs compute correlations, basic polynomial fitting, or piecewise interpolation, frequency analysis with FFTs etc. are available as well. This basic MATLAB functionality can be expanded by own programs, or similar as block sets in Simulink by toolboxes. In recent years several improvements and enhancements broadened the possibilities of MATLAB, e.g. parallel computing, GUI programming and object oriented programming (OOP). These concepts allow the development of additional tools and individual extensions to MATLAB and use its build-in functionality for one's own purposes. More details on MATLAB in general and its extensions can be found in [4].

3.2.3. Testing Embedded Software

We have already pointed out the importance of testing embedded software. Especially in the RCP process, testing is of utmost importance since the diverse development artefacts require frequent testing. Some of the test methodology is depicted in Fig. 3.1, e.g. Software in the Loop (SiL) and HiL, both being important and common in control software engineering.

Usually, in SiL the SuT is the code generated from the controller model, while the test bench consists of the modelled plant. The simulation is executed on the development computer, commonly an off-the-shelf PC. With such tests we can find errors that were introduced in the process of translation to C code or code generation. Also different behaviour due to data type conversion can be detected.

The compiled software is tested in HiL tests. As mentioned before, the compiled code is downloaded to an evaluation board which supplies the necessary periphery and interfaces. This evaluation board is connected to dedicated simulation hardware, on which plant, sensors, actuators, etc. are simulated in real-time. With this, we are not only able to test the collaboration of software and hardware of the ECU, but also real-time issues can be found and resolved. The evaluation ECU from the HiL is often used in further tests, e.g. engine test, vehicle test, etc. HiL test benches comprise not only the simulator, but also include measurement devices and clocks. Since real-time is an issue, these clocks have to be precise in order to sample the signals at the correct sampling rate.

Additionally to the methods discussed above, there are also model in the loop (MiL) and processor in the loop (PiL) test [16]. While the first one is similar to SiL, but uses the controller model instead of the code, the latter is similar to the HiL test but replaces the dedicated simulator by the development PC. Often, such methods are applied in SiL and HiL without naming them explicitly [5].

While SiL and HiL tests apply to the tests of the control algorithms code and its collaboration with its environment and the plant, there exist also methods to test the models. Here, the test description method is of importance, Conrad has compiled an overview in [12]. In recent time, enhancements to those methods and some new testing tools came up, one of them is PikeTec TPT [2]. In Time Partition Testing (TPT) the user models test cases and scenarios with time region automata, hence it is possible to model the behaviour of a SuT over time. The modelled test cases can be used to stimulate the SuT, e.g. a Simulink model, and check the validity of its behaviour.

Another new development is the introduction of Continuous TTCN-3 by Schieferdecker, et al. [23, 43, 44]. Originally Testing and Test Control Notation Version 3 (TTCN-3) is a modelling language for communication and network system test cases, thus it mainly focusses on discrete data interchange between software components. CTTCN-3 expands the syntax of TTCN-3 by the constructs for modelling continuous input and output variable. While TPT is in use in industrial applications, there seems to be no wide scale deployment of CTTCN-3 in industry, yet.

Part II.

Formal Ground

4. Abstract Behaviour Modelling

As stated in Chapter 1, our aim is to compare the signals resulting from tests in how far they are similar to each other. The comparison will happen with respect to an abstract behavioural model which will be introduced in this chapter. The abstract behavioural model will serve as a criterion, while e.g. the SuT's and the oracle's signal data will be checked whether they lie in the set of accurate behaviours inferred from the abstract model.

Abstract behaviour modelling is a technique which enables us to model behaviours of systems with continuous variables in an abstract way. Using this technique, we capture uncertainties and numerical inaccuracies that occur when dealing with continuous variables. This approach is based on the work originally presented by Wiesbrock et al. in [20, 23], which we already used for modelling requirements for control systems [37], and also for test data comparison [38–40].

We will start with the description of the modelling hierarchy introduced in the original work [37–40], before turning our focus on its importance for the approach of conformance checking. This abstract behaviour modelling technique was conceived when the author was collaborating in ZAMOMO, a project funded by the German Federal Ministry of Education and Research [3]. In this project, university laboratories (Embedded Software Laboratory, Institut für Regelungstechnik at RWTH Aachen University), research laboratories (Fraunhofer FIT), and enterprises (VEMAC GmbH & Co. KG, AVL AG) were working on dovetailing model-based control design and model-based software development.

An important aspect in this project was the fact that the enterprises were small or middle sized companies. Such companies usually depend on quick responses to customers' needs with little man-power and resources, i.e. when giving an estimate to a potential customer, early development stages have to be performed beforehand. Thus the employable solutions have to be put together with limited financial resources and man-power. Besides, due to competition such companies also have to be flexible and swift.

One of the problems approached in this project was the development of a modelling methodology that should be used in early software development stages to model control plants. With the resulting four-level modelling hierarchy the following challenges are addressed [37]:

1. Capturing of vague information on plants from requirements and specifications,

2. comparison of accurate plant models and experimental data on the one hand and requirements and specifications on the other hand,
3. early simulations based on incomplete knowledge of the plant, and
4. simulation of dynamic behaviour.

When abstracting the exact behaviour of a continuous system, we have to focus on the observable input and output signals. The system's behaviour can be described by the dependency between these signals or by setting up a transfer function, which for linear systems allows to predict all behaviours. The abstract modelling hierarchy in [37] consists of four abstraction levels – Property Flow Layer (PFL), Time Metric Layer (TML), Signal Function Layer (SFL), and Transfer Function Layer (TFL) – each of which differs in the abstraction made from the exact signal behaviour. Further, because the mentioned work copes with linear systems, the possibility to describe a system with the means of one transfer function is included. However, non-linear systems cannot be described by a transfer function in general, but only for certain state intervals, and the results of the describing transfer functions show discrepancies when the system leaves those. In the more general setup in [38] – which can also deal with non-linear systems – we exclude the transfer function as they are limited to linear systems only.

In the later work the focus moves to testing control systems. Here, the most accurate level is not needed anymore, since it is already included when using block diagrams. This gives more freedom and we are no longer limited to linear systems. The most accurate possibility for behaviour modelling is formulating signal functions, instead. These functions define mathematically exact the dependency between a continuous time set and continuous value sets. On the other hand, i.e. the most abstract model, we have a property-based model, where we have neither continuous time nor values. Instead, we put up sequences of mathematical properties of signals described in natural language. Between both extremes, we first add continuous time and then continuous values to refine the description of the system's behaviour.

4.1. Syntax and Semantics

To introduce the abstract behavioural modelling hierarchy, we start with the levels which are most useful for the similarity relation's definition. These are the Property Flow Layer (PFL) and the Signal Function Layer (SFL), the first one abstracts from both time and value, while the latter consists of absolute time and accurate values. Models on the PFL represent the definition of a set of test cases by capturing the interdependency between a system's inputs and outputs in an abstract way. The individual accurate test cases describing the system's behaviour are represented in SFL models. By this, we obtain accurate data for stimuli and for the property identification we introduce in Sec. 7.2. The Time Metric Layer (TML) serves as an intermediate level. Here, an absolute time scale is introduced, while an abstract definition of the signal values is preserved. Hence, TML lies between the aforementioned abstraction levels PFL and SFL.

System behaviour models have the same syntactic structure on all above mentioned levels, and the representation of the signals is the main difference. A *system* is defined by the set of *behaviours* it contains. Each of these *behaviours* is represented by input and output *signals*, and the *interdependencies* between them. Here, *signals* are no variables, but graphs, functions, or other representations of the real signals behaviour. Thus, we have to adjust the definition of *signals* to the level of abstraction we want to achieve.

A system's model consists of a finite set of behaviours, where a behaviour is a triple of one set of inputs, one set of outputs, and the interdependency relation between them. Assuming suitable definitions for input signals, output signals, and interdependencies given (we will come to this further down), we define:

Definition 4.1 (System)**.** A system S is a finite set of behaviours $S = \{\mathcal{B}_1, \dots, \mathcal{B}_k \mid k \in \mathbb{N}^+\}$ where all behaviours have the same number of input signals and output signals.

Definition 4.2 (Behaviour)**.** A behaviour is a triple $\mathcal{B} = (I, O, \chi)$, where I is a set of m input signals, O is a set of n output signals (where $m, n \in \mathbb{N}, n > 0$), and χ a finite set of interdependencies between elements of I and O.

To make the abstraction level explicit, we indicate the level L as superscript and write S^{L}, and likewise for behaviours, inputs, outputs and interdependencies.

In Def. 4.1 it becomes clear that we model every system as a black box and the view on the system's behaviour is a static one. Only the defined inputs and outputs can be treated, unknown inputs cannot be processed; this justifies the interpretation of system models and behaviours as test case definition. All these possible behaviours are composed in a set, representing the system model. The intended interpretation as test case definition is also the reason why we demand the set of behaviours to be finite. Each of these behaviours contains two sets of signals, representing inputs and outputs, resp. and the interdependency relation in between the signals' components (cf. Def. 4.2). This interdependency allows us to model causalities which may exist between the input and output signals of a system. After explaining the general structure of the behavioural models, we proceed with the signals and interdependencies on the aforementioned abstraction levels. First, we define the syntax of signals on each level formally; a detailed discussion of the components and the semantics follows afterwards.

Definition 4.3 (Signal)**.** Given two finite sets E of events and P of phases, let

- $\bar{\mathbb{R}}^+ = \mathbb{R}^+ \cup \infty$ be the set of positive reals including infinity,

- $\mathfrak{L}_E$ and $\mathfrak{L}_P$ be two sets of labels for events and phases, resp., and
- $\mathfrak{F}$ be the set of all continuous functions with one real variable.

Additionally let $T = (0, t_1, \ldots, t_l)$ be a finite, strictly ordered sequence of $t_i \in \bar{\mathbb{R}}^+$, i.e. $t_i < t_{i+1}$.
A **signal on PFL** is a tuple

$$\mathcal{Y}^{\mathrm{PFL}} := \left(E, P, \sigma, \theta, \lambda_E^{\mathrm{PFL}}, \lambda_P^{\mathrm{PFL}}, e_0, e_\perp\right) \tag{4.1}$$

where

- $e_0, e_\perp \in E$ are start and end events,
- $\sigma : E \setminus \{e_\perp\} \to P$ and $\theta : P \to E \setminus \{e_0\}$ are two bijective mappings,
- $\lambda_E^{\mathrm{PFL}} : E \to \mathfrak{L}_E$ and $\lambda_P^{\mathrm{PFL}} : P \to \mathfrak{L}_P$ are labelling functions.

A **signal on TML** is a tuple

$$\mathcal{Y}^{\,\mathrm{TML}} := \left(T, \lambda_E^{\mathrm{TML}}, \lambda_P^{\mathrm{TML}}\right) \tag{4.2}$$

with the labelling functions $\lambda_E^{\mathrm{TML}} : T \to \mathfrak{L}_E$ and $\lambda_P^{\mathrm{TML}} : T \setminus \{t_l\} \to \mathfrak{L}_P$.
A **signal on SFL** is a tuple

$$\mathcal{Y}^{\mathrm{SFL}} := \left(T, \lambda^{\mathrm{SFL}}\right) \tag{4.3}$$

with $\lambda^{\mathrm{SFL}} : T \setminus \{t_l\} \to \mathfrak{F}$ assigning a function to each interval.

Table 4.1 sums up the diverse elements of the behavioural models.

Let us discuss the signals on SFL first, which, amongst the signals defined above, are the most similar to mathematical descriptions of signals. The signals on are the nearest to usual mathematical descriptions of signals we discuss them first. On this abstraction level, the signal models are quite comparable to the actually measured signals, as the time intervals between two elements of T are labeled with a function of the form $y = f(t)$. The complete signal is represented by a sequence of function fragments. As each of these function fragments has a unique value at each point, and their intervals do not overlap, the complete signal is fully und uniquely defined on this level. This justifies the classification of measured signal data as SFL signals, and the use of a set of measurement data for modelling a system.

Moving on to the signal models on TML, we take an abstracting step: the intervals are not labeled with functions anymore, but with elements of the set $\mathfrak{L}_P$ instead. This set contains natural language representations of mathematical functions' properties, specifically properties that have a duration, e.g. function values in a specific interval,

		Level		
		Property Flow	Time Metric	Signal Function
System	S	$\{\mathcal{B}_1, \ldots, \mathcal{B}_k \mid k \in \mathbb{N}^+\}$		
Behaviour	$\mathcal{B}$	(I, O, χ)		
Signal	$\mathcal{Y}$	$\left(E, P, \sigma, \theta, \lambda_E^{\mathrm{PFL}}, \lambda_P^{\mathrm{PFL}}, e_0, e_\perp\right)$	$\left(T, \lambda_E^{\mathrm{TML}}, \lambda_P^{\mathrm{TML}}\right)$	$\left(T, \lambda^{\mathrm{SFL}}\right)$

Table 4.1.: Overview on the elements of behavioural models. System, behaviour and signal are defined in Def. 4.1, 4.2, and 4.3, resp.

function gradients, etc.; we call such characteristics *phases*. The elements of $\mathfrak{L}_P$ can vary for different application domains. $\mathfrak{L}_P$ can hold whatever is suitable for the application domain, and will typically have information such as ‘a constant between 0 and 1’ or ‘a slope between 1 and 3’, for example. These labels can also include information on the duration, although this is not necessary on the TML. However, we will see that this proves to be helpful here, and even more on the more abstract PFL.

$\mathfrak{L}_E$’s elements are also natural language descriptions of mathematical functions’ characteristics, but in contrast to $\mathfrak{L}_P$ they represent *events*, i.e. characteristics that take place instantaneously, e.g. maxima, jumps, and changes of phases. Event labels can contain information on time. However, this information does not refer to the duration, but to the allowed times of occurrence instead. As with the phases, also the possible elements of the events are domain-specific, thus no further specification of them is required here. Note that we cannot define exact semantics of signals on TML, but it should become obvious that in general at one specific point in time there is not only one exact value specified but other intervals of possible values.

The set of labels $\mathfrak{L}_P$ is also used in the definition of signals on PFL along with $\mathfrak{L}_E$. These labels are attached to the elements of a bipartite graph: E and P are the sets that contain vertices representing events and phases, respectively. As we

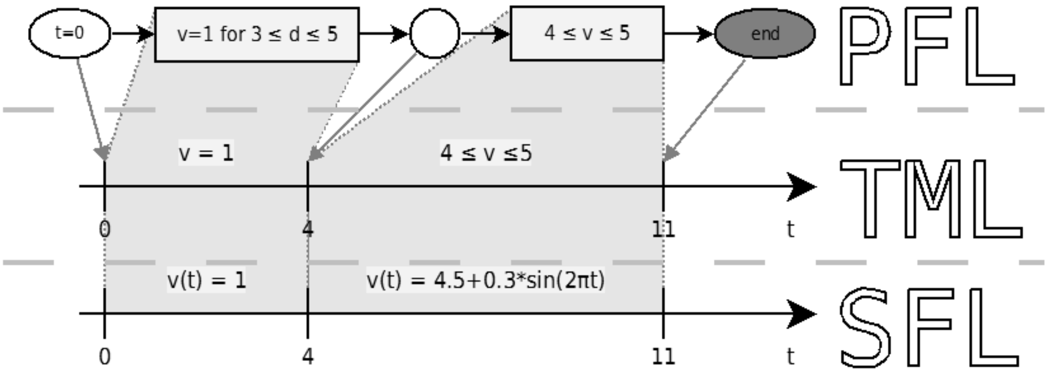

Figure 4.1.: Abstraction levels and their associations.

can see in the definition of σ and θ, every vertex has at most one predecessor and one successor, with the exception of the start and end events e_0 and $e_\perp$. Thus the vertices are connected to each other forming a linear sequence. Each vertex is labeled with a description of the mathematical properties, including possible specifications on the start time intervals or the durations. Consequently, we do not have any information on absolute time, but only relative one depending on the labels of the vertices. Hence, on this level, ambiguities – i.e. nondeterminism in the signal – are not only limited to the function values, but can also occur in the timing of the properties as well.

The example in Fig. 4.1 illustrates how signals on the different abstraction levels can be associated. Events on PFL are associated with elements of T on TML and SFL, while phases are associated with intervals between two succeeding elements in T. Events' and phases' labels are linked to the mathematical functions used on SFL to describe the exact value of the signal. In particular, we can order the abstraction levels, PFL being the most abstract one, SFL the least, and TML lying in between.

One element in the behaviours' definition 4.2 is still undefined, these are the interdependencies. In PFL, they can be used to represent temporal dependencies between events and phases of two different signals, e.g. an input and an output signal. Thus, interdependencies are syntactically additional transitions which connect vertices of different signals of one behaviour. Since they carry temporal information, they are labelled.

Definition 4.4 (Interdependency). Let $I = \{\mathcal{I}_1, \ldots, \mathcal{I}_m\}$ be a set of input signals, and $O = \{\mathcal{O}_1, \ldots, \mathcal{O}_n\}$ be a set of output signals on PFL. Let E_i^I ($i \in \{1, \ldots, m\}$) and E_j^O ($j \in \{1, \ldots, n\}$), resp. be their event sets (cf. Def. 4.3). An **interdependency**

for these inputs and outputs on PFL is a relation

$$\chi^{\mathrm{PFL}} \subseteq \bigcup_{i=1}^{m} E_i^I \times \bigcup_{j=1}^{n} E_j^O \times \bar{\mathbb{R}}^+ \times \bar{\mathbb{R}}^+ \tag{4.4}$$

Interdependencies connect events of input and output signals and represent causalities. They express the fact that an event has to occur in a specified time after the occurrence of another event in an input. The lower and upper limit for the admissible time points of the interdependencies are represented by the two positive real numbers in the definition.

One could define them for the more accurate levels, too, but there the interdependencies do not have to be labelled. The absolute time axis resolves all temporal interdependencies, thus this syntactical construct is not necessary on TML or SFL.

For the following definition of semantics we will introduce the notion of *family of functions* in a semiformal way. Let us take a set of functions defined on the same domain. This set is a family of functions, if the function values, the values of the first and second derivative meet the same requirements. One possible definition of these families can be based on a finite set of *reference functions*, and on a criterion that decides whether two functions lie close to each other.

Definition 4.5 (Family of functions)**.** Given a finite set of funtions F and a criterion to check closeness of functions $\Phi(x, y)$. F and Φ define a *family of functions* $\mathfrak{F}$ as follows:

$$f \in \mathfrak{F} \Leftrightarrow \exists g \in F : (f, g) \models \Phi(x, y) \tag{4.5}$$

A more formal definition of this concept would require an elaboration on the details of the requirement definitions, and the closeness criterion. For instance, we can examine whether their values' are correlated or whether the squared difference of the values for same inputs lies within predefined tolerances. An example for a family of functions are all monotonic rising functions with the first derivative between 0.9 and 1.1, and second derivative between -0.1 and 0.1.

The idea of our semantics' definition is quite straightforward: a behaviour induces a number of function families, one family for each input signal and each output signal. Each family consists of functions over the positive real numbers as time domain. The values of the function depend on the behaviour itself, and will often be real values as well, e.g. when the function represent velocity, distance, pressure, etc. Each of the function families defining a behaviour's semantics is an admissible evolution of the behaviour's input signals and output signals.

Definition 4.6 (Semantics)**.** Let $\mathcal{B} = (I, O, \chi)$ be a behaviour, where $I = \{\mathcal{I}_1, \ldots, \mathcal{I}_m\}$ and $O = \{\mathcal{O}_1, \ldots, \mathcal{O}_n\}$. Let $V_1^I, \ldots, V_m^I$, $V_1^O, \ldots, V_n^O$ be suitable value domains for the inputs and outputs, resp. Then the semantics of $\mathcal{B}$, written as $\mathfrak{b}(\mathcal{B})$, is a set of $n+m$ families of functions meeting the properties specified in the behaviour's signals, where each family consists of functions $\mathbb{R}^+ \to V_i^I$ and $\mathbb{R}^+ \to V_j^O$ resp.

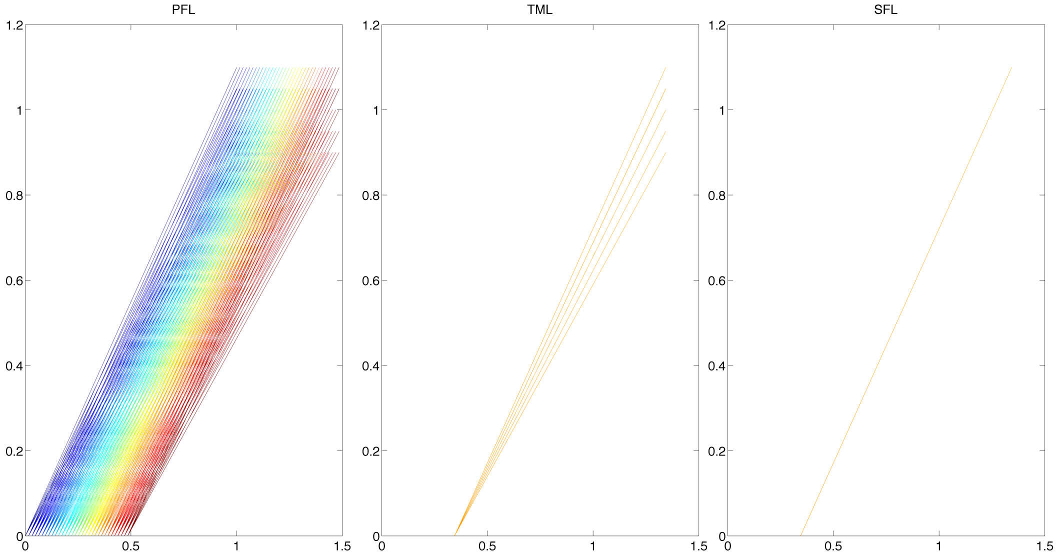

Figure 4.2.: Elements of the semantics of a signal on PFL (left), TML (center), and SFL (right). The signal stays constant for a certain time, and then starts to rise. For better readability, the plots are coloured w.r.t. the start time of the slope.

Let us illustrate the notion of semantics with an example using a minimalistic system model S^{L} on each layer L. The system S^{L} contains one behaviour $\mathcal{B}^{\mathrm{L}}$ with one output signal $O^{\mathrm{L}} = \left\{\mathcal{O}^{\mathrm{L}}\right\}$, $I = \chi = \emptyset$. Let $\mathcal{O}^{\mathrm{PFL}} = \left(E, P, \sigma, \theta, \lambda_E^{\mathrm{PFL}}, \lambda_P^{\mathrm{PFL}}, e_0, e_\perp\right)$ with

- $E = \{e_0, e_1, e_\perp\}$,
- $P = \{p_1, p_2\}$,
- $\sigma = \{(e_0, p_1), (e_1, p_2)\}$,
- $\theta = \{(p_1, e_1), (p_2, e_\perp)\}$,
- $\lambda_E^{\mathrm{PFL}} = \{(e, \varepsilon) \mid e \in E\}$, ε being the empty label,

- $\lambda_P^{\mathrm{PFL}} = \{(p_1, \text{`}x = 0 \text{ for max. 0.5 second'}), (p_2, \text{`}\dot{x} \in [0.9, 1.1] \text{ for 1 second'})\}$

This definition describes a signal that equals 0 for at most 0.5 second, and then rises for 1 second with a rate of approximately 1. Some of the signals meeting this specification are depicted on the left in Fig. 4.2, these signals are elements of $\mathfrak{b}\left(\mathcal{B}^{\mathrm{PFL}}\right)$.

On TML a refined signal $\mathcal{O}^{\mathrm{TML}}$, namely one where $x = 0$ for $\frac{3}{8}$ seconds and then rises with a rate of approximately 1, can be defined as follows. Let $\mathcal{O}^{\mathrm{TML}} = \left(T, \lambda_E^{\mathrm{TML}}, \lambda_P^{\mathrm{TML}}\right)$ with

- $T = \left(0, \frac{3}{8}, \frac{11}{8}\right)$,
- $\lambda_E^{\mathrm{TML}} = \{(t, \varepsilon) \mid t \in T\}$, ε being the empty word,
- $\lambda_P^{\mathrm{TML}} = \left\{(0, \text{`}x = 0 \text{ for max. 0.5 second'}), \left(\frac{3}{8}, \text{`}\dot{x} \in [0.9, 1.1] \text{ for 1 second'}\right)\right\}$

Some of the resulting signals are depicted in the center plot in Fig. 4.2. Analogously, those are also elements of $\mathfrak{b}\left(\mathcal{B}^{\mathrm{TML}}\right)$.

Finally, a further refined $\mathcal{O}^{\mathrm{SFL}} = \left\{T, \lambda^{\mathrm{SFL}}\right\}$ can be defined using T as above:

- $T = \left(0, \frac{3}{8}, \frac{11}{8}\right)$,
- $\lambda^{\mathrm{SFL}} = \left\{(0, x = 0), \left(\frac{3}{8}, x = 1.1t\right)\right\}$

On the right side in Fig. 4.2 this function is depicted; it is the only element in $\mathfrak{b}\left(\mathcal{B}^{\mathrm{SFL}}\right)$. Fig. 4.2 illustrates, how the refined syntactic definition lead to 'tighter' semantics.

Note that as the behaviour is a domain-specific description, there is no generic way to generate (or even compute) the semantic of a behaviour. In fact, it is also not obvious how to decide for a given function family whether it belongs to the semantics of the behaviour or not. We illustrate the problem with the following example.

Assume a system with just one behaviour consisting of one input acceleration and one output velocity. From domain-knowledge, we infer that acceleration is the first derivative of velocity. The abstract model defines the acceleration such that after eight seconds the system will be accelerated with $a = 1\ \mathrm{m} \cdot \mathrm{s}^{-2}$ for at least three and at most six seconds, and there is no acceleration before or after. Thus, for the acceleration, any function $f_a(t)$ with

$$f_a(t) = \begin{cases} 1 & 8 < t \leq t_0 \text{ with } t_0 \in [11, 14] \\ 0 & 0 \text{ else} \end{cases}$$

qualifies. Further, for the speed, any function $f_s(t)$ that

1. starts with some value and stays constant for the first eight seconds,
2. then has a slope of one as long as the acceleration is one, and
3. stays constant afterwards,

qualifies as the velocity function in the family. Although the characteristics stated above can be formalized, the crucial information to put those up comes from domain-knowledge: acceleration is velocity's first derivative.

Even in this simple example, it is clear that there is no straightforward computational generation of the semantics. It is also not obvious how we can derive a algorithm that decides for a given arbitrary family of acceleration functions and one of velocity functions whether both families are in the semantics of the behaviour – although in this simple case, we have in fact a method for this. In general, such a method must somehow find suitable candidates where to split the given functions into parts that need to be checked for the different properties of the abstract model. If we can guess these time points, then it will often be possible to define algorithms for checking whether the function fragments are evolutions that are correct w.r.t. the specified properties. Our method is mainly based on 'guessing' these time points but using a reference signal and cross correlation.

4.2. Association to Automata

The system modelling technique introduced in this chapter resembles some automata models, e.g. finite state machines, real time and hybrid automata, etc. (cf. Sec. 3.1.1). In fact one can argue that the model of one signal, regardless of the abstraction level, is an automaton in which each state has at most one entering and at most one exiting edge Therefore, the signal could also be regarded as an automaton's run. We will discuss this view in detail for PFL, TML and SFL now, starting with PFL.

From Def. 4.3 we learn that a signal on PFL is modelled as an alternating sequence of two kinds of vertices representing phases and events. We will show that we can associate a PFL signal $\mathcal{Y}^{\mathrm{PFL}} := \left(E, P, \sigma, \theta, \lambda_E^{\mathrm{PFL}}, \lambda_P^{\mathrm{PFL}}, e_0, e_\perp\right)$ to a NFA and one of its runs, resp. For this we construct an automaton $\mathcal{A} = (V, \Sigma, \delta, v_0, F)$ such, that

- $V = P \cup E$,
- $\Sigma = \lambda_E^{\mathrm{PFL}} \cup \lambda_P^{\mathrm{PFL}}$,
- $v_0 = e_0$, and
- $F = \{e_\perp\}$.

The automaton's transition function δ is constructed based on the transitions σ and θ in such a way that

- δ maps the same events to phases as σ and the same phases to events as θ, and
- the input symbol serving as parameter for δ equals the label of the target phase and event, resp.

The result of this operation is a 'linear automaton', where each vertex has exactly one successor, except the accepting state which has none. This automaton accepts exactly one word consisting of the labels of the PFL signal.

On TML, a signal is modelled as a sequence of points in time, with intervals and points in time labelled with the phase and event descriptions (cf. Def. 4.3). Analogously to PFL signals, we can transform these into automata. Here, a TML signal is mapped to a timed automaton. For this purpose, we introduce one clock that represents the continuous time in the TML signal, the points in time defined there are used to construct the guards for the transition function. With every transition this clock is set back again. In comparison to the transformation of a PFL signal, we have to change the input alphabet because in the semantics of timed automata, we cannot capture instantaneous events in locations. While the NFA's Σ is the union of both labelling sets, for the timed automaton we have to concatenate one event label and one following phase label. By this, we guarantee that the automaton has exactly one run, accepting the sequence of labels at the points in time defined in the TML signal.

Similar to the former transformations, a SFL signal is mapped to a hybrid automaton. We can use the clock introduced above for transition guards, and add location invariants that are derived from the set $\mathfrak{F}$ in Def. 4.3. In contrast to the former automata we do not need an input alphabet anymore, since the transitions are taken at the defined points in time captured in the guards. Again, we obtain a linear automaton, every location has at most one successor. Additionally, all continuous variable values are well defined at every point in time.

Above, we chose three different automaton models to demonstrate this association, but alternatively we are able to use only hybrid automata. For this we would have to evaluate the labelling of phases, etc. to construct invariants, guards, etc. In the end we would obtain the same hybrid automaton for a SFL signal as above, for a TML signal the guards would still incorporate exact clock values, while for PFL the guards would consist of intervals in time. We will not elaborate on the representation of all abstraction levels' signal models as hybrid automata, since it is not significant for the further topics.

Essentially, a signal model on an arbitrary abstraction level can be associated to a linear automaton, where each state and location, resp. has exactly one successor, except the finite state or location. In fact, these automata represent one path in a more general automaton, we will discuss this problem in the following paragraph.

With two operations we can transform a system's model defined according to Def. 4.1 to automata. One operation is the composition of one 'signal' automaton per signal in a behaviour to obtain concurrent automata representing this behaviour. The other is merging all automata of one specific signal to one automaton accepting all defined property sequences in the PFL signal. After executing these operations in an arbitrary order, we would obtain a complex collection of concurrent automata. We can expect that correctly building such a collection from scratch is a difficult and time consuming task, as would be maintaining it afterwards. Therefore, we represent a system's behaviour with the abstract modelling technique introduced in Section 4.1 instead of using the sketched automata representation.

5. Conformance and Similarity

Using the abstract behavioural models, we define a *similarity relation* that compares two systems showing continuous behaviour. In general terms, one can consider two systems to be similar when they behave in a similar way. Hence the proposed definition of similarity of systems uses an analogous similarity relation of behaviours. The abstract modelling of systems which was introduced in Chapter 4 to deal with the limitations of an exact time-value description, serves as a tool for these relations. We will compare accurate signal data obtained in test from two development artefacts on the one hand with an abstract behavioural system model on the other hand. If both artefact's signals accord to the abstract model, the artefacts are considered to be similar. The most abstract level, PFL, will serve as comparison level, while the accurate data will be represented on SFL.

The starting point will be a notion that describes which (more exact) behaviours conform to a given abstract behaviour. With this relation, for instance, we can express whether a SFL behaviour conforms to a given PFL behaviour. Using this, we define the similarity relation with respect to an abstract model, before turning to a practically usable method to decide the conformance of continuous behaviours to abstract models.

Let us now define a conformance relation between a more accurate behaviour and a more abstract model. As already stated before, in general we are not able to list all allowed behaviours explicitly. Thus, we need to approach our conformance problem from the lower levels of abstraction: we will check whether the semantics of a more accurate behaviour is contained in the semantics of a more abstract one. As this is in fact a containment relation, we reuse the usual set inclusion symbol for our relation:

Definition 5.1 (Conformance)**.** A behaviour $\mathcal{A}$ conforms to a behaviour $\mathcal{B}$ iff the semantics of $\mathcal{A}$ is contained in the set of behaviours allowed by $\mathcal{B}$, $\mathfrak{b}\left(\mathcal{B}\right)$:

$$\mathcal{A} \subseteq \mathcal{B} \iff \mathfrak{b}\left(\mathcal{A}\right) \subseteq \mathfrak{b}\left(\mathcal{B}\right) \tag{5.1}$$

A behaviour $\mathcal{A}$ conforms to a system S iff there exists a behaviour $\mathcal{B}$ in S such that $\mathcal{A}$ conforms to $\mathcal{B}$:

$$\mathcal{A} \subseteq S \iff \exists \mathcal{B} \in S : \mathcal{A} \subseteq \mathcal{B} \tag{5.2}$$

Although this definition does not specify the abstraction levels of both behaviour and model, we can assume that S should be at least as abstract as $\mathcal{A}$. Nevertheless it is possible to construct a TML and a PFL behaviour in such a way, that the PFL behaviour conforms to the TML behaviour, e.g. by specifying exact points of time for the PFL vertices and 'relaxing' the labels in TML behaviour.

We now use this relation to define the similarity of behaviours with respect to a behavioural model, and extend this definition to allow statements on the similarities of systems in general.

Definition 5.2 (Similarity)**.** Two behaviours $\mathcal{A}$ and $\mathcal{B}$ are similar with respect to a system S iff both behaviours conform to the same behaviour $\mathcal{S}$ of the system S:

$$\mathcal{A} \approx_S \mathcal{B} \iff \exists \mathcal{S} \in S : \mathcal{A} \subseteq \mathcal{S} \wedge \mathcal{B} \subseteq \mathcal{S} \tag{5.3}$$

Two systems A and B are similar with respect to a system S, iff for every continuous behaviour $\mathcal{A}$ of A there exists a continuous behaviour $\mathcal{B}$ of B, such that $\mathcal{A} \approx_S \mathcal{B}$, and vice versa:

$$\begin{aligned} A \approx_S B \iff & \quad \forall \mathcal{A} \in A . \exists \mathcal{B} \in B : \mathcal{A} \approx_S \mathcal{B} \\ & \wedge \forall \mathcal{B} \in B . \exists \mathcal{A} \in A : \mathcal{B} \approx_S \mathcal{A} \end{aligned} \tag{5.4}$$

The conformance relation in Eq. (5.2) of Definition 5.1 suggests that a behaviour conforms to a system, if the behaviour's semantics is included in the semantics of a system's behaviour. Thus, if we have two behaviours $\mathcal{A}$, $\mathcal{B}$ and both of them conform to the same behaviour $\mathcal{S}$ in the system, the sets of functions inferred from $\mathcal{A}$ and $\mathcal{B}$ will have common elements. This justifies the definition which calls them similar with respect to the abstract model. When we extend this definition to systems, we have to make sure that for each behaviour of one system there is a similar behaviour in the other system (cf. Eq. (5.4)).

Let us go back to the example on page 40, where we illustrated the syntax and semantics on the three abstraction levels with help of a simple signal. There, we had defined systems S^{L} consisting of one behaviour $\mathcal{B}^{\mathrm{L}}$ each, each of which in turn consists of one output signal. From the definitions and Figure 4.2 we see that

$$\mathfrak{b}\left(\mathcal{B}^{\mathrm{SFL}}\right) \subseteq \mathfrak{b}\left(\mathcal{B}^{\mathrm{TML}}\right) \subseteq \mathfrak{b}\left(\mathcal{B}^{\mathrm{PFL}}\right) \tag{5.5}$$

and hence

$$\mathcal{B}^{\mathrm{SFL}} \subseteq \mathcal{B}^{\mathrm{TML}} \subseteq \mathcal{B}^{\mathrm{PFL}}. \tag{5.6}$$

So, we can conclude that $\mathcal{B}^{\mathrm{SFL}} \subseteq S^{\mathrm{PFL}}$.
From Equations (5.5) and (5.6) we can infer that

$$\mathcal{B}^{\mathrm{SFL}} \approx_{S^{\mathrm{PFL}}} \mathcal{B}^{\mathrm{TML}} \tag{5.7}$$

and in particular

$$S^{\text{SFL}} \approx_{S^{\text{PFL}}} S^{\text{TML}} \tag{5.8}$$

Now we have the necessary relations to compare systems showing accurate continuous behaviour. When we apply Def. 5.2 to the accurate measured behaviours of two such systems, we can state the similarity with respect to a model on a more abstract level, e.g. PFL. From the preceding definitions we see, that we can break this relation down to the question of conformance of behaviours in Def. 5.1. In consequence, a fitting iteration of a method evaluating the conformance of behaviours as defined is sufficient to answer this question.

Our main goal is to check the similarity of two systems by checking the similarity of their generated behaviours based on the input and output signals. Typically, the information on the signals has the form of sampled continuous data, so we need to connect this data with our definitions. A straightforward way would be checking for a whole sampled signal whether it lies in the semantics of an abstract signal. Although this would work out for models on SFL, this method would not be feasible on higher abstraction levels. Therefore, we have to take a different approach.

At first, we have to associate sampled continuous data with models on higher abstraction levels, in particular on PFL.

Theorem 5.3. *Let $\mathcal{C}$ be a sampled continuous data signal and $\mathcal{P}$ a behaviour on PFL. Then $\mathcal{C} \subseteq \mathcal{P}$ iff there exists an behaviour $\mathcal{R}$ on SFL such that $\mathcal{C} \subseteq \mathfrak{b}(\mathcal{R}) \wedge \mathcal{R} \subseteq \mathcal{P}$.*

Proof. Let $\mathcal{R} \subseteq \mathcal{P}$ be an arbitrary behaviour on SFL. If $\mathcal{C} \subseteq \mathfrak{b}(\mathcal{R})$, then $\mathcal{C} \subseteq \mathfrak{b}(\mathcal{P})$ by Def. 4.6 and 5.1, and therefore $\mathcal{C} \subseteq \mathcal{P}$, and vice versa. ☐

Since this theorem does not give any hints on how to construct a conformance check, it is not sufficient to be used directly for conformance checking. In order to use this theorem effectively, we need to come up with a more constructive method to show conformance. The following lemmas are formulated to associate the different elements in a more detailed way. In the following let us assume that we can represent the data of a sampled continuous signal as

$$\mathcal{C} = \left\{ (t_1, x_1), \dots, (t_h, x_h) \mid t_i \in \bar{\mathbb{R}}^+ \wedge t_i < t_{i+1}, x_i \in \mathbb{R}, h \in \mathbb{N}^+ \right\} \tag{5.9}$$

Lemma 5.4. *Let $\mathcal{C}$ be a sampled continuous data signal and $\mathcal{R} = (T_\mathcal{R}, \lambda_\mathcal{R})$ a signal on SFL. $\mathcal{C} \subseteq \mathcal{R}$ iff there exists a partition $\Pi = \{\pi_1, \dots, \pi_l\}$ of $\mathcal{C}$, such that*

1. $\pi_i = \left\{ (t_j, x_j) \in \mathcal{C} \,\middle|\, t'_j \geq t_j > t'_{j+1} \text{ with } t'_j, t'_{j+1} \in T_\mathcal{R} \right\}$.

2. $x_i = (\lambda_{\mathcal{R}}(t_{1'}))(t_i)$ *holds for all* $(t_i, x_i) \in \pi_r = \{(t_{1'}, x_{1'}), \ldots\}$. $(t_{1'}, x_{1'})$ *denotes the smallest element in* π_r*:* $\forall (t_{k'}, x_{k'}) \in \pi_r : t_{1'} \leq t_{k'}$.

Note that the elements in each π_r get a new enumeration $1', 2', \ldots$. $\lambda_{\mathcal{R}}(t_{1'})$ is one piecewise defined function from $\mathcal{R}$; the according interval in the signal data is represented by π_r.

Proof. First we can state that $\mathfrak{b}(\mathcal{R})$ has one element, which can be inferred from $\mathcal{R}$ directly (cf. Def. 4.3). This one element is a piecewise defined function.

Let us now assume $\mathcal{C} \nsubseteq \mathcal{R}$. Then there is at least one tuple (t_f, x_f), such that it is not equal to the value of $\mathfrak{b}(\mathcal{R})$ at t_f. Let $\lambda_{\mathcal{R}}$ map the interval which holds t_f to λ', then $x_f \neq \lambda'(t_f)$. So, the second condition cannot hold.

Let us now assume that there is a partition Π of $\mathcal{C}$, such that both conditions hold. The consequence is that all the x_i in the tuples of $\mathcal{C}$ are the appropriate evaluations of the functions associated by $\lambda_{\mathcal{R}}$ at each t_f. Thus, they lie within the semantics of $\mathcal{R}$ and therefore $\mathcal{C} \subseteq \mathcal{R}$. ☐

With this lemma we have established a connection between the signal data obtained from measurements, simulation, etc. and the behavioural models defined in Cha. 4. In particular, the lemma shows how to represent such data on SFL. Note that the original continuous signal, that is sampled, may not be equal to the SFL signal and still be considered as conforming due to the sampling.

In the next step we have to connect the system models on SFL and PFL in a similar way. The following lemma will associate a signal on SFL representing the measured data with a signal model on PFL being the comparison instance.

Lemma 5.5. *Let*

- $\mathcal{R} = (T_{\mathcal{R}}, \lambda_{\mathcal{R}})$ *be a signal on SFL, and*
- $\mathcal{P} = (E, P, \sigma, \theta, \lambda_{\mathcal{P},E}, \lambda_{\mathcal{P},P}, e_0, e_\perp)$ *a signal on PFL.*

$\mathcal{R} \subseteq \mathcal{P}$ *iff there is a bijective mapping* $T_{\mathcal{R}} \to E, t_i \mapsto e_i$*, such that*

1. *the elements of* $T_{\mathcal{R}}$ *lie in the intervals allowed for the events,*[1]
2. *the events defined by* $\lambda_{\mathcal{P},E}(e_i)$ *are fulfilled by*
    - $(\lambda_{\mathcal{R}}(t_i))(t_i)$*, and by*
    - $\lim_{t \to t_i} (\lambda_{\mathcal{R}}(t_{i-1}))(t)$,

---

[1] A more formal definition of 'intervals allowed for the events' depends on detailed definitions for event and interdependency labels. We left those to be determined by domain experts. Given detailed definitions, allowed intervals will have to be inferred from the event labels, the property labels and interdependency labels of the SFL model.

and

3. $\lambda_{\mathcal{R}}(t_i) \in \mathfrak{b}(\lambda_{\mathcal{P},P}(\sigma(e_i)))$

Proof. First recall from the proof of Lem. 5.4 that the semantics $\mathfrak{b}(\mathcal{R})$ consists of one element. On the other hand, the semantics of $\mathcal{P}$ are represented by a family of functions (cf. Def. 4.6).

Let us first assume $\mathcal{R} \subseteq \mathcal{P}$. Then we know that the semantics of $\mathcal{R}$ are an element of $\mathfrak{b}(\mathcal{P})$. In particular we can conclude, that $\mathcal{R}$ has to have certain characteristics. Those characteristics can be mapped to certain characteristics of the PFL signal whose semantics contain the semantics of the SFL signal. First, the points of time in $T_{\mathcal{R}}$ separate intervals of time on which functions are defined. Thus, the elements of $T_{\mathcal{R}}$ have to be mapped bijectively on the elements of E. Since the semantics of the SFL signal lies within the semantics of the PFL signal, timing and value constraints found in the labels of the PFL signal's vertices are met. This fact is expressed by the three restrictions in the lemma, the first one relates to the timing, while the other two relate to the event and phase labels.

Let us now assume that there is a bijective mapping $T_{\mathcal{R}} \to E$ satisfying the three constraints. Then the SFL signal meets the timing and value constraints specified in the labels of the PFL signal. In particular each piece of the piecewise defined function resulting from the SFL signal meets the constraints of the according labels. Thus, we can conclude that the whole semantics $\mathfrak{b}(\mathcal{R})$ meets those constraints and therefore is an element of $\mathfrak{b}(\mathcal{P})$, and $\mathcal{R} \subseteq \mathcal{P}$. ☐

From those two lemmas we can deduct an algorithm allowing us to evaluate the conformance of sampled signal data with an abstract model. We present this process in Sec. 7 after showing that we now have the necessary foundations to check conformance of systems and therefore, also the similarity of two of them with respect to an abstract model. For this, we conduct below corollaries from Lemma 5.5 in an inductive way. In both corollaries bijectivity and surjectivity, resp. are crucial: With a bijection we assure that the number of input and output signals is equal, while the surjective mapping allows several less concrete behaviours conforming one more abstract one.

Corollary 5.6. *Let* $\mathcal{R} = (I_{\mathcal{R}}, O_{\mathcal{R}}, \chi_{\mathcal{R}})$ *and* $\mathcal{P} = (I_{\mathcal{P}}, O_{\mathcal{P}}, \chi_{\mathcal{P}})$ *be now behaviours on SFL and PFL, resp.* $\mathcal{R} \subseteq \mathcal{P}$ *iff there exist two bijections* $I_{\mathcal{R}} \to I_{\mathcal{P}}$ *and* $O_{\mathcal{R}} \to O_{\mathcal{P}}$*, resp. such that* $\mathcal{Y} \subseteq \mathcal{X}$ *holds for all* $\mathcal{Y} \mapsto \mathcal{X}$*.* $\mathcal{Y} \in I_{\mathcal{R}}$*,* $\mathcal{X} \in I_{\mathcal{P}}$ *and* $\mathcal{Y} \in O_{\mathcal{R}}$*,* $\mathcal{X} \in O_{\mathcal{P}}$*, resp.*

Corollary 5.7. *Let* S *and* P *be two systems on SFL and PFL, resp.* $S \subseteq P$ *iff there is a surjective mapping* $S \mapsto P$*, such that Cor. 5.6 holds.*

These two corollaries complete the formal definition of the similarity relation with respect to an abstract model. In Lemma 5.5, we left open the question how to determine whether a SFL function lies in a PFL label's semantics: When does $\lambda_{\mathcal{S}} \in \mathfrak{b}\left(\lambda_{\mathcal{P},P}\right)$ hold? When addressing this problem we have to keep in mind that there is no straightforward way to generate every exact signal function allowed by PFL semantics. Nonetheless, we are able to obtain a subset of *reference signals* on SFL, which exhibits typical features of the PFL label. By restricting the cardinality of such a subset to be finite we can compare the signal resulting from a SFL function with every reference signal in finite time.

We already introduced such an idea in the definition of families of functions (Def. 4.5). Based on this, we can state the following.

Lemma 5.8. *Given*

- *a criterion for closeness* $\Phi\left(x,y\right)$ *of two SFL functions,*
- *a finite reference signal set* R *of a PFL phase label* $\lambda_{\mathcal{P},P}$*, and*
- *a signal* $\mathcal{S}$ *on SFL defined by a SFL function label* $\lambda_{\mathcal{S}}$*.*

Let $\mathfrak{R}$ *be the family of functions defined by* R *and* Φ*.*

$$\begin{aligned} \exists \mathcal{R} \in R : \left(\mathcal{S}, \mathcal{R}\right) \models \Phi\left(x,y\right) \Leftrightarrow \mathcal{S} \in \mathfrak{R} \\ \Rightarrow \lambda_{\mathcal{S}} \in \mathfrak{b}\left(\lambda_{\mathcal{P},P}\right) \end{aligned} \tag{5.10}$$

Before we proof the lemma, let us remark that the labels used in SFL represent functions which can be interpreted as signals (cf. Def. 4.3, (4.3)).

Proof. Since the labels in SFL represent functions we can apply a criterion for closeness Φ. Given a finite set of reference signals, we can decide whether there is one reference signal that is close to an arbitrary signal $\mathcal{S}$ defined by $\lambda_{\mathcal{S}}$. According to Def. 4.5, S belongs to the family of functions defined by R and Φ.

Since the elements of R are reference signals of a phase label $\lambda_{\mathcal{P},P}$, functions that are elements of the family of functions $\mathfrak{R}$ are elements of the semantics of $\lambda_{\mathcal{P},P}$ (cf. Def. 4.6). $\square$

Note that we cannot decide whether a SFL signal is an element of the semantics of an PFL phase label; the choice of both closeness criterion and set of reference signals have a crucial impact on the construction of the family of functions. Depending on how the PFL labels for phases and events are formulated, it may be necessary to take a sequence of these labels into account.

Part III.

Implementation – The Approach in Practice

6. Implementing Abstract Behavioural Models

Originally, the author of this work developed abstract behavioural modelling hierarchy in the scope of the ZAMOMO project from 2006 till 2009. That project dealt with the problem of integrating model-based approaches in control engineering and software engineering with focus on small and medium sized enterprises. As already pointed out in [37], the found solutions were implemented in such a way that only a few tools are needed to make use of the solutions. For several reasons a tool chain with few elements is helpful for such enterprises. The most important aspects are lower costs when purchasing software tools, and less effort maintaining and using the tool chain.

Because the hierarchy connects the requirement management and the controller design process, the project partners involved in the according working package decided to use the tools used for those tasks. In this project the group working on requirement and system modelling implemented their solutions in a tool based on OpenOME, a framework for agent-based modelling [1]. The OpenOME approach uses the following additional tools:

- Telos is used for implementing new frameworks for OpenOME [34],
- ConceptBase is a database where the model information in OpenOME is stored [28].

One aim in ZAMOMO was to connect requirement engineering as known from software development with modelling methods used in control engineering with an automotive focus (cf. [15, 46–49]). The mentioned tools used for requirement capturing and engineering were connected to those used in controller design, mainly MATLAB/Simulink. Hence, the abstract modelling hierarchy was implemented in the same tool environment to establish the connection between those two domains.

In [37], the authors implement the more abstract modelling layers PFL and TML with OpenOME, while for the more accurate SFL and TFL[1], Simulink is used. In particular for the TFL, the choice was easy, since Simulink is a tool incorporating the use of transfer functions both implicitly and explicitly. The most abstract layer PFL was implemented using Telos with OpenOME as front end. Furthermore, with the Telos/ConceptBase environment the authors were able to create new models and gather information on existing ones with little effort.

---

[1] TFL is not used in this work and therefore is not introduced in Cha. 4.

The intermediate layers were implemented in both Simulink and OpenOME. The SFL models consist of lookup tables representing the piecewise functions, the switching from one interval to the following interval is achieved by using Simulink's `Clock` block and blocks representing logical and relational functions. The TML links both tool domains and here, the change from OpenOME to Simulink is performed. In OpenOME we used a customised version of the PFL's framework which includes such features as absolute timing information, removal of interdependencies, etc. Using ConceptBase, we can transform a PFL model to a TML model in such a way, that the latter's semantics is included in the first one's.

On the one hand, as pointed out in Chapter 4, the semantics of an abstract signal's graph on PFL and TML induce a set of possible accurate signals. So, there is no unambiguous mapping of an abstract signal to exactly one accurate one. On the other hand, in Simulink one is not able to represent several signals in one simulation run. Thus, we have used a different approach to model TML signals in Simulink. Simulink provides a block library Model Verification that facilitates checks of signal values, e.g. whether a signal is zero, or lies within a certain interval. We use these blocks to validate whether a generated signal meets the property definitions of the OpenOME implemented model. The switching between the intervals is achieved similarly as in the SFL implementation (cf. Fig. 6.1).

For modelling requirements and linking those to the design process, the implementation in two tool domains, OpenOME and MATLAB/Simulink, was necessary. But this is not very suitable for test case modelling and analysis purposes. Here, we have measurements from different sources which can be analysed using mathematical tools such as MATLAB. For instance, the algorithm that determines the conformance of models is implemented in MATLAB (cf. Cha. 7). Thus, we will take a look at an object oriented approach for the implementation of abstract behavioural models in MATLAB.

As is demonstrated in [40], the modelling hierarchy can be made feasible with help of MATLAB/Simulink. Here, the formal concepts introduced before are applied. We have to implement both abstract models and algorithms to compute the similarity of systems. Since we are dealing with embedded systems here, it is straightforward to choose an environment commonly used in this domain. As mentioned in Chapter 3, MATLAB is a common tool suite in the embedded software domain. A conceptual description about how to implement the abstract modelling of systems in MATLAB will be presented in the following paragraphs.

For the abstract behavioural models we choose an object-oriented implementation; the UML class diagram of the abstract models is depicted in Fig. 6.2. The classes are defined according to the formal definitions, thus an instance of `Abstract behavioural system model` is composed of at least one `behaviour` object. Accordinging to Def. 4.2, each behaviour contains an arbitrary number of input signals and at least one output signal – `input_signal` and `output_signal`. Interdependencies between input and output signals are components of a behaviour, too, because the interdependencies are links between events and phases of different signals – `propertyflow_interdepency`. As we already have reasoned in Chapter 4, they are only used for signals on PFL.

Input and output signal representing classes are derived from the class `signal`. Each signal can contain representations of any abstraction level, i.e. on PFL build up by events and phases, and absolute time signals on TML and SFL. Each `signal` object contains therefore a `propertyflow` object representing a signal on PFL, and is composed of `event` and `phase` objects. The according classes are derived from an abstract `property` class and inherit also two additional classes that are not defined in Def. 4.3:

1. `referencesignal` object containing an accurate representation for the property,
2. `occurrence` object containing the shifts where the property was found in a signal's data.

Both classes are used for the computations of the property identification algorithm (cf. Sec. 7.2). The transitions between events and phases are established within the `propertyflow` class.

The `signal` class also contains signal representations on the layers that use an absolute time scale, TML (`timemetric`) and SFL (`signalfunction`). These can be used for intermediate steps when refining a PFL representation, and as exact representation, resp. Observe that one `signal` can contain several objects of the classes `timemetric` and `signalfunction`. This is a consequence of the definition of a system's semantics in Def. 4.6, a signal induces a set of mathematical functions.

In general, an `abstract behavioural system model` object will contain at least the PFL representations of the behaviours. Hence it can serve as the abstract model that one system should conform to while the measured signals can be represented in a `signalfunction` object. The conformance check algorithm can now check property by property, whether it can be found in the measured signal data, and also determine if it occurs at the right time and in the right order. Here, the storage class `occurrence` is used. The algorithm will use the information in the `referencesignal` objects to search for the property.

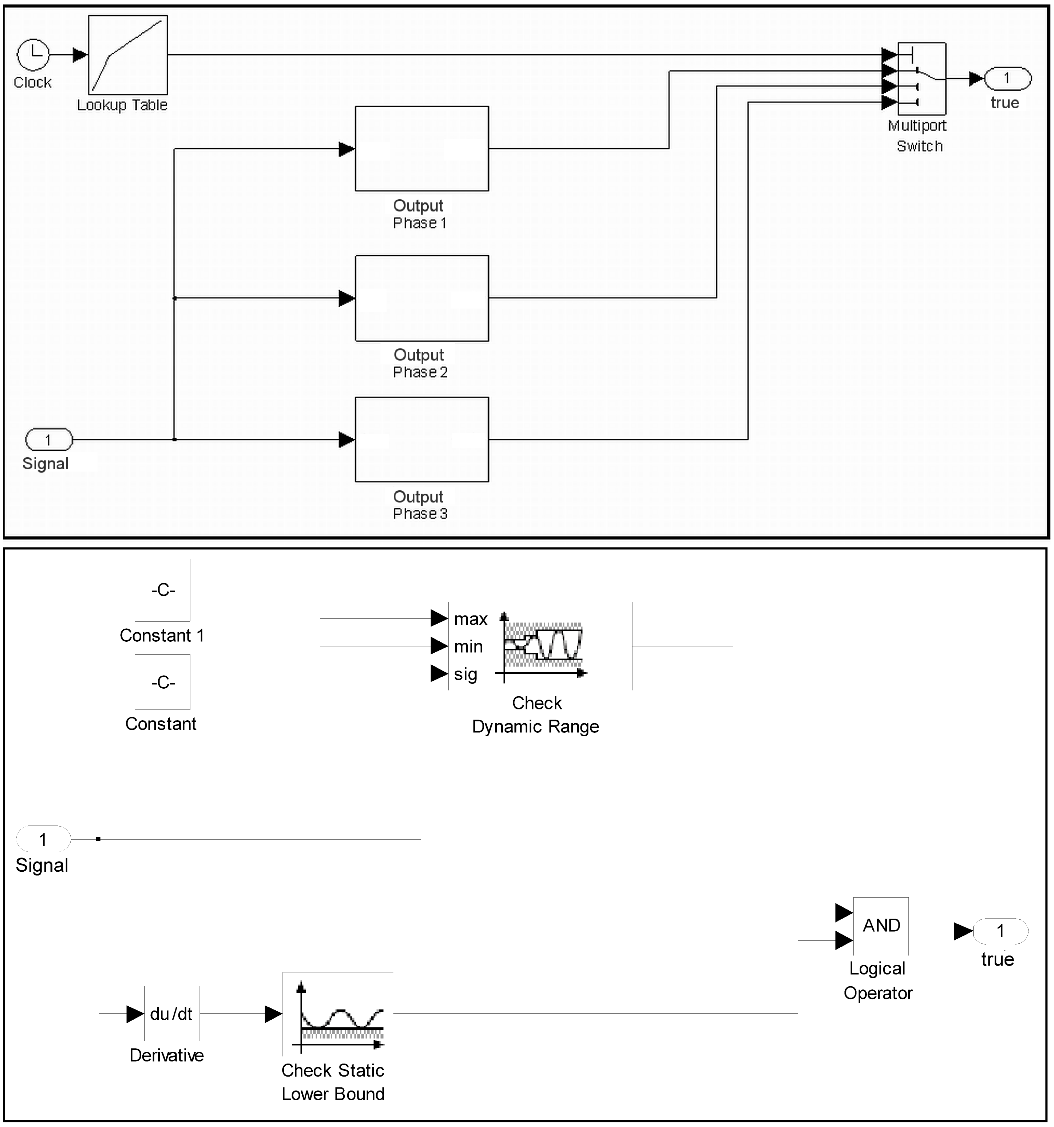
Clock
Lookup Table
Multiport Switch
1
true
Output Phase 1
Output Phase 2
Output Phase 3
1
Signal
-C-
Constant 1
-C-
Constant
max
min
sig
Check Dynamic Range
1
Signal
AND
Logical Operator
1
true
du/dt
Derivative
Check Static Lower Bound

Figure 6.1.: Simulink model on TML as implemented in the ZAMOMO project. Top: The overall model for a signal with three phases, the switching between phases is achieved by `Clock` and `Mulitport Switch`. Bottom: One of the phases subsystem is shown in detail.

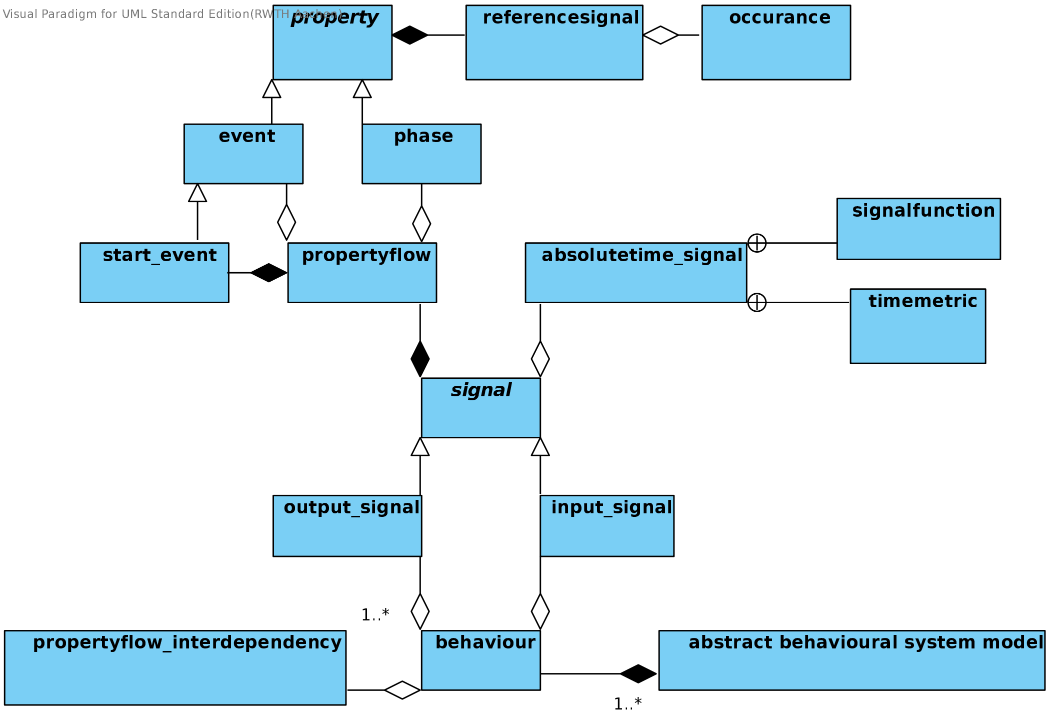
Visual Paradigm for UML Standard Edition(RWTH Aachen)
property
referencesignal
occurance
event
phase
signalfunction
start_event
propertyflow
absolutetime_signal
timemetric
signal
output_signal
input_signal
1..*
propertyflow_interdependency
behaviour
abstract behavioural system model
1..*

Figure 6.2.: UML class diagram of abstract behavioural models.

7. Conformance Checking

Together with the corollaries, Lemmata 5.4 and 5.5 describe a constructive approach to checking the conformance of a signal's sampled data with a PFL signal. We can validate the conformance of continuous signal data with a PFL signal by inductively constructing a reference SFL signal from the PFL signal and checking whether the values of the continuous data are a subset of the resulting function. The problem lies in the generation of the reference signal, since in general the semantics of a PFL signal is an innumerable set.

According to Lemma 5.8, the main idea is to generate reference signals for each property defined in a PFL signal, and search for it in the sampled data. The generated reference signals belong to the SFL, and therefore Lem. 5.4 is applicable. Still, accurate equality of sampled data and reference signal is almost impossible to achieve, therefore we have to examine the continuous data using a closeness criterion. The crucial initial step is to identify times at which the sampled data is so similar to a reference signal, that we can assume the corresponding property being met. Having identified these times of occurrences for all properties we can validate

- whether the order is the same as in the PFL signal, and
- whether the times comply to the times in the abstract signal, too.

If we are able to validate these characteristics, the sampled signal conforms to the PFL signal. Thus, the result is that Lemmata 5.4 and 5.5, and the corollaries are satisfied.

Fig. 7.1 illustrates this process, where for example sampled continuous data is obtained from execution of test cases. In the following, we will focus on three of these steps, namely

1. generation of reference signals,
2. identification of properties in a signal, and
3. validation of the properties' order and timing.

Note that some steps are taken in the modeling domain and in the SuT as well. In the first step, a reference signal is generated on the model side, while on the SuT side an accurate stimuli is generated as input for the test cases (cf. Sec. 7.1). Then, the stimuli are used for executing the test cases on the SuT side, e.g. a simulation

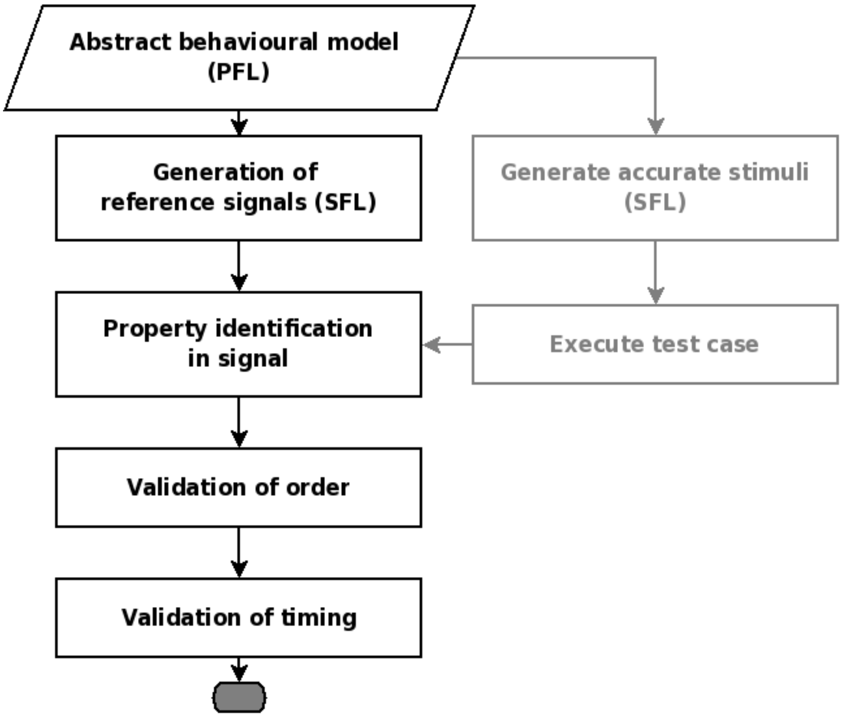
Abstract behavioural model (PFL)
Generation of reference signals (SFL)
Generate accurate stimuli (SFL)
Property identification in signal
Execute test case
Validation of order
Validation of timing

Figure 7.1.: Activities and work flow of conformance checking of continuous signal data with an abstract model; the continuous data origins from test execution.

of a Simulink model, or a software in the loop test with the real target system or similar systems. From the test case execution we obtain sampled signals, i.e. the values of a continuous signal are measured at distinct points of time. Keep in mind that in general the original signal is continuous.

This sampled data has to be compared to the abstract model of the outputs which is done by means of the reference signals. The first part of the analysis is the search for points of time at which a property occurs in a signal (cf. Sec. 7.2). This is the property identification on the model side.

After all properties required by a signal's abstract model have been found, the last step follows. The first part is the validation of the order where we check whether the identified times of the properties' occurrences are in the correct order. If this is the case we can go ahead and validate the properties' timing with respect to other properties of the same and of related signals (Sec. 7.3). In case the timing could be validated, the signal lies in the behaviours allowed by the abstract model. By iterating these steps for every signal of every behaviour we check the conformance of the SuT with respect to the abstract model.

7.1. Generation of Reference Signals

Generation of reference signals of one single property involves steps of refinement with respect to the abstraction levels introduced in Sec. 4. While the properties are already defined on the PFL, we need to represent them on the SFL in order to enable property search in the accurate quasi-continuous signal obtained in the test. Generally speaking, it is obvious that one reference signal per property is not sufficient to cover the semantics of a PFL property; coverage is therefore a very interesting issue, which will have to be explored in future work (Sec. 12). System models on the PFL abstract from both time and value, thus we have to transform these abstract information to accurate ones. This can be done in one step or by taking an intermediate step to the TML, where only time is accurate. For better understanding the process will be described with this intermediate step in detail in the following paragraphs.

As the first step we have to define the sequence of points of time T; on the intervals between its elements the reference signal will be defined by piecewise functions. The information on the corresponding points of time can be extracted from the timing information in the properties' and interdependencies' labels. Since we now have removed the ambiguities with respect to time, we can also remove timing information from the labels, and there is no need to model interdependencies on TML and SFL (cf. page 38). Duration of phases are helpful, though, because it is a characteristic of the property itself, e.g. the duration of a rising signal determines its gradient. At this point we encounter the problem of coverage for the first time, because durations can be specified as intervals over time. Thus, we have to choose some reasonable T out of an uncountable set.

The recently described step results in a representation of a property on the TML. By replacing the natural language description with mathematical formulas we proceed to the SFL. Here we propose to apply Occam's razor, i.e. we keep the formulas as simple as possible, while meeting the required properties. In most cases linear functions and polynomials of low order are sufficient and ease the later step of property identification. Again, we would have to cope with coverage, and have to choose a finite number of functions out of an uncountable set.

It is straightforward to transform the phases in such a manner, but as stated in Lem. 5.5, neighbouring events are necessary to create correct reference signals. Hence, these have to be captured implicitly in representations of the phases. Iterating this process and connecting the SFL representations to complete signals could be used for creating stimuli for test case execution. Note that if "wrong" reference signals are used, it might be impossible to find a property though the continuous data contains it. However we can be sure that every property found with a reference signal also occurs in the sampled signal.

In the context of back-to-back testing we suggest to use the oracle as reference signal generator. For this we assume that exact stimulating signals on SFL already exist, and can be used as inputs for the oracle. The oracle's output is a sampled signal on SFL conforming to the abstract model on PFL, thus all required properties occur in the required order and timing. The user is able to identify these occurrences and mark them. By this, signal snippets are created, which can serve as properties' reference signals. In the long run, a database of reference signals for properties can be build. One possibility to create stimuli would be generic parametrisable signal snippets based on individual domain expert knowledge. This topic exceeds the focus of this work and will be discussed in Future Work (Cha. 12).

7.2. Property Identification in Signal

While in the preceding step we prepared the reference signal for applying Lem. 5.4, we will now use this lemma to obtain the data we need for Lem. 5.5. The identification of properties' occurrences in a signal is a crucial step in our work flow, since the following validation steps rely on well-founded and correct results. For every property expected to be in an obtained signal, two operations are executed.

1. Find points of time at which the property can occur (the first three steps in Fig. 7.2).
2. Check, whether the signal values are correct at these points of time (remaining steps in Fig 7.2).

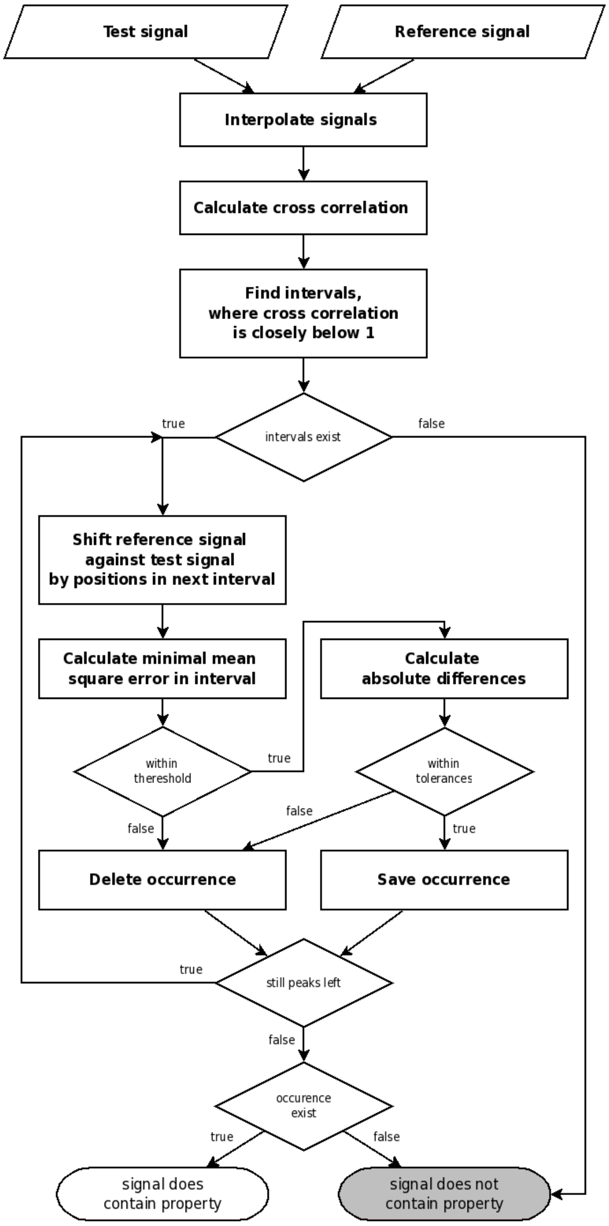
Test signal
Reference signal
Interpolate signals
Calculate cross correlation
Find intervals, where cross correlation is closely below 1
true
intervals exist
false
Shift reference signal against test signal by positions in next interval
Calculate minimal mean square error in interval
Calculate absolute differences
within thereshold
true
within tolerances
false
false
true
Delete occurrence
Save occurrence
true
still peaks left
false
occurence exist
true
false
signal does contain property
signal does not contain property

Figure 7.2.: Work flow of property identification. The first three steps (interpolate signals, calculate cross correlation, find intervals) compose the operation of finding possible times at which a property might occur, in the remaining ones the results are checked and filtered.

By executing these steps one after another, the correct and quick identification of occurrences is guaranteed. We subdivide the identification process in these two main operations in order to avoid time intense comparisons of reference signal and test signal along of the whole length of the test signal.

Before we can start with the identification of possible occurrences of a property in time, we need to take a preparing step and interpolate the signal based on the sampled data. This preparation is necessary because of the method which will be used to compute the cross correlation in order to identify the occurrences. In Section 3.1.2 we already introduced the definition of cross correlation (3.2), let us repeat it here for better understanding:

$$R_{xy}(\tau) = \lim_{T_F \to \infty} \frac{1}{T_F} \underbrace{\int_{-T_F/2}^{T_F/2} x^*(t) \cdot y(t-\tau)\, dt}_{=I} \tag{7.1}$$

An established numerical approach to compute the cross correlation of sampled, discrete data would be to omit the integration and sum up the product of the shifted data points; $m \in \mathbb{N}$ takes now the role of τ.[1]

$$\hat{R}_{xy}(m) = \begin{cases} \sum\limits_{n=0}^{N-m-1} x_{n+m} y_n^* & m \geq 0 \\ \hat{R}_{yx}^*(-m) & m < 0 \end{cases} \tag{7.2}$$

MATLAB's `xcorr` function in the Statistical Toolbox, for example, uses this equation to calculate cross correlation. This method has two drawbacks:

- it relies on sampling with a constant and fixed sampled rate, and
- it does not take into account the signal function's characteristics, e.g. change rate, etc.

Expecting that the test signals are sampled, but the sampled data not necessarily equidistantly distributed in time, we have to use the original definition. For this we either need to resample both of the signals with the same sample rate, or we can analytically derive the cross correlation function from the functions of the signals. As we will discuss in Sec. 8.3, the results obtained from applying `xcorr` to resampled signal are unsatisfactory. For both possibilities we need to interpolate the signal values in order to obtain intermediate values or an interpolation function.

---

[1] In (7.2), y_n^* and $\hat{R}_{yx}^*$ are the conjugate complex of y_n and $\hat{R}_{yx}$.

7.2.1. Fitting and Interpolating Sampled Signals

There exist several methods for obtaining additional values for intermediate steps from a given discrete set of pairs of breakpoints and values. We can distinguish those into two groups:

Approximation: fitting one polynomial or other function to the given function values while minimizing the error between function values at given breakpoints.

Interpolation: determine the values between the elements of the set with piecewise functions defined on the intervals between the breakpoints. The functions' values at the breakpoints are equal to the corresponding pairs.

The first group includes methods such as polynomial fitting, where one polynomial approximates the given set of values. Here we cannot guarantee that every element from the given set is met, and the error between the element and the polynomial's values at the breakpoint has to be minimized. Another fitting method are Fourier approximations using sums of sine and cosine functions of certain harmonics.

The second group interpolates in a piecewise manner, i.e. for each interval between two successive elements a function according to certain boundary conditions is calculated. These boundary conditions depend on the chosen interpolation method and they can restrict, e.g., the gradient near original values. In Tab. 7.1 we list the characteristics of some fitting and interpolation methods. While the first two methods, polynomial and Fourier, are fitting methods, the others are piecewise interpolation methods.

In Fig. 7.3 we apply those methods to show how differently sampled data is approximated and interpolated, resp. The original function (reference in grey colour) is constantly 0 up to the time $t = 0$, then oscillates with a rising amplitude. This signal is sampled with a rate of 1s^{-1}, and diverse fitting and interpolation methods are applied to reconstruct the original signal function. High-order polynomials tend to exhibit oscillatory behaviour, known as Runge's phenomenon, we see this behaviour around -10. This also leads to the problem that not all nodes are met when the original signal is oscillating. In contrary to this, Fourier approximation meets all nodes, and shows decent results when the original function oscillates. Still, the resulting function oscillates for constant function values. This example with a switching function exhibits the disadvantages of fitting such functions.

Compared to this, piecewise methods perform better by following the data points quite closely. The simple piecewise linear interpolation connects successive data points by straight lines. Due to this, the interpolation result is not differentiable in the data points. Spline interpolation, in contrary, leads to differentiable results, as we see with Hermite and cubic splines. Hermite splines follow jumps in the original data set more closely then cubic splines, which tend to oscillate.

Method	Characteristics
Polynomial fitting	One polynomial approximates all given elements Shape depends on maximum order Oscillation possible (Runge's phenomenon)
Fourier approximation	Sum of weighted sine and cosine functions Suited for periodic functions
Linear interpolation	Simple piecewise method No side conditions at data points Not differentiable at data points
Spline interpolation	Piecewise cubic polynomials First and second derivative at nodes is constant Resulting graph is "smooth"
Hermite interpolation	Piecewise cubic Hermite polynomials First derivative is constant at nodes

Table 7.1.: Characteristics of a choice of fitting and interpolation methods.

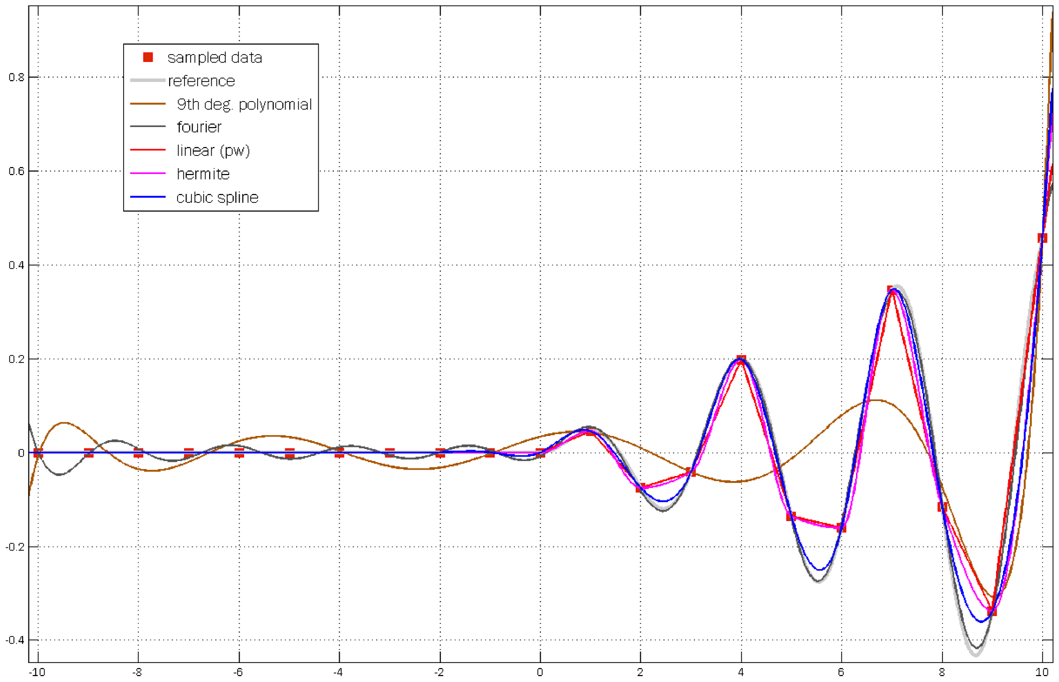

Figure 7.3.: Comparison of interpolation methods introduced in Table 7.1. The original signal is depicted in grey colour.

When we keep in mind that often in control system we have to cope with signal functions exhibiting jumps and switching, piecewise interpolation methods are the better choice when computing intermediate values. Additionally, sampling takes place at short, possibly varying intervals, and differentiability is not required. Assuming a high sampling rate, piecewise linear interpolation is a good compromise being a simple method, and reconstructing the original signal function.

7.2.2. Analytically Deriving Cross Correlation

Having interpolated our set of signal data, we can continue with the search for properties in it. From signal processing and statistics we know *cross correlation* which we already introduced in Sec. 3.1.2. Remember that cross correlation gives us a measure of how similar two sets or signals are.

We make use of the fact that our functions are piecewise polynomials of order n and can subdivide the integral I in Eq. (7.1) using intervals in such a way that the coefficients of the interpolation polynomials of x and y are constant on each interval. The subdivision depends on both sampling time sets T_x and T_y, and the shift τ. The sampling time sets can be obtained from the sampled data sets as defined in (5.9), $T_x = \{t \mid (t,x) \in \mathcal{C}_x\}$ and $T_y = \{t \mid (t,y) \in \mathcal{C}_y\}$, $\mathcal{C}_x$ and $\mathcal{C}_y$ being the data sets of signals x and y, resp.

To compute the aforementioned intervals we construct an increasing sequence $\mathcal{T}'(\tau)$. First, we define

$$T_y(\tau) = \{t + \tau \mid t \in T_y\} \tag{7.3}$$

Secondly we restrict the unified time set to those values covered by both sets

$$\begin{aligned} T(\tau) = \{&t \in T_x \cup T_y(\tau) \mid \\ &t \geq \max(\min(T_x), \min(T_y(\tau))) \wedge t \leq \min(\max(T_x), \max(T_y(\tau)))\} \end{aligned} \tag{7.4}$$

Finally we construct the increasing sequence with the elements of $T(\tau)$

$$\mathcal{T}(\tau) = (t'_0, \ldots, t'_m) \text{ with } t'_l \in T(\tau), l \in \{0, \ldots, m\} \tag{7.5}$$

The set $T(\tau)$ contains all sample times which lie in the common interval of both signals, i.e. we obtained the sample times of the intersection of the continuous time intervals of both signals.

Based on this sequence we can divide I into $m-1$ subintegrals

$$I = \sum_{l=0}^{m-1} I_l \tag{7.6}$$

Each of these I_l can be solved as follows (a more detailed derivation can be found in App. A).

$$I_l = \int_{t'_l}^{t'_{l+1}} \sum_{i,j=0}^{n} x_i\left(t\right) \cdot t^i \cdot y_j\left(t-\tau\right) \cdot \left(t-\tau\right)^j \; dt \tag{7.7}$$

We set $x_i\left(t\right) = x_{t'_l,i}$ for $t \in \left[t'_l, t'_{l+1}\right]$ and $y_i\left(t\right) = y_{t'_l,i}$ for $t \in \left[t'_l, t'_{l+1}\right]$:

$$I_l = \sum_{i,j=0}^{n} x_{t'_l,i} y_{t'_l-\tau,j} \int_{t'_l}^{t'_{l+1}} t^i \cdot \left(t-\tau\right)^j dt \tag{7.8}$$

$$I_l = \sum_{i,j=0}^{n} x_{t'_l,i} y_{t'_l-\tau,j} \sum_{k=0}^{j} \left(-1\right)^j \binom{j}{k} \int_{t'_l}^{t'_{l+1}} t^{i+k} \tau^{j-k} dt \tag{7.9}$$

Setting $\delta_l = t'_{l+1} - t'_l$ we obtain

$$I_l = \sum_{i,j=0}^{n} x_{t'_l,i} y_{t'_l-\tau,j} \sum_{k=0}^{j} \frac{\left(-1\right)^j}{i+k+1} \binom{j}{k} \delta_l^{i+k+1} \tau^{j-k} \tag{7.10}$$

The limit function in Eq. (7.1) is not applicable in our case and needs to be substituted, since our data has limited range in time. The term $\lim_{T_F \to \infty} \frac{1}{T_F}$ is used to scale the result of I, so we have to introduce a different scaling. There are different methods to scale the cross correlation function, one of these is computing the cross correlation of the shifted signal with itself for shift τ, i.e. $y\left(t-\tau\right)$ and $y\left(t\right)$ shifted by τ, and setting it to 1. This, in fact, is the autocorrelation of the shifted signal y, and we denote it by the following equation (cf. Eq. (7.1))

$$R^u_{y_\tau y}\left(\tau\right) = \int_{t'_0}^{t'_m} y^*\left(t-\tau\right) \cdot y\left(t-\tau\right) \; dt \tag{7.11}$$

The integration limits t'_0 and t'_m are the minimum and maximum sample times shifted by τ.

In the end we obtain piecewise polynomials of τ with the same order n as the original interpolation polynomial.

$$\begin{aligned} R_{xy}\left(\tau\right) &= \frac{1}{R^{u}_{y_\tau y}\left(\tau\right)} \sum_{l=0}^{m-1} I_l && (7.12)\\ &= \frac{1}{R^{u}_{y_\tau y}\left(\tau\right)} \sum_{l=0}^{m-1} \underbrace{\sum_{i,j=0}^{n} x_{t'_l,i} y_{t'_l-\tau,j} \sum_{k=0}^{j} \frac{(-1)^j}{i+k+1} \binom{j}{k} \delta_l^{i+k+1} \tau^{j-k}}_{\text{(cf. Eq. (7.10))}} && (7.13)\\ &= \frac{1}{R^{u}_{y_\tau y}\left(\tau\right)} \sum_{l=0}^{m-1} \sum_{s=0}^{n} \underbrace{\left(\sum_{r=s}^{n} (-1)^r \sum_{i=0}^{n} \frac{\delta_l^{i+r-s+1}}{i+r-s+1} x_{t'_l,i} y_{t'_l-\tau,r} \right)}_{a_{l,s}} \tau^s && (7.14) \end{aligned}$$

The idea of the step from Eq. (7.13) to Eq. (7.14) is to solve the Eq. (7.13) for τ. Observe that $R^{u}_{y_\tau y}$ can be computed with Eq. (7.14) by setting the denominator to 1. In general the values for δ_l differ in the terms of numerator and denominator. With the cross correlation function at hand we determine shifts of the reference signal against the test signal, at which the normalised cross correlation shows values close to 1. These are the points in time at which the property from which the reference signal has been created is expected to occur in the test signal.

The individual coefficients' values $a_{l,s}$ in Eq. (7.14) can be computed and used in an algorithm for cross correlation, `axcorr` [38, 40]. This newly implemented algorithm uses Eq. (7.14) to determine the value of cross correlation of a property reference signal and the measured test signal for given τ. In `axcorr` we make use of vectorised functions and are able to compute the values for $R_{xy}\left(\tau\right)$ for a whole interval of values of τ, e.g. $\{\tau = 0.1k \mid k \in \{-1000, -999, \cdots, 0, \cdots, 999, 1000\}\}$.

Starting with the sampled data sets of both signals x and y, e.g. test signal and a property's reference signal, and an array with the values of τ, we compute the piecewise interpolation of both signals. The individual values of the τ array determine the shifts for which cross correlation is computed. For performance reasons we precompute the sequences $\mathcal{T}\left(\tau\right)$ for each element of the τ array in the second step. We look up the interpolation polynomials' coefficients according to the sequence's elements and calculate the coefficients $a_{l,s}$. For linear interpolation, we obtain following coefficients:

$$\begin{aligned} a_{l,0} &= \sum_{r=0}^{1} (-1)^r \sum_{i=0}^{1} \frac{\delta_l^{i+r+1}}{i+r+1} x_{t'_l,i} y_{t'_l-\tau,r} && (7.15)\\ &= \delta_l x_{t'_l,0} y_{t'_l-\tau,0} + \frac{\delta_l^2}{2} x_{t'_l,1} y_{t'_l-\tau,0} - \frac{\delta_l^2}{2} x_{t'_l,0} y_{t'_l-\tau,1} - \frac{\delta_l^3}{3} x_{t'_l,1} y_{t'_l-\tau,1} && (7.16)\\ a_{l,1} &= -\frac{\delta_l^2}{2} x_{t'_l,0} y_{t'_l-\tau,1} - \frac{\delta_l^3}{3} x_{t'_l,1} y_{t'_l-\tau,1} && (7.17) \end{aligned}$$

By summing up the cross correlation's polynomial we obtain the value of unscaled $R_{xy}(\tau)$. The scaling factors for every shift τ are also determined with the same algorithm.

The `axcorr` algorithm is implemented as a MATLAB function and makes use of the predefined interpolation function `interp1`. Additionally to the steps mentioned above we have to recompute the interpolation coefficients, since MATLAB's interpolation function generates coefficients relative to the last sampling point.

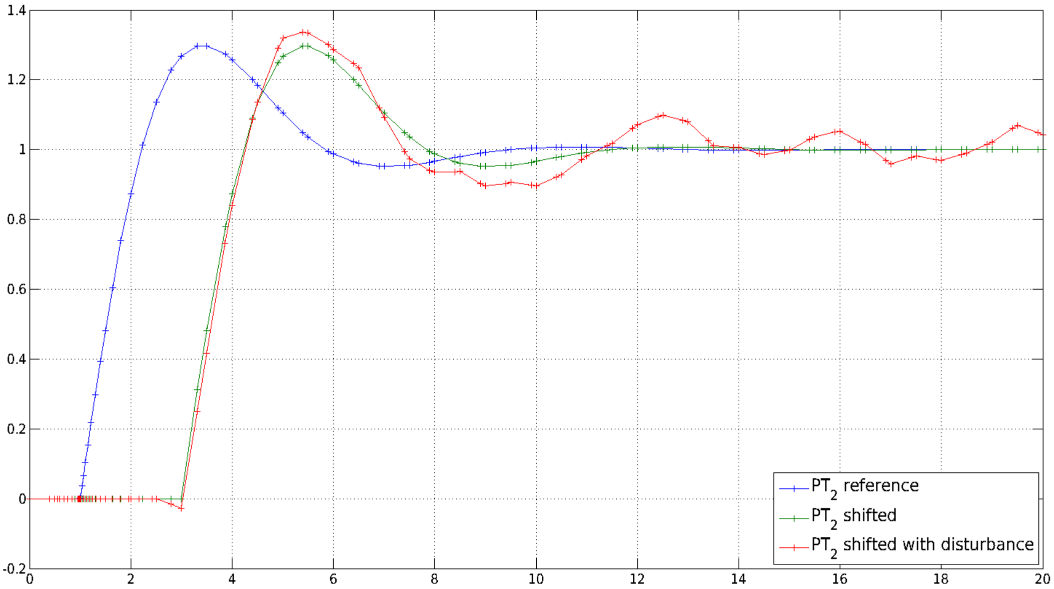

Figure 7.4.: Shifted PT$_2$ behaviour serving as example for early feasibility check.

The feasibility and advantages of `axcorr` will be discussed with help of the following example. In control engineering, the step response is used to identify the behaviour of a system. For this the input variable is set instantly from one value to another, e.g.

$$u(t) = \begin{cases} 0 & t < t_{\text{step}} \\ 1 & t \geq t_{\text{step}} \end{cases}$$

One possible response is called PT$_2$ and is governed by the following differential equation

$$K\,u(t) = T^2\ddot{y}(t) + 2dT\dot{y}(t) + y(t)$$

where $K > 0$ is the gain, $T > 0$ the time constant and $d > 0$ the damping constant. The blue curve in Fig. 7.4 is a plot of a step response at $t_{\text{step}} = 1$ showing PT$_2$

behaviour. In comparison to this, the green curve is shifted against the blue one by 2, while the red one is artificially disturbed by adding noise. Observe that all step responses are sampled at different points in time, which makes the analysis of the sampled signals more difficult.

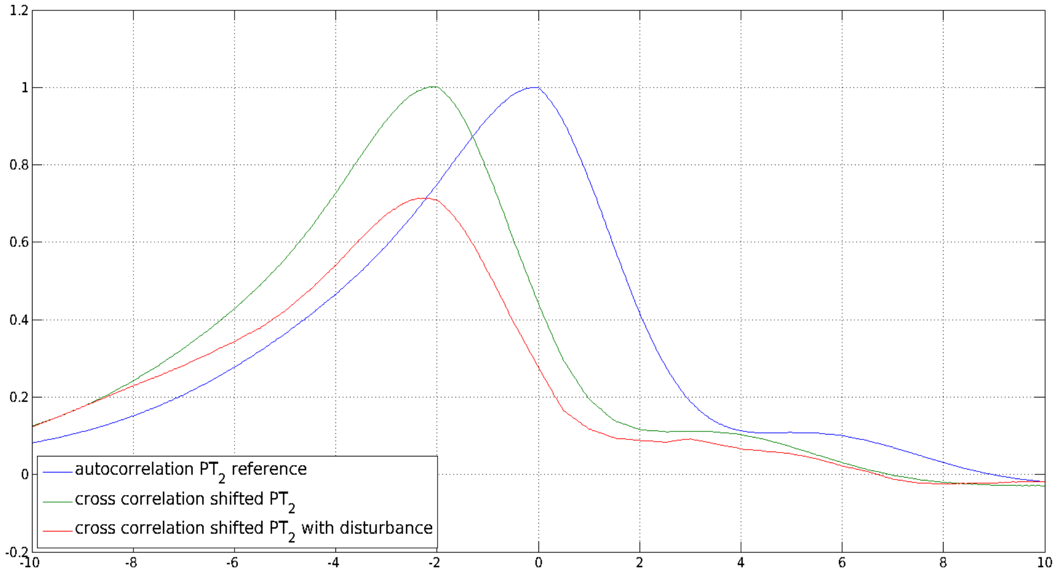

Figure 7.5.: Computing the analytically derived cross correlation numerically with `axcorr`. The autocorrelation exhibits a maximum of 1 at $\tau = 0$, while the cross correlation of the original and the shifted signal exhibits the maximum at $\tau = -2$. The cross correlation of the original and the shifted disturbed signal has its maximum at $\tau = -2$, but the maximum is now approximatly 0.7.

The results of computing the cross correlation of all three signals with the blue one are plotted in Fig. 7.5. The curves represent the values computed for an interval of values of τ, their colours correspond to those in Fig. 7.4. Computing the normalized autocorrelation, we expect a maximum of 1 at $\tau = 0$. In fact the blue one exhibits such a maximum. For the shifted green signal, a similar curve should be the result, but shifted by 2, while for the shifted and disturbed one we expect a lower maximum around 2.

The peak of the disturbed signal's cross correlation is lower than one would assume (below 0.8). Although the rising edge in the original signal can easily be found in the undisturbed ones, but from $t = 10$ on, the signals show different behaviour: while the undisturbed signals remain constant, the disturbed one oscillates. This shows the need for further validation of found occurrences, which will discussed in the following section.

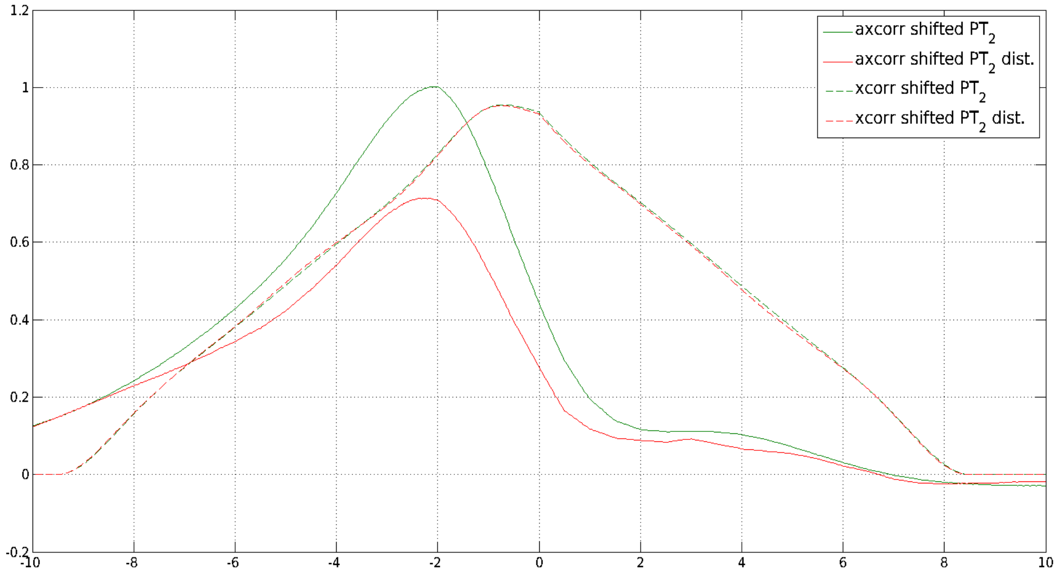

Figure 7.6.: Comparison between `xcorr`'s and `axcorr`'s results.

Let us first compare the results of `axcorr` with the ones obtained with `xcorr` applied to a linearly interpolated and resampled signal. The latter ones do not exhibit maxima at the expected shifts, and additionally, the resulting cross correlation values barely differ. In Fig. 7.6, the values for both shifted signals, undisturbed and disturbed, are plotted as dashed curves. When compared to those with the already known ones obtained by `axcorr`, we can observe that the computations made with `axcorr` represent the actual relations better. Reasons for this may be:

- In `axcorr` we work with the original sampling of the data and do not have to resample with a fixed rate.
- In `axcorr` we compute the cross correlation using the interpolated functions, thus taking into account the evolution of the signal in between. In contrast to this, (7.2) computes the cross correlation of only of the (re-)sampled data.

Additionally, in algorithms based on (7.2), the sampling rate controls the values for the shift τ, while in `axcorr` we can choose the τ's arbitrarily.

7.2.3. Checking the Occurrences

Since the values of the cross correlation function indicate the time of a property's occurrences, we have to validate the results of the step mentioned above. Additionally, we cannot conclude from the value how big the absolute differences from the reference are, and therefore, whether the test signal at that time conforms to the property. Hence, we determine certain intervals of potential occurrences of a property and use error metrics to compute the exact occurrences.

We choose to evaluate the shifts with help of mean square error, as well as absolute maximum positive and negative errors. Mean square errors are used to decide at what point of time a property really occurs, and since cross correlation gives a good hint we just have to calculate on certain intervals. For each interval of τ where $0.8 \leq R_{xy}(\tau) \leq 1.1$ holds we determine the minimal mean square error. This threshold is justified by the co-domain of the correlation, $[-1, 1] \subseteq \mathbb{R}$, where a perfect match would result in $R_{xy} = 1$. Values of $-0.7 < R_{xy} < 0.7$ indicate weak or no correlation at all.

Since we have to take disturbances or slightly different signals into account, we expect the value of R_{xy} to be less than 1. For every value of τ in one interval, we shift the property's reference signal to that corresponding position in the measured signal. Here, we compute the mean square error between the reference and a interval of the measured signal of the same length starting at that position. The shifts that exhibit minimal values for the mean square error are prime candidate values of τ for a property's occurrence.

Having determined possible occurrences of a property $p \in P$ in the signal, we have to filter those which lie in $\mathfrak{b}(p)$. This can be achieved by discarding the occurrences at which the absolute errors of sampled signal and reference signal lie outside of the properties semantics. Therefore we have to determine the tolerances of a reference signal when generating it from the property. These tolerances have to be included in the closeness criterion Φ (cf. Def. 4.5, and Lemma 5.8).

If no occurrence of a required property remains after the filtering process, the test signal does not contain all properties and therefore violates the abstract signal. If a faulty signal is recognised, the further progress varies and depends whether this signal is an input or an output. A faulty input signal indicates that accurate stimulus does not conform to the abstract model's version, thus the test case executed is not the valid one and the output signal does not have to be checked further. If the input signals do conform but one of the output signals is faulty, the reason for this lies in the SuT, since conforming stimuli are processed in a way that yields not conforming outputs.

7.3. Validation of Order and Timing

By now we have taken care of the second and third constraint in Lem. 5.5. However, we still did not check whether the first one holds. This constraint has two consequences:

1. The order of the properties' occurrences has to be compared with the one in the PFL signal.
2. The times of these occurrences have to meet the timing constraints derived from the abstract model.

In order to validate the order of the properties in the test signal, we sort the identified occurrences according to the corresponding times. By this we obtain a sequence of properties similar to the one defined in the abstract model of the signal. Comparing the order of both sequences shows, whether a test signal conforms to the abstract model. If a deviation between abstract model and the test signal's sequence occurs, the signal can not conform to the abstract model. As mentioned before, we would then have to distinguish between input and output signals.

A correct order of the property sequence is not sufficient for conformance of signals with respect to an abstract model when the correct timing is crucial. Hence, we have to check, whether the properties occur at the valid points of time. The timing information of the test signal's sequence is easy to obtain, but the information of the abstract model has to be processed to be usable in this step. As mentioned in Sec. 4, there are three kinds of timing information in the abstract model on the PFL: intervals in which a property has to occur, duration of phases (cf. Def. 4.3), and interdependencies between signals (cf. Def. 4.4). We have to take into account all of these information for computing acceptable times for a property's occurrence in a predefined order.

The information easiest to obtain are the intervals in time in which a property has to occur. These can be read directly from the model and are the base for our further calculations. If for a property no interval is given in the abstract model, all positive reals are possible times for this property at the beginning. In the following steps we restrict these intervals according to the durations and interdependencies. The first restriction comes in with the possibly defined durations of phases, the durations of preceding phases are summed up and restrict the interval of occurrence of the succeeding properties. The second restriction are the interdependencies, which also can be labelled with intervals, too. These intervals define the time span between the start of the originating property and start of the target property. After these restricting steps we can compare the times of the properties' occurrences with the computed intervals in the PFL signal, and thus check whether these times meet the first restriction in Lem. 5.5.

After all time information has been taken into account, we can compare the required times calculated in this step with the occurrences we got from property identification. If the times of all occurrences lie within the according intervals, then the signal must lie in the set of allowed signals, otherwise the timing should be revised. Iterating this process for all signals of all behaviours of a system decides whether it conforms to its abstract model.

Again, we have to stress that the process we introduced here is a semi-decision procedure. Due to the arbitrary choice of one or more reference signals we cannot exclude false negatives, i.e. in case we did not find a property it might still occur in the test signal. If different reference signals have been chosen, we would have been able to find it. Still, we can exclude false positives, i.e. a identified property's occurrence is a “real” one.

Part IV.

Application

8. Back-to-Back Testing of Models and Autogenerated Code

This section demonstrates the general idea of the proposed approach for behaviour-based testing of embedded systems. The method is based on a conformance relation between two systems A and B described by their discrete and continuous behaviour. Let us assume that for the discrete part a suitable conformance relation is given, and only focus on the continuous behaviour of a system. The continuous behaviour of a system is given by a set of signals, in pairs of input/output signals.

8.1. Approach to Validation of Two Development Artefacts

We use the similarity relation with respect to the abstract model introduced in Chapter 5 to compare two development artefacts behaviour by behaviour. One scenario, in which the conformance relation is put to good use, is the one which originally motivated our work (cf. Sec. 2.1). In the model-based development of embedded software we usually have artifacts of different nature, maturity, etc., e.g. models and the code which was derived from the models manually or automatically. The question arises whether both of them behave in a way that is similar enough to be regarded as correct behaviour. Since we are dealing with embedded systems, continuous variables are a crucial issue and might show different behaviours in both kind of artifacts as was stated by Pehinschi, et al. in [41]. Still, both behaviours, though different, can be acceptable with regard to a specification; this situation is depicted in Fig. 8.1. Taking advantage of abstract behavioural models and the defined conformance relation, we can decide whether the behaviours of two systems, e.g. model and code, conform the abstract model and therefore, the systems themselves are similar with respect to these models.

8.2. Generalized Approach to Conformance Validation

The generalisation of the above described scenario results in a method to validate whether a system showing continuous behaviour conforms to its specification (Fig. 8.2). From this specification accurate continuous reference signals are created, to which the behaviour of the developed system should conform. Again, we can

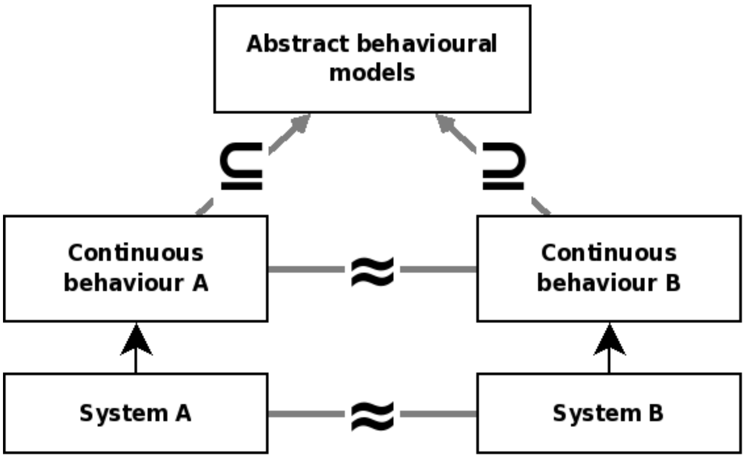
Abstract behavioural models
⊆
⊇
Continuous behaviour A
≈
Continuous behaviour B
System A
≈
System B

Figure 8.1.: Similarity with respect to abstract model; if the behaviours of both systems conform to the abstract behavioural model, their behaviours are similar with regard to this model.

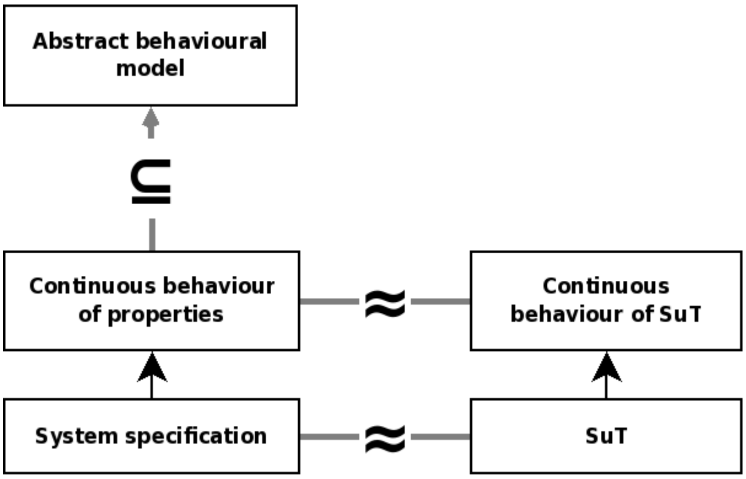
Abstract behavioural model
⊆
Continuous behaviour of properties
≈
Continuous behaviour of SuT
System specification
≈
SuT

Figure 8.2.: Testing using the conformance relation.

take advantage of abstract models and the conformance relation. The continuous behaviour from the specification can be obtained in different ways, e.g. by refining abstract behavioural models as mentioned in [37]. Alternatively, the simulation of a model yields continuous behaviour, too, and as mentioned in Sec. 3.2.2, the model in MBD represents requirements in executable form.

Showing that the continuous behaviour of a SuT lies in the set of allowed behaviours of the abstract models, implies that the SuT is similar to its specification with regard to the abstract model. The test suite with the test cases can be derived from the specification, and the test stimuli are sets of continuous signals. For test execution, the input signal is fed to the SuT, and the output of the signal is measured. Then, this measured output signal from the SuT is checked for the conformance with the output signal of the test case specification. The test case is passed when the output signals conform to each other with respect to the abstract model, the test suite is passed when all test cases are passed.

8.3. Results

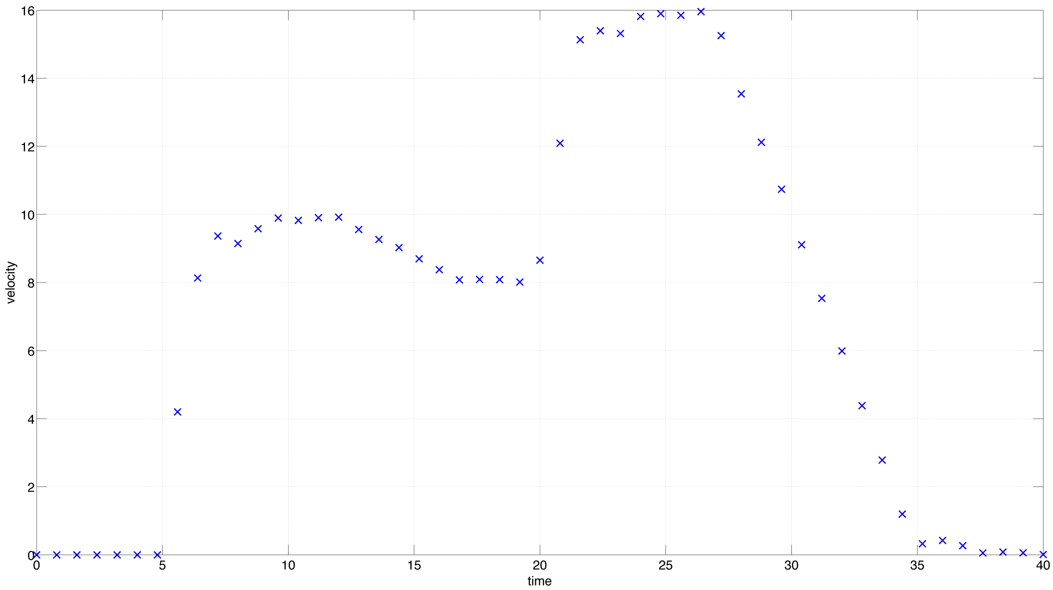

Figure 8.3.: Measured disturbed signal of a vehicle’s velocity.

To determine whether two systems behave in a similar way with respect to an abstract model, we have to check the conformance of the measured signal data with the signal definition in the abstract model. In this section we will be working with an example taken from the automotive domain, similar to the one found in [23], and will show central aspects of the property identification process with one signal. We start with measurements of a vehicle's velocity (Fig. 8.3) and an abstract signal definition on PFL, which belongs to one of the abstract system model's behaviours (Fig. 8.4). In Fig. 8.4 the PFL model of the output signal is depicted, events are depicted as ovals or circles, phases as rectangles. Here, in the vertices' labels the properties are defined by the following four variables:

- t represents timing information, i.e. in what interval in time a property should start,
- d denotes the interval of a phase's duration,
- v is the signal variable, i.e. the vehicle's velocity, and
- a stands for the acceleration $\dot{v}$, the rate of change of v.

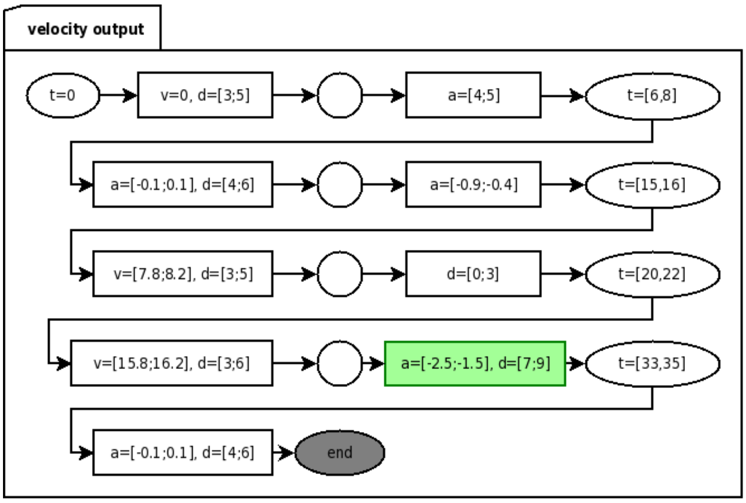

Figure 8.4.: Graphical representation of an example signal definition on PFL. Rectangles represent phases, ellipses and circles represent events; variables t and d capture timing and duration information, while v is the signal's value and a the rate of its change. In the property identification step we will search for the phase marked with green colour.

The graphical representation of conforming signals on the more accurate layers are depicted in Fig. 8.5. Here, we see both a TML and a SML signal model using the same time scale. The TML signal uses the same labels as defined in

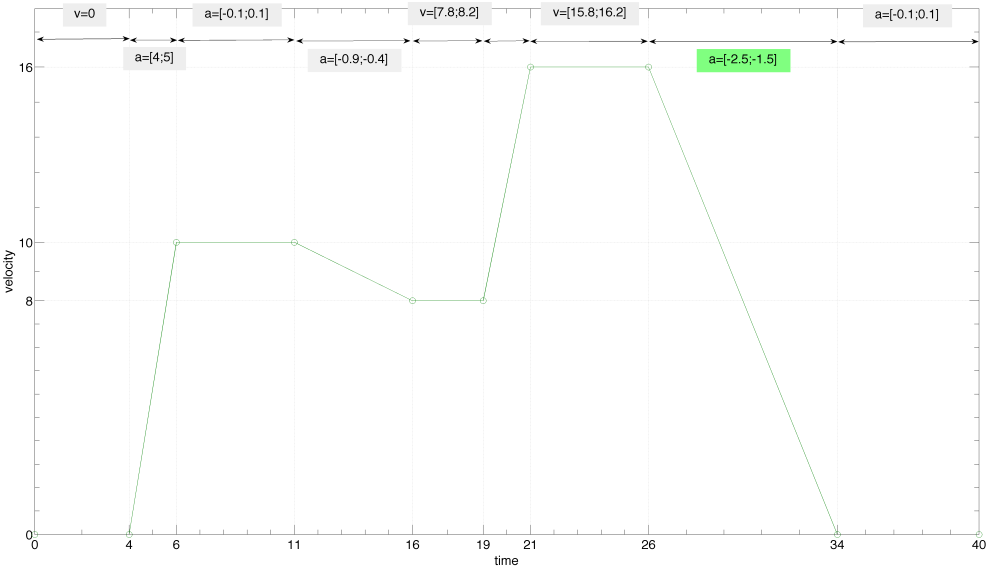

Figure 8.5.: Graphical example signal definition on TML and on SFL. The TML representation on top uses the same time axis as the SFL representation below. The SFL signal is represented by its semantics.

the aforementioned PFL model. Hence, one can validate that the points of time correspond to the durations and the timing intervals in the PFL model. The SFL model is represented by its sole element in the semantics, namely a piecewise defined function using affine functions on the intervals. When we graphically compare the SFL signal with the measured data in Fig. 8.6, we see that there are obvious similarities. Still neither the measured signal's timing is the same as the one of the SFL signal, nor are the values.

In order to check whether the signal conforms to the abstract model, first of all we have to find the properties specified in the abstract model. We pick the phase marked with green colour in Fig. 8.4 as an example. This is a "braking" phase where the signal v has a gradient (vehicle's acceleration a) between -2.5 and -1.5, and this phase takes between 7 and 9 seconds. In order to generate a fitting reference signal we also have to take a look at the predecessor or the successor of the corresponding phase.[1] Those provides us with the information that the starting value of our ramp lies around 16 (vehicle's velocity). This is why we choose the following times and

---

[1] As mentioned in the context of Lemma 5.8, it may be necessary to use sequences of labels to create appropriate reference signals.

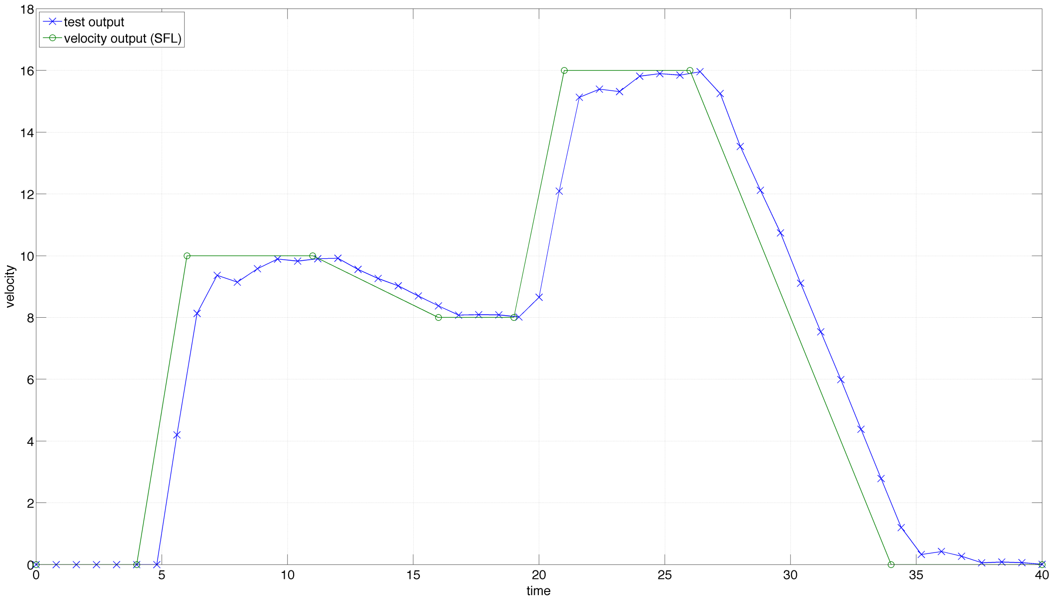

Figure 8.6.: Graphical comparison of the interpolated measured signal and SFL model from Fig. 8.5

functions for our reference signal (cf. Fig. 8.7):

- $T_x = \{0, 1, 9, 10\}$,
- $v = \begin{cases} 16 & \text{for } t \in [0,1] \\ -2t + 16 & \text{for } t \in [1,9] \\ 0 & \text{for } t \in [9,10] \end{cases}$

By adding short signals originating in the neighbouring properties at the beginning and the end of the reference itself, we take the second restriction in Lem. 5.5 into account. By this, we are able to check the connection of the corresponding phase to its temporal neighbours. The occurrences of that phase in a different context within the measured signal can be omitted more easily.

With this reference signal at hand, we can search for an occurrence of this property in the measured signal. In general measurements yield discrete sampled data, thus we have to interpolate it first. As discussed in Section 7.2.1, with piecewise linear interpolation we obtain good results (cf. Fig. 8.6). This interpolation technique also supplies the functions we need for computing the cross correlation of reference signal with the test signal. The result of this is shown in Fig. 8.8, where also the interval where $0.8 \leq R_{xy}(\tau) \leq 1.1$ is marked. As one can see, a peak lies in this

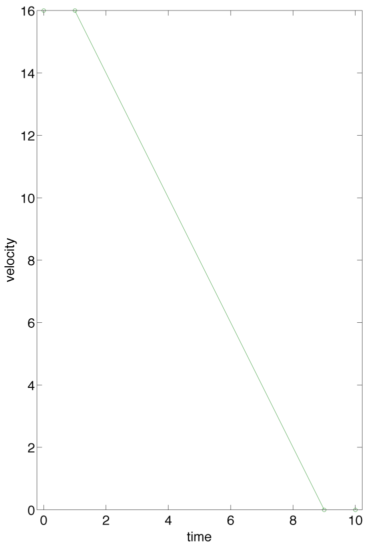

Figure 8.7.: Reference signal for the phase marked with green colour in Fig. 8.4.

interval at a lag of $\tau = 25.4$. When we shift the reference signal by this lag against the measured signal, we see that the property is found with an acceptable quality (Fig. 8.9, left).

The next step is to validate the occurrence by computing the mean square error for the marked interval. When we determine the minimal mean square error on the surrounding interval, we obtain a lag of $\tau = 25.95$. In fact, when we compute the absolute errors Δv at both lags, we get

$$\Delta v \in \begin{cases} [-1.20, 0.15] & \text{for } \tau = 25.4 \\ [-0.60, 0.53] & \text{for } \tau = 25.95 \end{cases} \tag{8.1}$$

Fig. 8.9 shows the reference signal shifted by the computed values; the computation of cross correlation already gives a very good hint (left), while additionally taking into account the minimal mean square error gives an almost perfect match (right). This is one example indicating that computing cross correlation analytically with `axcorr` is a valid method for finding properties in sampled signals obtained through measurements; we will discuss another one in the following chapter. Furthermore, additionally computing the minimal mean square error on the intervals where the cross correlation lies within the defined boundaries can be used to validate and improve the result.

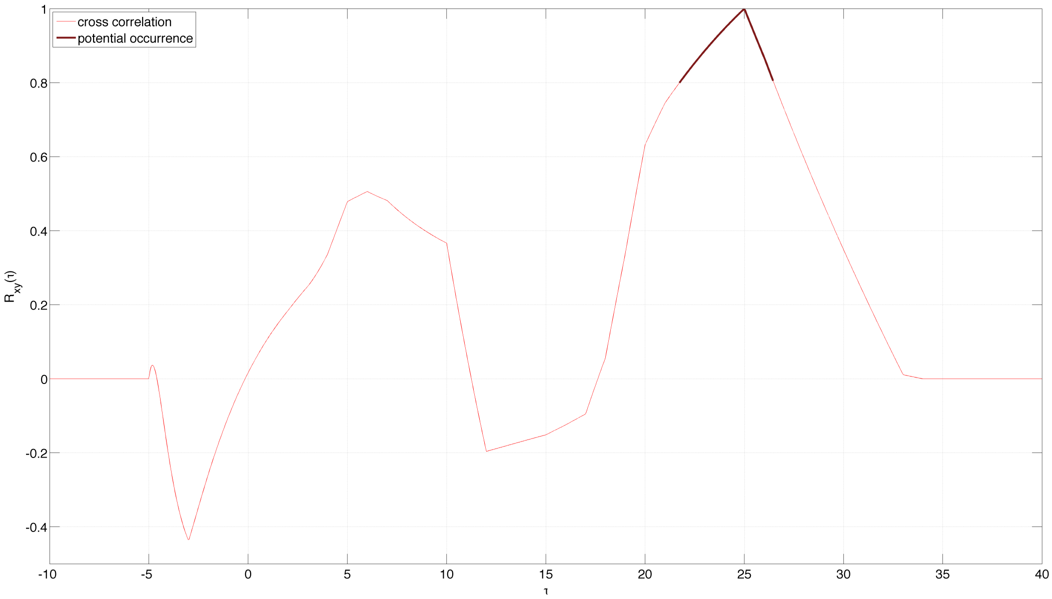

Figure 8.8.: Analytically computed cross correlation of our example test signal and reference signal, mean square error is computed in marked interval.

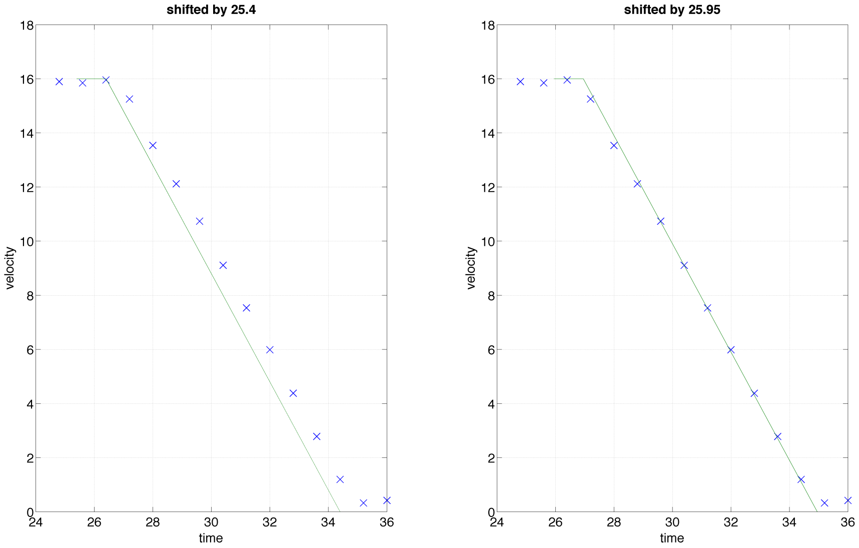

Figure 8.9.: Reference signal shifted against test output (left: after computing cross correlation; right: after additionally computing minimal mean square error).

After iterating this process for all properties, we can assure that the order and timing conforms to the abstract model. Similarly to above example, the lags of the other properties except the first phase lie approximately one second after their occurrences in the SFL model in Fig. 8.5. Having computed the shifts, the signals are compared with the timing and duration information obtained from the PFL model. Since these steps become clear from their description in Sec. 7.3, we omit a detailed elaboration on validating. To sum up, we were able to validate that the measured signal conforms to the PFL model. Hence, if further measurements are available, we could check the similarity of those measurements with respect to this model.

9. Clock Drift Estimation

Hardware measurements systems can exhibit different timing behaviour due to drifts in intern clocks. This may lead to wrong sampling of continuous signals, and in turn to wrong test results. This chapter describes how the proposed conformance check can be applied to solve such a problem which originates from a real testing problem in industry. The knowledge on the drift of a clock helps to correct such measurement faults, and therefore is crucial for testing real time behaviour.

9.1. Approach

We will use the property-based conformance checking to estimate the drift of an unknown measurement system. To do so, another measurement system with known drift is required to derive a test oracle. The unknown systems serves as SuT. Thus both system measure the same continuous signals to yield sampled data. Each of these sampled data should exhibit similar representations of the same properties in the same order. In contrast to the order, the measured values over time will differ if the resulting signals values are naively mapped on the same time scale. This leads to a growing discrepancy over time, as depicted in Fig. 9.1

An expert tester would easily realise that the signals are similar. He recognises that similar signal properties follow each other in the same order. By identifying these properties and calculating the shift of their occurrences, the drift per sample can be estimated. In fact, when comparing the graphs of two functions, humans try to recognise similar characteristics and patterns. This is equivalent to the process of modelling signals with help of properties, i.e. this is equivalent to PFL. We use this fact to utilise the conformance check with `axcorr` to solve this problem.

Figure 9.2 represents the workflow of how to apply the conformance checking of sampled continuous data with respect to an abstract model to estimate clock drift. From the oracle's measurement the user cuts out intervals in time at which a property occurs, we call this *interval reference snippet.* By recognising a property, the user implicitly uses an abstract representation of it. After having found an appropriate snippet he refines this abstract representation from PFL to SFL. The next step is to find a part of the SuT's measurement that is similar to the reference snippet.

1. First, we calculate the cross-correlation of snippet and SuT's measurement

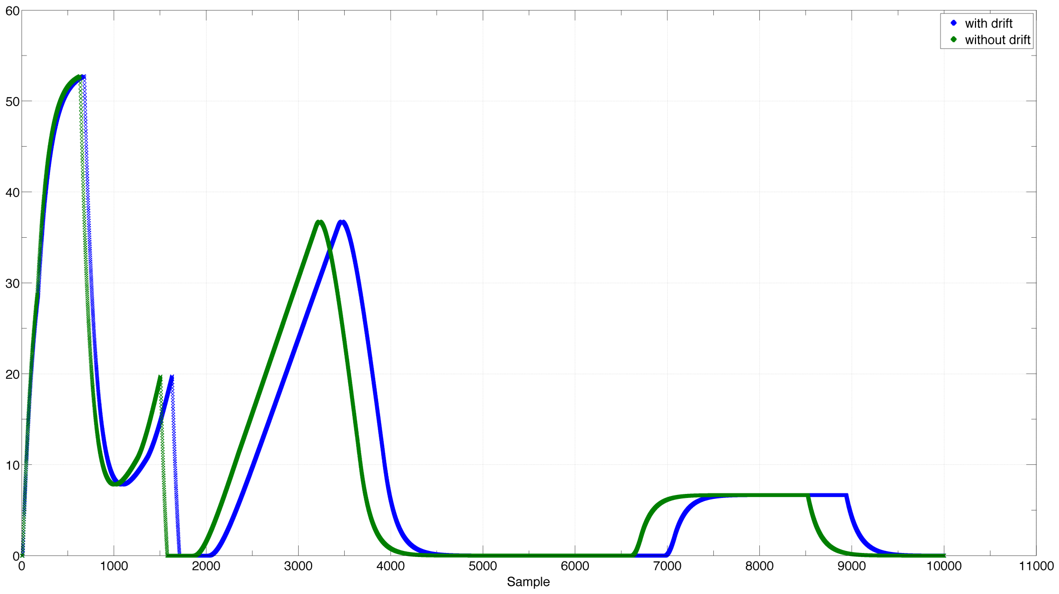

Figure 9.1.: The device with drift samples the signal more often, thus more values are obtained. When mapped on the same time scale along with the correctly working device, the signal seems to be stretched and longer. This figure is the same as Fig. 2.2, p. 15 and repeated here for better readability.

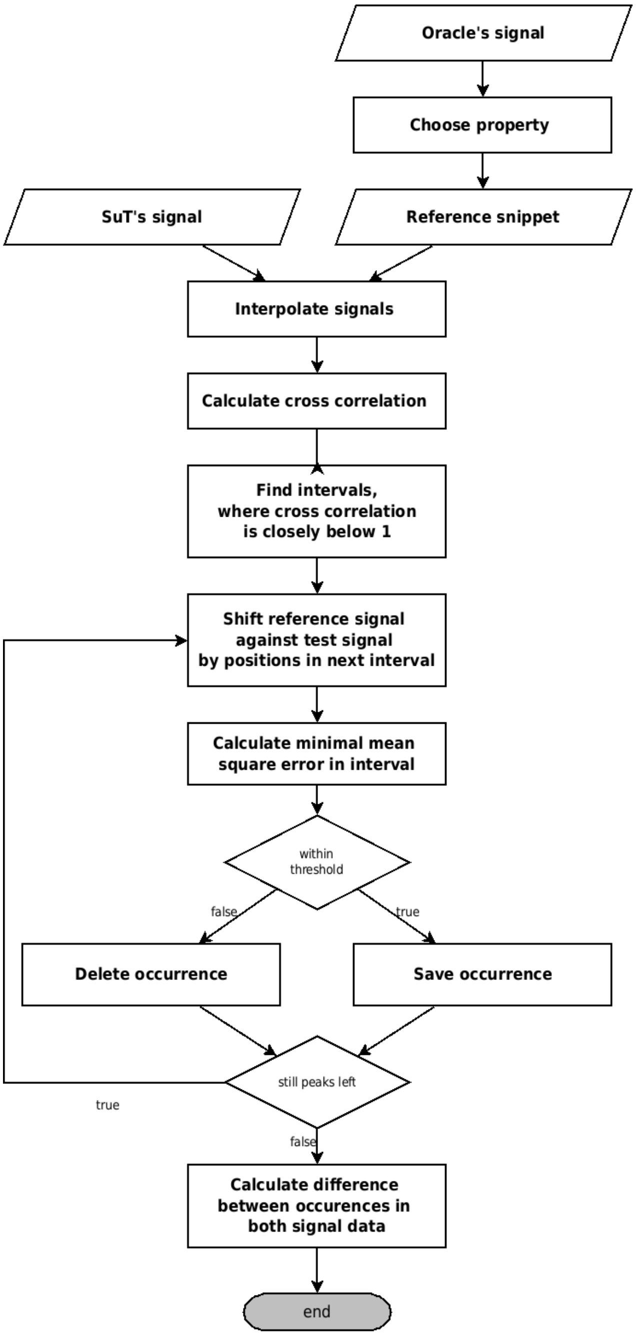
Oracle's signal
Choose property
SuT's signal
Reference snippet
Interpolate signals
Calculate cross correlation
Find intervals, where cross correlation is closely below 1
Shift reference signal against test signal by positions in next interval
Calculate minimal mean square error in interval
within threshold
false
true
Delete occurrence
Save occurrence
still peaks left
true
false
Calculate difference between occurences in both signal data
end

Figure 9.2.: Using conformance checking for drift estimation.

with a coarse time scale. This step yields intervals on which the property is likely to occur.

2. Second, we refine the result by finding the minimal mean square error on the identified property occurrence intervals.

Subsequently, we obtain points of time, at which the property occurs in the SuT's measurement and can therefore determine the difference in time between the occurrences in both measurements. This directly gives us the drift per sample.

9.2. Results

The data we are working with origins from a simulation of a Simulink model with two measurement devices implemented in Stateflow (cf. Fig.9.3). Both devices are hybrid automata with one state. The measurement takes place, when the transition is triggered each $\delta t = 10$ ms. While in one device $\dot{t} = 1$, in the other $\dot{t}$ is set to 1.1. Thus the second clock runs faster and the signal is sampled more often. The simulation itself is executed at a sample rate of 0.01. Due to numerical issues we expect that the mean drift per sample will be less then 1.1. This leads to a longer, stretched signal, then it is in reality (cf. Fig. 9.1).

Starting with two sampled data sets for one signal, the user identifies characteristic properties in the set obtained by the correct working measurement system. Cutting the appropriate data from the data set creates a signal snippet that serves as property's reference on SFL. With this at hand we can apply `axcorr`. In this example we choose the property in the correctly measured signal in Fig. 9.1 occurring at $t = 700$ s. We create a snippet of this property by extracting the values at simulation samples 69000 and 75000 (Fig. 9.4).

Calculating the cross correlation with `axcorr` yields several intervals where the function value is near 1 (cf. Fig. 9.5). We validate these results by calculating the minimal mean square error on these intervals with a finer subdivision of τ. As depicted in Fig. 9.6, there is one minimal value for mean square error at a shift of 726.8 seconds from the signal's start. In the correctly sampled data, this property occurs at sample 69000, i.e. at $t = 689.24$ s. Thus, the property appears to be shifted by 37.56 s which implies a drift rate of 0.0545. This means that during each measurement sample of the correctly working device 1.0545 samples of the drifting one are executed.

In Fig. 9.5 we observe, that `axcorr` calculates values higher than 1 and lower than -1, which should not occur. The origin of this seldom observed effect is not known yet. When it was observed, we were able to validate that no potential occurrences were missed. Getting rid of these miscalculations will be one of the tasks still to solve.

In this example we did not take into account jitter on purpose, in order be able to get first validation results. A more elaborate future validation should include such additional disturbances.

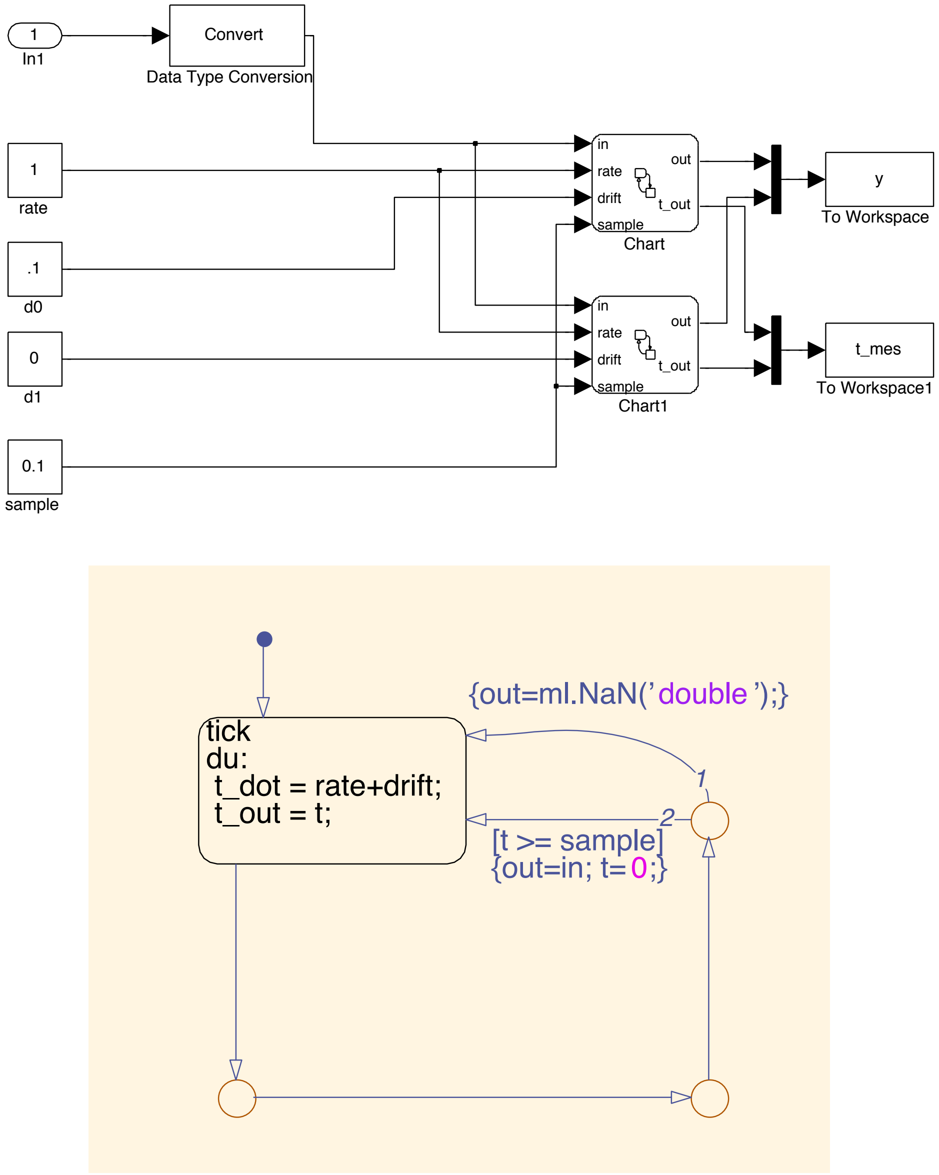
1
In1
Convert
Data Type Conversion
1
rate
.1
d0
0
d1
0.1
sample
in
rate
drift
sample
out
t_out
Chart
in
rate
drift
sample
out
t_out
Chart1
y
To Workspace
t_mes
To Workspace1
tick
du:
t_dot = rate+drift;
t_out = t;
{out=ml.NaN('double');}
1
2
[t >= sample]
{out=in; t=0;}

Figure 9.3.: Simulink model implementing a correctly working and a drifting clock. The Stateflow charts are build in the same way and differ only in the input `drift`.

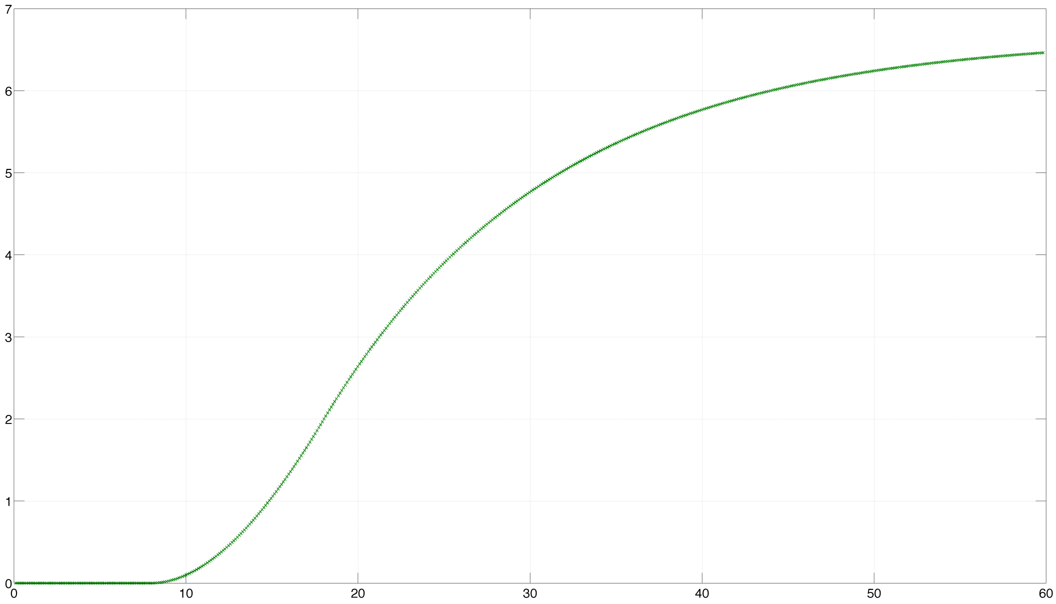

Figure 9.4.: Reference snippet of rising signal occurring at samples 69000 to 75000.

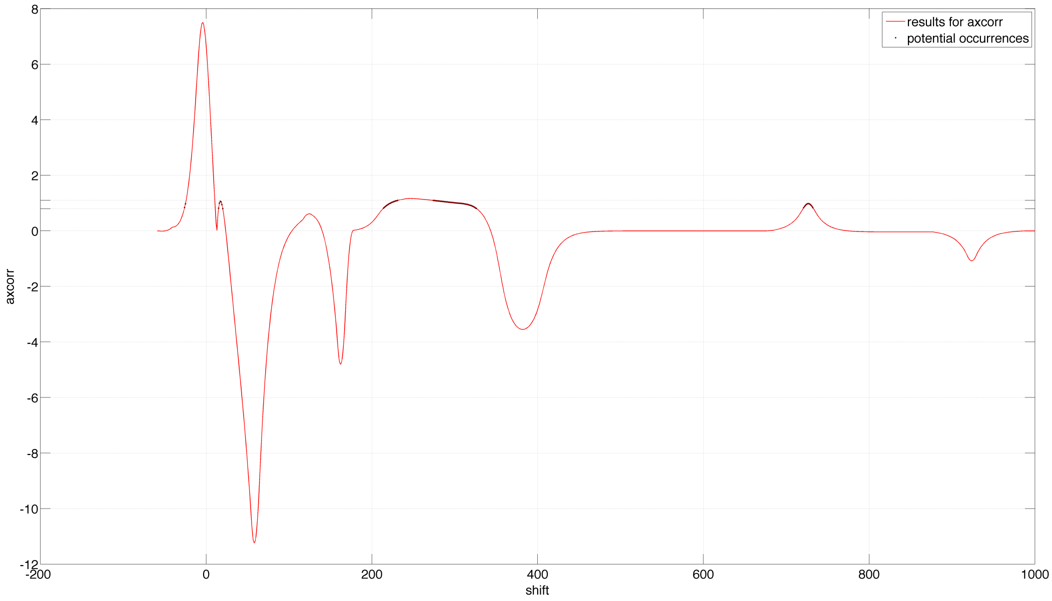

Figure 9.5.: Potential occurrences found with `axcorr`.

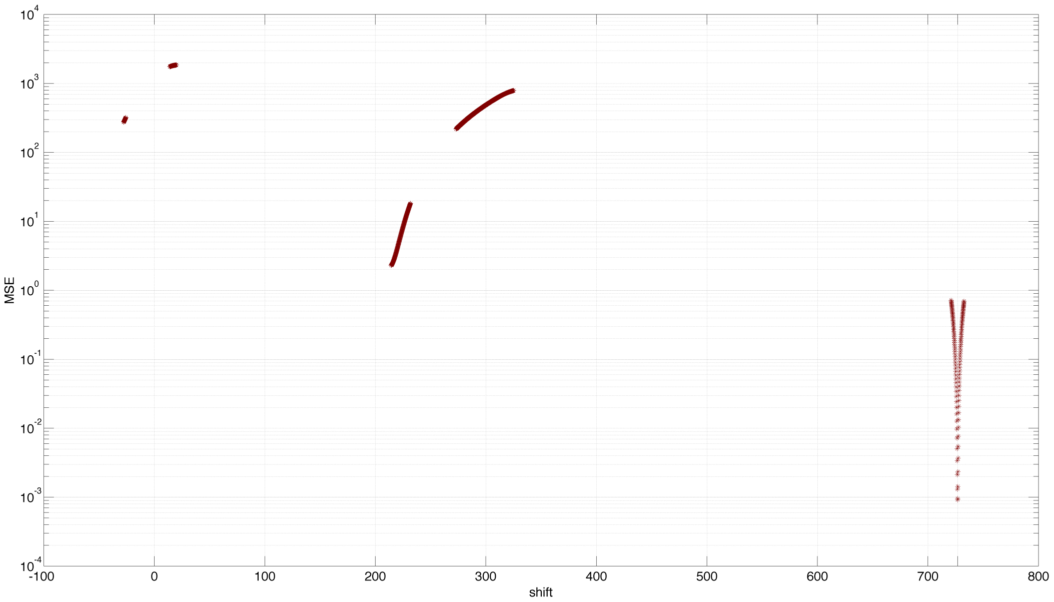

Figure 9.6.: Values of mean square error calculated at the intervals shown in Fig. 9.5. Notice the logarithmic scale.

Part V.

Discussion & Outlook

10. Related Work

Conformance checking is an important topic in software system testing. Driven by a practical point of view, different authorities defined several standards approaching conformance testing (cf. [19, 29, 55]). Firm theoretical basics to conformance checking was developed by several authors, e.g. Tretmans, Cavalli and others (cf. e.g. [11, 53, 54]). This is not only true for pure software systems, but also for real-time systems where timed automata (see [6]) and relations like timed ioco (see [45]) play an important role.

As mentioned in the literature on embedded systems, a growing number of those systems are developed in a model-based approach, i.e. the controller software and the corresponding controlled system are modelled together in which the code is generated manually or automatically afterwards [13, 26]. Especially, the validation of automatically generated code is an issue. According to Stürmer, Conrad, et al., although there are sophisticated methods to prove the correctness of compilers and code generators, testing is seen as the most feasible one [50, 51]. Here, the proposed approach can be used to compare the behaviour of model and generated code in back-to-back test.

In recent work several novel approaches are demonstrated that tackle the question of how to cope with the inaccuracies and other problems, when it comes to continuous data. Surveys as the one conducted by Pehinschi [41] show that this is an issue of practical relevance. We discuss briefly two of those, namely TTCN-3 extensions and online test evaluation in Simulink. The latter is presented in detail by Zander-Nowicka, et al. in [56–58], where a multi-level testing methodology for automotive software is suggested. This also incorporates the modeling of test signal with help of features, which corresponds to the properties used in the approach presented in this work and are inspired by Gips and Wiesbrock [20].

Also using the ideas of properties of continuous signals is the work by Schieferdescker, et al. [23, 43, 44] introducing cTTCN-3, an enhanced version of TTCN-3. Originally, TTCN-3 [18] is a programming language for modelling test cases for testing communication protocols. While the first two publications introduce the concept of continuous signals to TTCN-3, the latter adds the notion of properties. Another approach to continuous signals in TTCN-3 is μTTCN by Bräuer, et al. [9], but there seems to be no further publication on this specific topic. All the work mentioned above approaches the problem of modeling continuous signals, but does not elaborate on checking of test results' conformance to those models. Such a conformance checking is one of the main contributions in the work presented here.

Piketech's TPT[10, 32], originally developed by Bringmann (nee Lehmann), et al. in cooperation with Daimler AG, is a tool for modelling and executing test cases in embedded software development. It offers a modelling method similar to the time region automata introduced by Alur and Dill in [6]. For communication with Simulink models, TPT provides interfaces with MATLAB/Simulink. The modelling method is similar to the PFL signals presented in this work (cf. Sec. 4.2), but it also contains additional elements for modelling feedback loops and concurrent systems. Besides that, TPT explicitly allows taking decisions in the individual test cases, hence facilitating the representation of a set of test cases in one model. The test cases represented by such a model do not have to be similar.

The proposed methodology is intentionally designed to use a more single test case centered view, by not allowing feedback and conditional flows, i.e. each behaviour model represents one test case. As argued in Section 4.2, the main reason was the growing complexity of such automata, when more and more runs are introduced and in the same time, one has to exclude the prohibited ones. Moreover, the proposed work included more precise levels of abstraction, which are important for the test assessment. In [32] Bringmann does not elaborate on the test assessment methodology in detail. While he discusses the necessity of abstraction, he presents little details on how this abstraction should take place. Although he mentions a concept of similarity, it does not become clear how this similarity is defined and checked.

Regarding the implementation, both approaches, cTTCN-3 and TPT, bring their own stand-alone applications. These can connect to modelling environments as Simulink via data import and export or directly. Still, this means that an additional tool has to be added to the overall tool chain, which can lead to glitches and dependencies with regard to versions, builds, etc. In contrast to this, our work aims to keep the amount of different tools as low as possible. Hence, the proposed solution is directly implemented with MATLAB avoiding the use of any additional Tool Boxes. That is, new functionality can be added without increasing the amount of tools. A further advantage of this approach is the utilisation of several build-in features of MATLAB, e.g. working with vectorised functions on large arrays.

In Section 4.2 it is shown that timed automata [6] or hybrid systems [25] could be used for the abstract models' definition. The main idea is that events and phases are described by the logical formulas allowed as transition guards in these formalisms. In the case of timed automata, our semidecidable method would then become decidable. The relation of the presented modelling method to timed and hybrid automata is not yet understood in all detail. The impact of a potential link between hybrid automata and the presented technique on the decidability of the latter would be an interesting field of investigation.

Dang and Nahhal use hybrid automata as specification for conformance tests in

[14, 35]. They, too, use a behaviour based approach, comparing in-/output behaviour of a specification automaton and an SuT. In contrast to the approach presented in this work, they rely on the accurate signal and focus on coverage and validation of “completeness”. For this, they suggest a relation called star discrepancy that measures the distribution of test case trajectories.

In [21, 22, 30], Girard, Julius, Pappas, et al. present methods for robust testing of hybrid systems using hybrid automata. The question how to obtain such a hybrid automaton from a systems behaviour is not covered there. In the proposed work, a methodology to obtain an abstract model-based on the system’s behaviour without the need of constructing an automaton is described.

Kuipers introduced qualitative simulation, which takes a different approach to modelling of physical systems [31]. This approach is based on the so called qualitative differential equations, which allow a dynamic, yet abstract view on physical systems. Similar to ordinary differential equations, qualitative differential equations describe a systems behaviour with help of changes of state variables. The change itself can be either increasing, steady or decreasing, and additional constraints can be stated. A set of qualitative differential equations comprises all solutions of the ordinary ones that comply to them. Convoluting such qualitative differential equations leads to a of possible and diverging behaviours. Apart from that, there seems to be no methodology to refine such qualitative models systematically, a problem addressed in this work.

As mentioned before, an important aspect when talking about continuous test data is test coverage of continuous systems. Interesting approaches by Dang and Nahhal, and Girard et al. are presented in e.g. [14, 22]. However, this issue is out of scope of this work. One direction in which research could be perpetuated based on the work presented here will be discussed in Sec. 12.2.

11. Summary

Model-based approaches in the development of embedded software become more and more wide spread. In these approaches, different models of one target system are the central elements. The most common algorithm modelling technique used in engineering are block diagrams. In these models, blocks represent the algorithm's mathematical functions which are connected by signal lines representing the variables. One popular tool for modelling and simulating block diagrams is The MathWorks Simulink.

Recently, a growing number of algorithms and functionality of embedded software are designed using such model-based approaches, a prominent one is Rapid Control Prototyping (RCP). Its main advantage is the possibility to develop functionality of both controlling embedded software and controlled system in one modelling environment. Additionally, the developer can simulate the whole system and improve its performance, debug algorithms, etc. in early development stage.

In all development stages of RCP iterative testing takes place, yielding different test results and system behaviours. Important test phases are Software in the Loop (SiL), and Hardware in the Loop (HiL). In this work, we have discussed two problems that may occur during these phases:

1. Taking into account the continuous characteristics of system variables, how can we safeguard a consistent evolution of the development artefacts?
2. How can we analyse correctly test results obtained from real time test devices that do not have certified clocks?

For the first question, test results are compared in order to safeguard a consistent evolution of the development artefacts. Since different artefacts were tested at different stages of the development under different conditions, we cannot expect the results of each test case to be exactly equal, but at most similar. This similarity check of continuous systems is addressed in this work.

The methodology presented here is based on models of system behaviour with different levels of abstraction. A behaviour is represented by the input and output signals of a system and their interrelationship. Thus, we obtain a black-box model of our system in the time domain. A collection of such behaviours enables the observation to get more insight into the systems nature. On each level of abstraction, a system's model is put up by a set of behaviours, each of which consists of input signals and the according output signals created by the system.

The description of the signals themselves varies strongly, and was originally driven by the needs of requirement capturing for dynamic continuous systems. In early development stages, we use the most abstract models of the PFL, representing a signal by a sequence of natural language, paraphrasing the mathematical properties. Here, neither absolute time nor exact function values have to be used. We refine the abstract models by adding first information on time, then on exact signal values. On the most accurate level SFL, we describe a signal by piecewise defined functions, again representing the mathematical properties.

Here, we apply these models to represent and evaluate the behaviour of a SuT, and use the models on PFL as a reference to compare it with a oracle's behaviour. For this comparison we have to introduce a similarity relation with respect to an abstract model. Two systems are similar with respect to an abstract model A, when their behaviours conform to this abstract model A. Conformity is given, if all properties defined in the PFL-model's signals can be found in the measured data, in the correct order and at appropriate points of time.

Thus, the central question is how we can find properties in measured signal data. The proposed work describes a method in which signal snippets are used to a property on the SFL, i.e. as a mathematical function on a time interval. To find one snippet in a set of measured signal data, we compute the cross correlation of the interpolated measured data and the snippet. By this, we find on which time interval of the measured data one property potentially occurs and refine this result by computing the mean square error of both signals on this interval. Repeating this for all properties yields all occurrences of the properties in the system's abstract behaviour model.

After all the properties are localised, we proceed to the next step and validate whether the found properties' order is the same as specified in the abstract model. If this is the case for all specified property flows, we examine the points in time of the occurrences. These have to lie in the specified interval of the property, which can be inferred from the timing constraints in the property flow. After successfully checking these conditions for all specified behaviours of both systems, we know that these systems conform the abstract model and therefore are similar with respect to the abstract model.

An important application of the similarity and the conformance relations is evaluating test cases in regression test. Here, the older development artefact, e.g. a Simulink model, serves as oracle, while the SuT is a newer artefact generated during the development process, e.g. automatically generated C code compiled to an executable. Both systems are stimulated with the same inputs, which conform to the abstract model's inputs. If both systems' outputs also conform to the abstract outputs, the systems are similar with respect to this model. In case that one output of the SuT does not conform, the regression test found a faulty behaviour of the SuT. The feasibility of this method is demonstrated in an example using measured signal data from a Simulink simulation model. In addition, the similarity relation can also be used for other behavioural tests, especially when the user has a library of snippets. A brief outlook on this topic will be discussed in Chapter 12.

When dealing with the second problem, the validation of test hardware timers, we can apply conformance checking, too. The motivation here are possible differences in sample timing due to not exactly working hardware timers. Due to this, a drift occurs in the sampled data which leads to distorted signals and seemingly wrong timings of events. The original timing of these pieces was obtained in test runs with certified reference test hardware. In both test results, we search for certain properties, and obtain the points in time of the occurrences. Using these time information, we can estimate the difference of the sample times, i.e. the drift. For the search for properties based we use the cross correlation approach already used for regression tests. We have shown the feasibility of this approach using a Simulink/Stateflow model in order to work with an undisturbed signal without jitter. We discussed the example in Chapter 9.

12. Future Work

In this work a methodology to compare signals of test measurements with respect to an abstract behavioural model is proposed. The following closing remarks will throw a light on possible steps to continue the work.

The components of the proposed methodology focuses especially on the computational and modelling aspects. Arbitrary signal snippets are chosen as reference signals for those computations. The process of obtaining those such signal snippets is not covered in depth. Additionally, testing continuous behaviour with respect to an abstract behaviour in the proposed way poses the question of test case coverage:

- How many test cases are required to cover one abstractly modelled behaviour?
- How well does a given set of behaviours describe the target system?

Dealing with aspects such as reference signal generation or test coverage are two possible ways to enhance, augment, and continue this work.

12.1. Reference Signal Snippets

As indicated in Section 7.1, reference signal generation is not the main focus in this work. Nevertheless we have to keep this topic in the back of our mind, since these snippets are substantial for the conformance check. The proposed initial approach to obtain this snippets is to choose fundamental mathematical functions such as constant values, linear polynomials, etc. Furthermore, this initial approach only defines one sequence of these functions, i.e. one snippet per property. The final result, however, needs to be picked from these functions by hand. For further investigation, two approaches seem promising:

1. Interactive collection of snippets in a library
2. Parametrisable templates

During the model-based development process several runs of simulations and X-in-the-loop tests are performed, yielding signal data. Over time, numerous development cycles for different products take place, creating a wide variety of domain-specific data which is typical for a company or department. From this data, developers can choose good examples for certain properties. By associating this signal intervals with according property descriptions in a database, a development team, department, etc. would be able to create a collection of typical signal snippets. Such snippets data base can be used afterwards for conformity checks with respect to to an abstract behavioural model.

The second approach makes use of a template for each property, which can be parametrised by the user. The general idea here is to determine typical properties of signals used in tests and produce a template for each property. For the conformity check, the user can assign a set of parameters to the respective property templates to instantiate an according set of snippets. Examples of parameters that could be used for instantiation of a snippet are initial value, ending value, duration, derivatives, etc.

Realistically, a combination of both approaches should prove as most promising. A collection of snippets collected during tests and simulations would be used to obtain templates and typical parameters. These now domain-specific templates, in turn, would be used to instantiate new snippets.

12.2. Test Case Coverage

When it comes to software intense embedded system, several interesting problems stem from the combination of discrete and continuous mathematics. The discrete one governs software and computer environments, while the continuous one the physical world. While addressing the aspect of comparability of dynamic behaviour in both mathematical domains in this work, it is observed that the problem of test coverage is one worth further investigation. The central questions driving this problem are discussed in the following.

The first one, being the more specific one, asks for the number of test cases needed to test a system exhibiting continuous behaviour. This can be reformulated to the question whether there is a metric telling how much of an abstractly modelled behaviour is covered by certain accurate test cases. When checking the conformance and similarity of a system, we try to assess the developed algorithm with a finite number of test cases. Given a system with at least one continuous variable, we have an innumerable set of possible behaviours. Thus, even when focussing only on one abstractly described behaviour, the number of accurate behaviours included in it cannot be enumerated.

The second question takes this topic one step further by asking how well a system's behaviours can be described by a given set of behaviours. For a verification process, an exhaustive evaluation of all input signal functions would be the ideal, yet unfeasible approach. To simplify this, the system's specific dynamics can be taken into account, e.g. not all inputs make sense for physical reasons. For testing purposes we have to restrict ourselves to those realistic cases. Still, because of the aforementioned question this leaves us with a potentially innumerable number of test cases.

As mentioned in Chapter 10, Dang and Nahhal suggested a relation called star discrepancy to measure test coverage of continuous and hybrid systems [14, 35]. With star discrepancy one can determine whether a set is distributed uniformly in a variable space. Thus, Dang and Nahhal's approach estimates boundaries of the coverage of the potentially reachable space using star discrepancy. This method could also be applied in our case. The basic idea is to restrict the reachable space to the values defined by the abstract behavioural model. Additionally, one could take into account the evolution of the input and output signals in form of the derivatives.

A brief overview over other approaches to the coverage problem can be found in [30]. Here, as in [21, 22], similar methods as applied by Dang and Nahhal are used to measure the coverage of robust test cases for hybrid systems. As mentioned in Chapter 10, tests are performed on hybrid systems. Combining this approaches with the modelling method presented here, might be an interesting enhancement of robust test case generation.

Part VI.

Appendix

A. Computation of Integrals I_l

In Section 7.2.2, in Equation (7.6) cross correlation is described with a sum of integrals I_l:

$$I_l = \int_{t'_l}^{t'_{l+1}} \sum_{i,j=0}^{n} x_i\left(t\right) t^i \cdot y_j\left(t-\tau\right)\left(t-\tau\right)^j \; dt \tag{A.1}$$

There, we also introduced the monotonically rising sequence of points of time

$$\mathcal{T}\left(\tau\right) = \left(t'_0, \ldots, t'_m\right) \text{ with } t'_l \in T\left(\tau\right), l \in \left\{0, \ldots, m\right\} \tag{A.2}$$

where

$$\begin{aligned} T\left(\tau\right) = \{&t \in T_x \cup T_y\left(\tau\right) \mid \\ &t \geq \max\left(\min\left(T_x\right), \min\left(T_y\left(\tau\right)\right)\right) \wedge t \leq \min\left(\max\left(T_x\right), \max\left(T_y\left(\tau\right)\right)\right)\} \end{aligned} \tag{A.3}$$

T_x is the set of sample points of time of the reference signal, and $T_y\left(\tau\right)$ is the corresponding set of the property signal snippet shifted by τ.

From this construction of $\mathcal{T}\left(\tau\right)$ we can infer that the coefficients of the piecewise interpolation are constant on each interval that is enclosed by two adjacent elements of $\mathcal{T}\left(\tau\right)$. Hence, for $t \in \left[t'_l, t'_{l+1}\right]$ we can substitute

$$x_i\left(t\right) = x_{t'_l,i} \qquad\qquad y_i\left(t\right) = y_{t'_l,i} \tag{A.4}$$

and get

$$I_l = \int_{t'_l}^{t'_{l+1}} \sum_{i,j=0}^{n} x_{t'_l,i} t^i \cdot y_{t'_l-\tau,i} \left(t-\tau\right)^j \, dt \tag{A.5}$$

$$I_l = \sum_{i,j=0}^{n} \int_{t'_l}^{t'_{l+1}} x_{t'_l,i} t^i \cdot y_{t'_l-\tau,i} \left(t-\tau\right)^j \, dt \tag{A.6}$$

$$I_l = \sum_{i,j=0}^{n} x_{t'_l,i} y_{t'_l-\tau,j} \int_{t'_l}^{t'_{l+1}} t^i \left(t-\tau\right)^j dt \tag{A.7}$$

$$I_l = \sum_{i,j=0}^{n} x_{t'_l,i} y_{t'_l-\tau,j} \int_{t'_l}^{t'_{l+1}} \sum_{k=0}^{j} (-1)^j \binom{j}{k} t^{i+k} \tau^{j-k} dt \tag{A.8}$$

$$I_l = \sum_{i,j=0}^{n} x_{t'_l,i} y_{t'_l-\tau,j} \sum_{k=0}^{j} (-1)^j \binom{j}{k} \int_{t'_l}^{t'_{l+1}} t^{i+k} \tau^{j-k} dt \tag{A.9}$$

$$I_l = \sum_{i,j=0}^{n} x_{t'_l,i} y_{t'_l-\tau,j} \sum_{k=0}^{j} (-1)^j \binom{j}{k} \tau^{j-k} \int_{t'_l}^{t'_{l+1}} t^{i+k} dt \tag{A.10}$$

$$I_l = \sum_{i,j=0}^{n} x_{t'_l,i} y_{t'_l-\tau,j} \sum_{k=0}^{j} (-1)^j \binom{j}{k} \tau^{j-k} \; \frac{1}{i+k+1} t^{i+k+1} \Big|_{t'_l}^{t'_{l+1}} \tag{A.11}$$

Setting $\delta_l = t'_{l+1} - t'_l$ we obtain

$$I_l = \sum_{i,j=0}^{n} x_{t'_l,i} y_{t'_l-\tau,j} \sum_{k=0}^{j} \frac{(-1)^j}{i+k+1} \binom{j}{k} \delta_l^{i+k+1} \tau^{j-k} \tag{A.12}$$

Bibliography

[1] OpenOME. URL `http://www.cs.toronto.edu/km/openome/OpenOME.html`. accessed 2009-10-16.

[2] Piketec tpt. URL `http://www.piketec.com/products/tpt.php?lang=de`. accessed 2012-07-10.

[3] ZAMOMO web page. URL `http://web.embedded.rwth-aachen.de/zamomo/`. accessed 2010-06-22.

[4] The MathWorks homepage. URL `http://www.mathworks.com`. accessed 2012-01-23.

[5] D. Abel and A. Bollig. *Rapid Control Prototyping*. Springer, 2006.

[6] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994. URL `citeseer.ist.psu.edu/alur94theory.html`.

[7] José Manuel Bischof. Paths, distances and eccentricity. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.169.7351&rep=rep1&type=pdf`. accessed 2012-07-10.

[8] Barry Boehm. Guidelines for verifying and validating software requirements and design specifications. In *EURO IFIP 79*. North-Holland Publishing Co., 1979.

[9] Jens Bräuer, Henning Kleinwechter, and Andreas Leicher. μTTCN - an approach to continuous signals in TTCN-3. In *Software Engineering (Workshops)*, volume 106 of *LNI*, pages 55–64. GI, 2007.

[10] Eckard Bringmann and Andreas Krämer. Systematic testing of the continuous behavior of automotive systems. In *Proceedings of the 2006 international workshop on Software engineering for automotive systems*, SEAS '06, pages 13–20, New York, NY, USA, 2006. ACM. ISBN 1-59593-402-2. doi: http://doi.acm.org/10.1145/1138474.1138479. URL `http://doi.acm.org/10.1145/1138474.1138479`.

[11] Ana R. Cavalli, Jean Philippe Favreau, and Marc Phalippou. Formal methods for conformance testing: Results and perspectives. In *Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems VI*, pages 3–17, Amsterdam, The Netherlands, The Netherlands, 1994. North-Holland Publishing Co. ISBN 0-444-81697-6.

[12] Mirko Conrad. *Modell-basierter Test eingebetteter Software im Automobil.* PhD thesis, TU Berlin, 2004.

[13] Mirko Conrad, Ines Fey, Matthias Grochtmann, and Torsten Klein. Modellbasierte Entwicklung eingebetteter Fahrzeugsoftware bei DaimlerChrysler. In Bernhard Rumpe and Wolfgang Hesse, editors, *Modellierung*, volume 45 of *LNI*, pages 31–41. GI, 2004. URL `http://dblp.uni-trier.de/db/conf/modellierung/modellierung2004.html#KleinCFG04`.

[14] Thao Dang and Tarik Nahhal. Coverage-guided test generation for continuous and hybrid systems. *Formal Methods in System Design*, 34(2):183–213, 2009. doi: http://dx.doi.org/10.1007/s10703-009-0066-0.

[15] Peter Drews, Frank-Josef Heßeler, Kai Hoffmann, Dirk Abel, Dominik Schmitz, Andreas Polzer, and Stefan Kowalewski. Entwicklung einer Luftpfadregelung am Dieselmotor unter Berücksichtigung nichtfunktionaler Anforderungen. In *AUTOREG 2008 - Steuerung und Regelung von Fahrzeugen und Motoren*, pages 91–102. VDI-Berichte Nr. 2009, 4. Fachtagung Baden-Baden, 2008.

[16] dSPACE. Simulation modes. URL `http://www.dspace.de/de/gmb/home/products/sw/pcgs/simulation.cfm`. accessed January 30, 2011.

[17] Tom Egel, Michael Burke, Michael Carone, and Wensi Jin. Applying model-based design to commercial vehicle electronics systems. *SAE International Journal of Commercial Vehicles*, 1(1):392–396, 2009. URL `http://tagteamcontent.mathworks.com/tt/sl.ashx?z=501ba437&dataid=11241&ft=1`.

[18] *TTCN-3 Standards.* ETSI CTI. URL `http://www.ttcn-3.org/StandardSuite.htm`.

[19] *Protocol and profile conformance testing specifications.* European Telecommunications Standards Institute, 1995. `http://portal.etsi.org/mbs/testing/conformance/conformance.htm`.

[20] C. Gips and H.-W. Wiesbrock. Notation und Verfahren zur automatischen Überprüfung von temporalen Signalabhängigkeiten und -merkmalen für modellbasiert entwickelte Software. In *MBEES 2007*, 2007.

[21] A. Girard and G. J. Pappas. Approximation metrics for discrete and continuous systems. *Automatic Control, IEEE Transactions on*, 52(5):782–798, May 2007. ISSN 0018-9286. doi: 10.1109/TAC.2007.895849.

[22] Antoine Girard, A. Agung Julius, and George J. Pappas. Approximate simulation relations for hybrid systems. *Discrete Event Dynamic Systems*, 18(2): 163–179, 2008. doi: http://dx.doi.org/10.1007/s10626-007-0029-9.

[23] Jürgen Großmann, Ina Schieferdecker, and Hans-Werner Wiesbrock. Modeling property based stream templates with TTCN-3. In Kenji Suzuki, Teruo Higashino, Andreas Ulrich, and Toru Hasegawa, editors, *TestCom/FATES*, volume 5047 of *LNCS*, pages 70–85. Springer, 2008. ISBN 978-3-540-68514-2. doi: 10.1007/978-3-540-68524-1_7.

[24] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8:231–274, June 1987. ISSN 0167-6423. doi: 10.1016/0167-6423(87) 90035-9. URL `http://portal.acm.org/citation.cfm?id=34884.34886`.

[25] T. A. Henzinger. The theory of hybrid automata. In *LICS '96: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, page 278, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7463-6. doi: http://doi.ieeecomputersociety.org/10.1109/LICS.1996.561342.

[26] K. Hoffmann, F. Hesseler, and D. Abel. Rapid control prototyping with Dymola and Matlab for a model predictive control for the air path of a boosted diesel engine. In *Proc. of E-COSM - Rencontres Scientifiques de l'IFP*, pages 33–40, October 2006.

[27] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Prentice Hall, 2007.

[28] Informatik V (Information Systems), RWTH Aachen University. ConceptBase. URL `http://www-i5.informatik.rwth-aachen.de/CBdoc/`.

[29] *ISO/IEC13210: Standard for Information Technology Test Methods for Measuring Conformance to POSIX.* Institute of Electrical and Electronics Engineers, 1994. `http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4140774`.

[30] A. Agung Julius, Georgios E. Fainekos, Madhukar Anand, Insup Lee, and George J. Pappas. Robust test generation and coverage for hybrid systems. In Alberto Bemporad, Antonio Bicchi, and Giorgio C. Buttazzo, editors, *HSCC*, volume 4416 of *Lecture Notes in Computer Science*, pages 329–342. Springer, 2007. ISBN 978-3-540-71492-7. doi: http://dx.doi.org/10.1007/978-3-540-71493-4_27.

[31] B. Kuipers. Qualitative simulation. In R. E. Meyers, editor, *Encyclopedia of Physical Science and Technology*, pages 287–300. Academic Press, third edition, 2001.

[32] Eckard Lehmann. *Time Partition Testing – Systematischer Test des kontinuierlichen Verhaltens von eingebetteten Systemen*. PhD thesis, Technische Universität Berlin, 2003.

[33] J. Lunze. *Automatisierungstechnik*. Oldenbourg, München, 2003.

[34] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis. Telos - representing knowledge about information systems. *ACM Transactions on Information Systems*, 8(4):325–362, October 1990.

[35] Tarik Nahhal and Thao Dang. Test coverage for continuous and hybrid systems. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 449–462. Springer, 2007. ISBN 978-3-540-73367-6. doi: http://dx.doi.org/10.1007/978-3-540-73368-3_47.

[36] Object Management Group. OMG SysML. URL `http://www.omgsysml.org`. `http://www.omgsysml.org`.

[37] Jacob Palczynski and Stefan Kowalewski. Early behaviour modelling for control systems. In *UKSIM European Symposium on Computer Modeling and Simulation*, volume 0, pages 148–153, Los Alamitos, CA, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3886-0. doi: http://doi.ieeecomputersociety.org/10.1109/EMS.2009.69.

[38] Jacob Palczynski, Carsten Weise, and Stefan Kowalewski. Testing continuous systems conformance using cross correlation. In *Proceedings of the 22nd IFIP International Conference on Testing Software and Systems: Short Papers*, pages 31–36, Montréal, 2010. CRIM. ISBN 978-2-89522-136-4.

[39] Jacob Palczynski, Carsten Weise, Stefan Kowalewski, and Daniel Ulmer. Estimation of clock drift in HiL testing by property-based conformance check. In *Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW, TAIC PART)2011*, pages 590 – 595. IEEE Computer Society, 2011. ISBN 978-1-4577-0019-4.

[40] Jacob Palczynski, Carsten Weise, Sebastian Moj, and Stefan Kowalewski. Comparing continuous behaviour in model-based development of embedded software. In *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VII (MBEES 2011)*, pages 61 – 70. fortiss GmbH, 2011.

[41] Mario G. Pehinschi, Srikanth Gururajan, Bojan Cukic, and Marcello Napolitano. Investigation of issues in the conversion of simulink models to ANSI C code for a neuronal network based adaptive control system. Technical report, West Virginia University / NASA IV&V, December 2005. URL `http://sarpresults.ivv.nasa.gov/ViewResearch/91.jsp`.

[42] Jörg Schäuffele and Thomas Zurawka. *Automotive Software Engineering.* ATZ-MTZ-Fachbuch. Vieweg, 2003.

[43] Ina Schieferdecker and Jürgen Großmann. Testing hybrid control systems with TTCN-3: an overview on continuous TTCN-3. *Int. J. Softw. Tools Technol. Transf.*, 10(4):383–400, 2008. ISSN 1433-2779. doi: http://dx.doi.org/10.1007/s10009-008-0081-2.

[44] Ina Schieferdecker, Eckard Bringmann, and Jürgen Großmann. Continuous TTCN-3: testing of embedded control systems. In *SEAS '06: Proceedings of the 2006 international workshop on Software engineering for automotive systems*, pages 29–36, New York, NY, USA, 2006. ACM. ISBN 1-59593-402-2. doi: http://doi.acm.org/10.1145/1138474.1138481.

[45] Julien Schmaltz and Jan Tretmans. On conformance testing for timed systems. In Franck Cassez and Claude Jard, editors, *FORMATS*, volume 5215 of *Lecture Notes in Computer Science*, pages 250–264. Springer, 2008. ISBN 978-3-540-85777-8.

[46] D. Schmitz, P. Drews, F. Hesseler, M. Jarke, S. Kowalewski, J. Palczynski, A. Polzer, M. Reke, and T. Rose. Model-based requirements capture for software-based control systems (in German). In *Software Engineering*, LNI, Munich, Germany, 2008.

[47] D. Schmitz, H. W. Nissen, M. Jarke, T. Rose, P. Drews, F. J. Hesseler, and M. Reke. Requirements engineering for control systems development in small and medium-sized enterprises. In *16th Int. Requirements Engineering Conf.*, pages 229–234, Barcelona, Spain, September 2008. IEEE.

[48] D. Schmitz, , M. Zhang, T. Rose, M. Jarke, A. Polzer, J. Palczynski, S. Kowalewski, and M. Reke. Mapping requirements models to mathematical models in control systems development. In *Proc. of 5th European Conference on Model Driven Architecture Foundations and Applications (ECMDA-FA 2009)*, pages 253–264, Enschede, The Netherlands, 2009. Springer.

[49] Dominik Schmitz, Peter Drews, Frank Hesseier, Matthias Jarke, Stefan Kowalewski, Jacob Palczynski, Andreas Polzer, Michael Reke, and Thomas Rose. Modellbasierte Anforderungserfassung für softwarebasierte Regelungen.

In Korbinian Herrmann and Bernd Brügge, editors, *Software Engineering*, volume 121 of *LNI*, pages 257–271. GI, 2008. ISBN 978-3-88579-215-4.

[50] I. Stürmer and M. Conrad. Test suite design for code generation tools. In *Proc. 18th IEEE International Conference on Automated Software Engineering*, pages 286–290, 2003. doi: 10.1109/ASE.2003.1240322.

[51] Ingo Stürmer, Daniela Weinberg, and Mirko Conrad. Overview of existing safeguarding techniques for automatically generated code. *SIGSOFT Softw. Eng. Notes*, 30(4):1–6, 2005. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/1082983.1083192.

[52] Wolfgang Thomas. Automata on infinite objects. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 133–192. 1990.

[53] Gerrit Jan Tretmans. *A formal approach to conformance testing.* PhD thesis, Enschede, December 1992. URL `http://doc.utwente.nl/58114/`.

[54] Jan Tretmans. A formal approach to conformance testing. In *Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems VI*, pages 257–276, Amsterdam, The Netherlands, The Netherlands, 1994. North-Holland Publishing Co. ISBN 0-444-81697-6.

[55] *Conformance Testing and Certification Model for W3C Specifications.* World Wide Web Consortium, 2002. `http://www.w3.org/QA/2002/01/Note-qa-certif-20020102.html`.

[56] Justyna Zander-Nowicka. *Model-based Testing of Real-Time Embedded Systems in the Automotive Domain.* PhD thesis, Technische Universität Berlin, 2008. URL `http://opus.kobv.de/tuberlin/volltexte/2009/2186/pdf/zandernowicka_justyna.pdf`.

[57] Justyna Zander-Nowicka, Abel Marrero Pérez, and Ina Schieferdecker. From functional requirements through test evaluation design to automatic test data patterns retrieval - a concept for testing of software dedicated for hybrid embedded systems. In Hamid R. Arabnia and Hassan Reza, editors, *Software Engineering Research and Practice*, pages 347–353. CSREA Press, 2007. ISBN 1-60132-034-5.

[58] Justyna Zander-Nowicka, Xuezheng Xiong, and Ina Schieferdecker. Systematic test data generation for embedded software. In Hamid R. Arabnia and Hassan Reza, editors, *Software Engineering Research and Practice*, pages 164–170. CSREA Press, 2008. ISBN 1-60132-088-4.

Acronyms

DFA Deterministic Finite Automaton. 17

ECU Electronic Control Unit. 10, 19

HiL Hardware in the Loop. 11, 14, 28, 103

MBD Model Based Development. 3, 4
MSE mean square error. 21

NFA Nondeterministic Finite Automaton. 18, 43

PFL Property Flow Layer. 34–38, 53

RCP Rapid Control Prototyping. 3, 4, 11, 22, 24, 28, 103

SFL Signal Function Layer. 34–36, 38, 53
SiL Software in the Loop. 28, 103
SuT system under test. 10

TFL Transfer Function Layer. 34, 53
TML Time Metric Layer. 34–38, 53
TPT Time Partition Testing. 29, 100
TTCN-3 Testing and Test Control Notation Version 3. 29, 99

ZAMOMO Verzahnung von modellbasierter Softwareentwicklung und modellbasiertem Reglerentwurf. 6, 53

Aachener Informatik-Berichte

This list contains all technical reports published during the past three years. A complete list of reports dating back to 1987 is available from:

`http://aib.informatik.rwth-aachen.de/`

To obtain copies please consult the above URL or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen,
Email: biblio@informatik.rwth-aachen.de

2010-01 *	Fachgruppe Informatik: Jahresbericht 2010
2010-02	Daniel Neider, Christof Löding: Learning Visibly One-Counter Automata in Polynomial Time
2010-03	Holger Krahn: MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering
2010-04	René Wörzberger: Management dynamischer Geschäftsprozesse auf Basis statischer Prozessmanagementsysteme
2010-05	Daniel Retkowitz: Softwareunterstützung für adaptive eHome-Systeme
2010-06	Taolue Chen, Tingting Han, Joost-Pieter Katoen, Alexandru Mereacre: Computing maximum reachability probabilities in Markovian timed automata
2010-07	George B. Mertzios: A New Intersection Model for Multitolerance Graphs, Hierarchy, and Efficient Algorithms
2010-08	Carsten Otto, Marc Brockschmidt, Christian von Essen, Jürgen Giesl: Automated Termination Analysis of Java Bytecode by Term Rewriting
2010-09	George B. Mertzios, Shmuel Zaks: The Structure of the Intersection of Tolerance and Cocomparability Graphs
2010-10	Peter Schneider-Kamp, Jürgen Giesl, Thomas Ströder, Alexander Serebrenik, René Thiemann: Automated Termination Analysis for Logic Programs with Cut
2010-11	Martin Zimmermann: Parametric LTL Games
2010-12	Thomas Ströder, Peter Schneider-Kamp, Jürgen Giesl: Dependency Triples for Improving Termination Analysis of Logic Programs with Cut
2010-13	Ashraf Armoush: Design Patterns for Safety-Critical Embedded Systems
2010-14	Michael Codish, Carsten Fuhs, Jürgen Giesl, Peter Schneider-Kamp: Lazy Abstraction for Size-Change Termination
2010-15	Marc Brockschmidt, Carsten Otto, Christian von Essen, Jürgen Giesl: Termination Graphs for Java Bytecode
2010-16	Christian Berger: Automating Acceptance Tests for Sensor- and Actuator-based Systems on the Example of Autonomous Vehicles

2010-17	Hans Grönniger: Systemmodell-basierte Definition objektbasierter Modellierungssprachen mit semantischen Variationspunkten
2010-18	Ibrahim Armaç: Personalisierte eHomes: Mobilität, Privatsphäre und Sicherheit
2010-19	Felix Reidl: Experimental Evaluation of an Independent Set Algorithm
2010-20	Wladimir Fridman, Christof Löding, Martin Zimmermann: Degrees of Lookahead in Context-free Infinite Games
2011-01 *	Fachgruppe Informatik: Jahresbericht 2011
2011-02	Marc Brockschmidt, Carsten Otto, Jürgen Giesl: Modular Termination Proofs of Recursive Java Bytecode Programs by Term Rewriting
2011-03	Lars Noschinski, Fabian Emmes, Jürgen Giesl: A Dependency Pair Framework for Innermost Complexity Analysis of Term Rewrite Systems
2011-04	Christina Jansen, Jonathan Heinen, Joost-Pieter Katoen, Thomas Noll: A Local Greibach Normal Form for Hyperedge Replacement Grammars
2011-06	Johannes Lotz, Klaus Leppkes, and Uwe Naumann: dco/c++ - Derivative Code by Overloading in C++
2011-07	Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe: An Operational Semantics for Activity Diagrams using SMV
2011-08	Thomas Ströder, Fabian Emmes, Peter Schneider-Kamp, Jürgen Giesl, Carsten Fuhs: A Linear Operational Semantics for Termination and Complexity Analysis of ISO Prolog
2011-09	Markus Beckers, Johannes Lotz, Viktor Mosenkis, Uwe Naumann (Editors): Fifth SIAM Workshop on Combinatorial Scientific Computing
2011-10	Markus Beckers, Viktor Mosenkis, Michael Maier, Uwe Naumann: Adjoint Subgradient Calculation for McCormick Relaxations
2011-11	Nils Jansen, Erika Ábrahám, Jens Katelaan, Ralf Wimmer, Joost-Pieter Katoen, Bernd Becker: Hierarchical Counterexamples for Discrete-Time Markov Chains
2011-12	Ingo Felscher, Wolfgang Thomas: On Compositional Failure Detection in Structured Transition Systems
2011-13	Michael Förster, Uwe Naumann, Jean Utke: Toward Adjoint OpenMP
2011-14	Daniel Neider, Roman Rabinovich, Martin Zimmermann: Solving Muller Games via Safety Games
2011-16	Niloofar Safiran, Uwe Naumann: Toward Adjoint OpenFOAM
2011-17	Carsten Fuhs: SAT Encodings: From Constraint-Based Termination Analysis to Circuit Synthesis
2011-18	Kamal Barakat: Introducing Timers to pi-Calculus
2011-19	Marc Brockschmidt, Thomas Ströder, Carsten Otto, Jürgen Giesl: Automated Detection of Non-Termination and NullPointerExceptions for Java Bytecode

2011-24	Callum Corbett, Uwe Naumann, Alexander Mitsos: Demonstration of a Branch-and-Bound Algorithm for Global Optimization using McCormick Relaxations
2011-25	Callum Corbett, Michael Maier, Markus Beckers, Uwe Naumann, Amin Ghobeity, Alexander Mitsos: Compiler-Generated Subgradient Code for McCormick Relaxations
2011-26	Hongfei Fu: The Complexity of Deciding a Behavioural Pseudometric on Probabilistic Automata
2012-01	Fachgruppe Informatik: Annual Report 2012
2012-02	Thomas Heer: Controlling Development Processes
2012-03	Arne Haber, Jan Oliver Ringert, Bernhard Rumpe: MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems
2012-04	Marcus Gelderie: Strategy Machines and their Complexity
2012-05	Thomas Ströder, Fabian Emmes, Jürgen Giesl, Peter Schneider-Kamp, and Carsten Fuhs: Automated Complexity Analysis for Prolog by Term Rewriting
2012-06	Marc Brockschmidt, Richard Musiol, Carsten Otto, Jürgen Giesl: Automated Termination Proofs for Java Programs with Cyclic Data
2012-07	André Egners, Björn Marschollek, and Ulrike Meyer: Hackers in Your Pocket: A Survey of Smartphone Security Across Platforms
2012-08	Hongfei Fu: Computing Game Metrics on Markov Decision Processes
2012-09	Dennis Guck, Tingting Han, Joost-Pieter Katoen, and Martin R. Neuhäußer: Quantitative Timed Analysis of Interactive Markov Chains
2012-10	Uwe Naumann and Johannes Lotz: Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Direct Solvers for Systems of Linear Equations
2012-12	Jürgen Giesl, Thomas Ströder, Peter Schneider-Kamp, Fabian Emmes, and Carsten Fuhs: Symbolic Evaluation Graphs and Term Rewriting — A General Methodology for Analyzing Logic Programs
2012-15	Uwe Naumann, Johannes Lotz, Klaus Leppkes, and Markus Towara: Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Solvers for Systems of Nonlinear Equations
2012-16	Georg Neugebauer and Ulrike Meyer: SMC-MuSe: A Framework for Secure Multi-Party Computation on MultiSets
2012-17	Viet Yen Nguyen: Trustworthy Spacecraft Design Using Formal Methods
2013-01 *	Fachgruppe Informatik: Annual Report 2013
2013-02	Michael Reke: Modellbasierte Entwicklung automobiler Steuerungssysteme in Klein- und mittelständischen Unternehmen
2013-03	Markus Towara and Uwe Naumann: A Discrete Adjoint Model for OpenFOAM

2013-04	Max Sagebaum, Nicolas R. Gauger, Uwe Naumann, Johannes Lotz, and Klaus Leppkes: Algorithmic Differentiation of a Complex C++ Code with Underlying Libraries
2013-05	Andreas Rausch and Marc Sihling: Software & Systems Engineering Essentials 2013
2013-06	Marc Brockschmidt, Byron Cook, and Carsten Fuhs: Better termination proving through cooperation
2013-07	André Stollenwerk: Ein modellbasiertes Sicherheitskonzept für die extrakorporale Lungenunterstützung
2013-08	Sebastian Junges, Ulrich Loup, Florian Corzilius and Erika Ábrahám: On Gröbner Bases in the Context of Satisfiability-Modulo-Theories Solving over the Real Numbers
2013-10	Joost-Pieter Katoen, Thomas Noll, Thomas Santen, Dirk Seifert, and Hao Wu: Performance Analysis of Computing Servers using Stochastic Petri Nets and Markov Automata
2013-12	Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl: Alternating Runtime and Size Complexity Analysis of Integer Programs
2013-13	Michael Eggert, Roger Häußling, Martin Henze, Lars Hermerschmidt, René Hummen, Daniel Kerpen, Antonio Navarro Pérez, Bernhard Rumpe, Dirk Thißen, and Klaus Wehrle: SensorCloud: Towards the Interdisciplinary Development of a Trustworthy Platform for Globally Interconnected Sensors and Actuators

* These reports are only available as a printed version.
Please contact biblio@informatik.rwth-aachen.de to obtain copies.