Hardware-in-the-loop Simulation Framework

A dissertation submitted to the
Swiss Federal Institute of Technology (ETH)
Zurich

for the degree of
Doctor of Technical Sciences

presented by
Marco Aurelio Antonio Sanvido
Dipl. Informatik-Ing. ETH
born 03.01.1972
citizen of Lugano, Switzerland

accepted on the recommendation of
Prof. W. Schaufelberger, examiner
Prof. N. Wirth, co-examiner

March 2002
Considerate la vostra semenza:
Fatti non foste a viver come bruti
ma per sequir virtute e conoscenza.

Dante

_a Ciotolo_

---

Dante Alighieri, _La Divina Commedia_. Inf. XXVI, 118-120.
Consider ye the seed from which ye sprang;
Ye were not made to live like unto brutes,
But for pursuit of virtue and of knowledge.
Acknowledgment

First of all I would like to express my most sincere thanks to my advisors Prof. Schaufelberger and Prof. Wirth for their support during my dissertation, for their wise advise but also for giving me the possibility to work in an interdisciplinary project like the OLGA project. Furthermore, I would also like to thank the Heli-Team of the ETH Zurich, more precisely Christoph Eck, Markus Kottmann and Jacques Chapuis who gave me their confidence in using the HelyOS system and the MICOLA language for flying the helicopter. Not forgetting all their advice and helpful hints.

I would like to thank all the people at the Institute for Computer Systems – ICS – and at the Automatic Control Laboratory – IfA – for their friendship, support and fruitful discussions. In particular, my special thanks go to: Francesco Borrelli, Vaclav Cechticky, Gültekin Erdem, Sašo Jezernik, Pieter Muller, Patrik Reali and Jacques Supcik. I would further like to thank all the students who contributed in one or the other way with their semester and diploma projects. They are D. Satz [68], S. Suter [70], as well as R. Samedani and C. Schär [69].

I’m sincerely thankful to Sonja Sutz, Konrad Stadler and Anders Hagström who carefully proofread the manuscript and corrected many language mistakes.

My dearest thanks go to my mother Anna and to my brother Gigi, for their support, for giving me the chance to study at the ETH and for always motivating me in pursuing my studies. Without their help, all this would not have been possible.

Thank you.
Seite Leer / Blank leaf
Embedded control systems - ECSs - are becoming more and more present around us. Compared to usual computer programming implementations they have added problems like robustness, code efficiency, code compactness and safety which make them difficult to implement. Moreover, such systems are often used in critical and hazardous applications where a precise and methodological implementation is mandatory. Most ECSs are therefore very conservative in their implementation and the development costs are much bigger than for usual computer programs. Moreover, it is difficult to replace or correct software errors after shipping, and consequently the ability to test such systems methodologically before shipping is very important.

In this thesis we examine a new approach for testing embedded systems based on temporal logic and fault generation. We added those features to a generic Hardware-in-the-loop (HIL) framework which enables a simpler implementation of application specific HIL simulators. To prove the feasibility of our approach we applied it to the testing of two very different applications, a model helicopter controlled by a customized ECS, and a hydro-powerplant barrage controlled by a small ECS.

Our contribution to the field includes a novel approach for specifying fault sequences and the corresponding response of the embedded control system under test. Due to this addition to the HIL simulator, we are able to test more systematically embedded systems for their safety and robustness in hazardous or critical situations.
Prefazione

I sistemi di controllo integrati stanno acquistando sempre più importanza nel mondo che ci circonda. Lo sviluppo di tali sistemi si differenzia da quello dei sistemi tradizionali per la robustezza, la sicurezza, l’efficienza del codice e da un limitato uso delle risorse. Siccome tali sistemi sono integrati in impianti il cui malfunzionamento può essere estremamente pericoloso per gli utenti e l’ambiente circostante, il design e lo sviluppo si deve basare su un approccio sistematico e metodico tale da ridurre al minimo ogni possibile errore. Lo sviluppo di tali sistemi è perciò fatto con prudenza ed in modo conservativo. Ciò comporta costi molto più alti rispetto a quelli per lo sviluppo di applicazioni tradizionali. Inoltre, poiché la correzione di un errore è difficile se non impossibile dopo la loro messa in funzione, la ricerca di un metodo per testare il corretto funzionamento di tali sistemi diventa di fondamentale importanza.

In questa tesi esaminiamo un nuovo tipo di approccio per testare i sistemi di controllo integrati basato sulla generazione di errori e sulla logica temporale. Questi due aspetti assieme ad un framework generico per la simulazione “Hardware-in-the-loop” (HIL) permette di implementare rapidamente nuovi simulatori HIL dedicati. Per dimostrare tale approccio abbiamo applicato il “framework” a due tipi di applicazioni molto differenti: un modello di elicottero auto-pilotato ed un sistema di controllo di un sbarramento hydro-elettrico.

Il contributo di questa tesi nel capo della simulazione è lo sviluppo di un nuovo approccio per la generazione e la corrispondente analisi della risposta del sistema di controllo sottoposto al test. Grazie a tale aggiunta ad un simulatore HIL generico siamo ora in grado di testare sistematicamente sistemi di controllo integrati e di garantire la loro sicurezza e robustezza in situazioni critiche e pericolose.
Seite Leer / Blank leaf
# Contents

## 1 Introduction  
1.1 Motivation ......................................... 2  
1.2 Contribution ........................................ 2  
1.3 Overview ............................................. 3

## 1 Embedded System Development  

## 2 Embedded Systems  

2.1 Embedded Control Systems  
2.2 Design of a Control System  
2.3 A Survey of Design Tools for Embedded Control Systems  
  2.3.1 Real-time Operating Systems  
  2.3.2 Programming Languages  
  2.3.3 Middleware Frameworks and Design Patterns  
  2.3.4 Automatic Code Generation  
  2.3.5 Software Analysis  
  2.3.6 Verification
2.3.7  Testing ........................................ 14
2.3.8  Real-time Supervisors, Redundant and Backup Systems ............ 14
2.4  Case Study: The OLGA Embedded Control System .................... 14
  2.4.1  OLGA Computer System .................................. 15
  2.4.2  OLGA Software ........................................... 17
  2.4.3  Efficiency ............................................... 18
  2.4.4  MICOLA ................................................ 19
  2.4.5  Commercial Application ................................... 20

II  Embedded System Testing – The Framework .............................. 23
  3  Hardware-in-the-loop Simulators ...................................... 25
     3.1  Architectures ................................................. 25
     3.2  Frameworks .................................................. 26
     3.3  Frameworks for HIL .......................................... 27
           3.3.1  Chosen HIL Hardware Architecture ......................... 28
           3.3.2  HIL Simulation Framework Requirements or Finding the Hot-Spots ... 28
           3.3.3  Building an Application Specific HIL Simulation ................. 29
     3.4  Research and Commercial HIL Simulators .......................... 29
           3.4.1  Motion Simulator ....................................... 30
           3.4.2  TVE ................................................... 30
           3.4.3  HIL Simulators based on Matlab/Simulink and DSPACE ............ 31
     3.5  HIL in Education ............................................. 32

4  Fault Generation, Specification and Test ............................... 33
<table>
<thead>
<tr>
<th>Contents</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2 Real-time Scheduler Features for an HIL Simulator</td>
<td>57</td>
</tr>
<tr>
<td>5.2.1 HIL Resource Access</td>
<td>59</td>
</tr>
<tr>
<td>5.2.2 HIL Frequency</td>
<td>60</td>
</tr>
<tr>
<td>5.2.3 Hard Real-time Scheduler Implementation</td>
<td>61</td>
</tr>
<tr>
<td>5.2.4 Soft Real-time Scheduler Implementation</td>
<td>62</td>
</tr>
<tr>
<td>6 Signal Acquisition and Generation</td>
<td>65</td>
</tr>
<tr>
<td>6.1 HIL Simulation and Signal Generation-Acquisition</td>
<td>65</td>
</tr>
<tr>
<td>6.2 The Language Lola and its Extension LolaX</td>
<td>66</td>
</tr>
<tr>
<td>6.2.1 The Interface Definition</td>
<td>67</td>
</tr>
<tr>
<td>6.2.2 The Back-end</td>
<td>67</td>
</tr>
<tr>
<td>6.2.3 The Front-end</td>
<td>68</td>
</tr>
<tr>
<td>6.2.4 Example</td>
<td>69</td>
</tr>
<tr>
<td>6.3 The XNF Translation</td>
<td>70</td>
</tr>
<tr>
<td>6.4 Is the Analog Case Resolved?</td>
<td>72</td>
</tr>
<tr>
<td>7 Numerical Simulation</td>
<td>73</td>
</tr>
<tr>
<td>7.1 Description of a Process</td>
<td>73</td>
</tr>
<tr>
<td>7.2 System Identification</td>
<td>74</td>
</tr>
<tr>
<td>7.3 The Class Model</td>
<td>75</td>
</tr>
<tr>
<td>7.4 The Class Actuator</td>
<td>76</td>
</tr>
<tr>
<td>7.5 The Class Sensor</td>
<td>76</td>
</tr>
<tr>
<td>7.6 Binding Model, Actuators and Sensors</td>
<td>76</td>
</tr>
<tr>
<td>7.6.1 Static versus Dynamic Binding</td>
<td>76</td>
</tr>
<tr>
<td>7.7 Integration</td>
<td>77</td>
</tr>
</tbody>
</table>
III  Case Studies and Their Implementation 79

8  A Simulator for a Model Helicopter 81

8.1  The Helicopter Dynamics 81

8.1.1  Yahmaha R-50 Based Model 82

8.1.2  OLGA Based Model 83

8.2  The Actuators 85

8.2.1  Manually Actuated Servos 85

8.2.2  RC Servos 85

8.2.3  Replay Servos 89

8.3  The Sensors 89

8.3.1  Analog Gyroscopes and Accelerometers 89

8.3.2  Rotor Hall-Effect Sensor 90

8.4  3D OpenGL Visualizer 92

8.5  Testing and Results 92

8.5.1  Main Rotor Controller Testing 92

8.5.2  Hovering Controller Testing 95

8.5.3  Conclusion 97

9  A Simulator for a Hydro-Powerplant Barrage 99

9.1  The Barrage 99

9.1.1  Simulation Software and Dynamic Model 99

9.1.2  Fault Simulation 101
9.1.3 Results .............................................. 103

IV Conclusions .......................... 107

10 Conclusions .................. 109

10.1 Summary ...................... 109

10.2 Outlook .................... 110

V Appendix .......................... 111

A LolaX ............................ 113

A.1 LolaX EBNF Syntax .......... 113

A.2 LolaX Symbol File Format .. 114

B Replay Data Format ............ 117

C FAUSEL .......................... 119

C.1 FAUSEL EBNF ............... 119

C.2 Fault Object ............... 120

D FPGA Schema ................. 121

D.1 PWM Decoder and Rotor Signal Generator ........................ 121

D.2 LolaX Interface ............ 123

E Module List ................. 125

Bibliography ..................... 127
Chapter 1

Introduction

Due to the rapid development of digital-processor technology, embedded control systems (from now on simply called ECSs) control many devices we use in our daily lives. Moreover, many of these systems operate in safety-critical situations and therefore call for rigorous engineering. This crucial point is the reason why so much research effort is put into the development of methodologies for the design, development, implementation and, in our case, testing of ECS. In developing such products and systems, the testing and not the design usually is the more expensive, time-consuming and difficult activity [54].

Hardware-in-the-Ioop simulation (from now on called HIL simulation or HILS) is a kind of real-time simulation where the input and output signals of the simulator show the same time dependent values as the real process [39, 46]. Such simulators allow us to test the real embedded control system (ECS) under different real working loads and conditions. Other simulation methods do not allow us to test the real embedded control system as a complete system. Often the controller part, which is only about 20-30% of the ECS software [60], or the software components are tested independently. HILS makes it possible to test the complete ECS. Moreover, the temporal logic tester and the fault generator allow us to automate the testing procedure, which otherwise is done manually by an operator.

This dissertation covers an area which so far has not been recognized as a relevant academic research topic. This is proven by the relatively small number of publications and conferences about HIL simulation. There are several ways to test an ECS. Sometimes these multiple techniques are lumped into one “hardware-in-the-loop (HIL) simulation” category. However we prefer to use the definition of “HIL simulation” only for one of these testing techniques:
Hardware-in-the-loop (HIL): The real controller controls a computer simulation of a real system.

Rapid prototype (RP): A computer simulation of the controller, i.e. the controller is implemented on a prototype system. The implementation is usually automatically generated from a control design software.

Virtual prototype: Both the controller and the system under control exist solely as computer models. No real control of the process, nor hardware communication is involved.

1.1 Motivation

In 1997 the Measurement and Control Laboratory (IMRT), the Institute for Computer Systems (ICS) and the Automatic Control Laboratory (IfA) at the ETH Zurich [12] started an interdisciplinary project with the goal to build an autonomously flying model-helicopter. The developed autopilot system was based on a new ECS built by the team and it was fully tailored and optimized for the application. The ECS was called OLGA, an acronym of Oberon Language Goes Airborne. The system will be described in detail in Chapters 2 and 8. During the development stage of the control and navigation algorithms the team realized the necessity of testing the software in advance. The navigation was tested with sampled data and the controller was tested simulating a mathematical model of the helicopter. But when putting it all together the project team realized that the navigation and controller code were only a relatively small portion of the overall system – around 20%. Therefore, they were always forced to test on-ground all the components of the system. It is exactly at this stage that the need arose for a better and efficient way to simulate the system, to reduce the risks of a failure and to speed up the development cycles.

1.2 Contribution

A similar approach has not yet been proposed, neither in industry nor in the academic world. There are some commercial products which provide a common platform for HIL simulation. These platforms are, however, based on dedicated hardware or on COTS (components-off-the-shelf) hardware interfaces. This makes it difficult to adapt them for different HIL applications. Nevertheless, the need for HIL simulation is very high, as demonstrated by the many ad-hoc developed HIL simulators [39].

Our HIL framework is based on four different technologies – software and hardware – to provide a foundation for the construction of dedicated HIL simulators. The four techniques described in the next chapters of the thesis are: An object-oriented approach for modelling the simulated system; an automatic hardware/software interface mechanism for the generation and acquisition
of signals; a real-time layer for the time critical communication between the simulated and the tested system; and an automatic testing environment which enables to check the system for correct implementation.

1.3 Overview

The dissertation is subdivided into three parts. The first part describes how embedded systems are developed and which tools are available. In this part a case study of an ECS for an autonomously flying helicopter is also presented [12].

The second part describes how embedded systems can be tested for correctness using Hardware-in-the-loop, going into the details of the framework.
1 Introduction

The third part describes two example implementations of HIL simulations which are based on this framework. The first is the HIL simulation for the model helicopter mentioned above, and the second is an hydro-powerplant barrage simulation.

The fourth part contains conclusions and outlooks.

In the Appendix detailed information about LolaX, i.e. the extension of the Lola language and about the FAUSEL language is given. Moreover, the implementation of the FPGA circuit for the helicopter HIL is presented and the list of implemented Oberon modules is listed.
Part I

Embedded System Development
Seite Leer / Blank leaf
Chapter 2

Embedded Systems

Before starting to explain the hardware-in-the-loop simulation concept it is important to learn how the ECSs we want to test are built. This Chapter gives a short introduction to embedded systems development techniques and technologies.

2.1 Embedded Control Systems

Embedded control systems are specialized computers used to control devices such as automobiles, home and office appliances, handheld units of any kind as well as sophisticated machines like space vehicles. Often they are small in size and have a fixed set of functions. The operating systems and application functions are often combined in the same program, and are relatively small. Furthermore, embedded control systems are implemented with conservative safety and robustness requirements. This is because many ECSs are used in safety-critical applications where errors cannot be tolerated.

2.2 Design of a Control System

Figure 2.1 shows a schematic view of the design process of a control system. The first step in the design of a control system is to determine the needed actuators, i.e. the forces which apply to the actually controlled machinery, and the needed sensors, i.e. what is measured from the process.

The second step is to understand the system dynamics of the process, i.e. finding a mathematical/physical description of the system, its transfer function etc. There are many different and
well known approaches which help in the identification step [48]. The derived model is then validated comparing it with the system. This step is iterated until a satisfactory model is found.

The third step is the design of the controller. Also for this step many different methods and well known techniques are used in order to find the desired controller performance. However, to validate/test it a first software simulator, i.e. a virtual prototype has to be used in order to check the controller performance.

The fourth step is to test the controller on the real platform. This is usually done implementing the controller on a rapid prototype platform which allows to test the controller of the real system. In cases where the real platform is expensive or the test is safety critical, the first tests are usually done using a hardware-in-the-loop simulator. This is often done using automatic code generation: from a control design tool like Matlab/Simulink, the controller code is downloaded directly to a controller platform, i.e. a rapid prototype system.

The fifth step is the implementation of the real embedded control system – ECS – which will be deployed in the field. This step is lacking of mature methodologies which allow an automatic translation from the rapid prototype to the final system. An ECS is usually more complex than the
controller algorithm itself, since the ECS has to deal also with other tasks, like data transmission and system monitoring, all under hard real-time constraints. The implementation of the control algorithms represents only around 20-30% of the total application [60]. Some methods have been proposed to simplify such tasks, but they have not yet been proven on real problems.

As already mentioned, in developing such products and systems, the testing and not the design is usually the more expensive, time-consuming and difficult activity. Therefore, not only a tool for helping in the design and implementation is important but also tools which simplify the implementation of testing environments. Nevertheless, this is independent from the fact that the design/implementation process should be done as carefully as possible in order to avoid errors. Unfortunately, it is practically impossible to design/implement an error free system. If this would be possible, testing would be irrelevant.

### 2.3 A Survey of Design Tools for Embedded Control Systems

To understand the relevance of HIL simulation in the ECS development process we give a short overview of all available approaches. In the past the ECS software was written by engineers who understood the application very well but had little training in computer technology. This led to high-quality control systems but with poor software. Which made the systems very difficult to upgrade and maintain. To resolve this problem software tools have been developed in the past years trying to close the gap between controller design and embedded system development.

The tools can be classified in:

- Real-time operating systems
- Programming languages
- Middleware frameworks and design patterns
- Automatic code generation
- Software analysis
- Verification tools
- Testing tools
- Real-time supervisors, redundant and backup systems

In the next few sections we briefly summarize and discuss these tools.
2.3.1 Real-time Operating Systems

Real-time operating systems – RTOSs – differ from general operating system in the following features. The computation is temporally bounded, i.e. the correct computation depends both on the results and on the time when the results are available [9]. RTOSs define computational entities, usually called tasks, bound to real-time constraints, i.e. a periodicity and a deadline. In contrast to the general-purpose OS market, there are many commercially available RTOSs, and no real market leader is present – Wind River with its VxWorks RTOS is the biggest RTOS vendor holding only around 30% of the market. The main reason for this situation is that RTOSs are tidily coupled with the application and the computational platform, which makes it difficult to define and implement a generalized RTOS. Many efforts in standardizing RTOSs have been done, for example the Real Time Java Specification, and the uTRON standards.

2.3.2 Programming Languages

To simplify the development of embedded systems, different programming languages tailored for control systems have been proposed. Unfortunately many ECSs are implemented using assembler or low-level programming languages which do not encourage methodological and clean implementation.

"The purpose of a language, in general, is simply that it must convey meaning" (Confucius). In programming languages semantics, i.e. the meaning of the language, can be very machine dependent or of higher abstraction. The higher the abstraction is, the more we lose in efficiency (the penalty for abstraction). This is the reason why so many ECSs are still implemented with low-level programming languages, because in such systems it is not tolerable to sacrifice the efficiency factor.

The programming languages that we can call real-time friendly are languages which ensure the use of high-level constructs like information hiding [57] and strong typing, but which at the same time are efficient and resource-aware. To be real-time friendly is, however, not sufficient to ensure a good-quality ECS software.

There are many languages which can be qualified as real-time friendly. However, the decision which language to use for an ECS project is often subjective. This, because engineers are quite reluctant to learn new languages and paradigms. We refer to the statement made by Edward Lee. He asserts that most engineers resist new languages based on subjective motivations like: "Can be tough to learn new tricks", or "The way they did the last project was just fine".
2.3 A Survey of Design Tools for Embedded Control Systems

Procedural Languages

Procedural languages are the de-facto standard programming language paradigm. The imperative description of the functional execution sequences is the most widely used programming approach for embedded systems.

The main reason is that the code generated is directly executed on the target machine without any intermediate translation step. The programming language is a close abstraction of the machine and therefore the translation is straightforward. Moreover, the use of such languages makes it possible to do a worst-case execution analysis, which gives an a priori estimation of the execution time needed on the target platform. This is important in hard real-time systems.

The factor time is only indirectly coupled to the language constructs, all instructions describe an operation and are not bound to any time constraint. Writing a program is describing a sequence of instructions, the execution time is therefore given indirectly by the program itself. Therefore we need tools which are able to extrapolate from a program the execution time and this becomes even harder if we use caches, interrupts and so on.

Functional and Logical Languages

Functional and logical languages, like PROLOG and LISP describe the program not imperatively but using logical or functional constructs. Since they abstract the underlying hardware even more than imperative languages, they are in conflict with the goal of an efficient implementation. Consequently they are not well adapted for real-time systems. To improve efficiency and overcome this fallacy some languages have added some sort of imperative features. Nevertheless, in these languages the notion of time is even more decoupled. It is therefore difficult, or even impossible to make a worst-case analysis. These types of languages have rarely been used for embedded control system implementation.

Synchronous Languages

Synchronous languages differ from other programming languages for their model of computation. More precisely, a synchronous program is deterministic both from the functional and from the time point of view. These programming languages are often compiled/translated to state machine representations. Examples of such language are ESTEREL, LUSTRE [31] and Giotto [35].

The advantage of synchronous languages is that the time is tidily coupled in the program description. Moreover, they often include some sort of formal verification – functional but also based on timing. Traditional methods for proving correctness of programs were concerned with
their functional aspects only. Already in 1997 Wirth [75, 51] pointed out the need to distinguish between this kind of correctness and the satisfaction of timing requirements.

On the other hand the state machines have to be implemented somehow on the target system requiring another implementation step. This can lead to errors that would invalidate the timing assumptions and specifications made in the synchronous language.

2.3.3 Middleware Frameworks and Design Patterns

To simplify the design of ECSs some research projects started adopting a framework methodology [60]. The concept is simple: a predefined set of functions and structures is available and must be completed in order to implement the fully functional system. The framework approach has been successfully applied to many non-real-time application areas. We also used it for our HIL framework. In the real-time domain this approach has not yet been widely applied. The main reason for this is that such systems are usually not resource-aware and consequently not economical. Examples of projects using such an approach are the AOCS framework for satellite control [59] and the helicopter control system developed at the Georgia Institute of Technology which is based on the OCP framework — a CORBA based middleware [74].

To reduce the resource-aware problem a more radical design patterns [26] approach has also been tested, making the final application more resource-aware and application tailored. The design pattern approach — an architectural design hint for object-oriented software systems — has successfully been applied to many real-time systems [19]. Many real-time systems do not need a full object-oriented system and therefore the additional complexity can be spared.

2.3.4 Automatic Code Generation

Automatic code generation is a more radical approach to close the gap between controller design and implementation. These tools usually embedded in control design software tools — like Matlab — are able to generate source/object code for different target control platforms. This technique is already mature but only used for rapid prototypes. The generated code is rather difficult to handle: it is difficult to optimize for a target platform and it is difficult to add non-control software to the embedded control system.

Moreover, the code generation only works for a fixed number of platforms and extension cards, bounding the ECS development to standardized hardware platforms which are often not tailored to the application needs.

A more radical approach is the fully automatic generation of both software and hardware, generating a fully tailored design for the ECS. This approach is called automatic hardware/software
2.3 A Survey of Design Tools for Embedded Control Systems

2.3.5 Software Analysis

During the implementation process it is important to be able to analyze the software. Different software analyzer tools are available. They are able to extract useful information like data-flow, worst-case execution times from the source/object software code or do code optimization and assertion verification. Worst-case execution is a very important information needed by the real-time operating systems, since single tasks are scheduled based on the worst-case computations. The worst-case computation is not always computable and is very platform dependent – processor type, RAM size and types, caches.

2.3.6 Verification

A relevant aspect which is gaining importance in the real-time community is the ability to assure the correctness of the designed and implemented system. The verification and testing are therefore vital aspects to prove the correctness quality of the system.

Verification techniques allow us to fully verify a system implementation against a given specification. Obviously, should the specification be wrong, the verification will prove that the implementation is wrong too. Even worse, a wrong specification would prove that an actually correct implementation is wrong.

There are two major verification methods, one is based on theorem provers, the other is based on model checkers.

Theorem provers verify mathematically the correctness of a given program, or by constructing programs with inherent proof. To do this, Hoare’s notation [37] is used to specify what the correct computation is. Such provers are very complex and are not really able to check complex system implementation.

Model checkers [15], however, check that the implementation, usually a simplified model of the implementation called Kripke structure, is a refinement of the specification. This is done intersecting the set of all possible executions of the implementation with the set of all impossible executions of the specification. If the intersection is empty the implementation is a real refinement of the specification, otherwise it includes an erroneous execution. Obviously, the main problem in such techniques is the handling of the combinatorial state explosion. This is solved by either symbolically representing the states using binary decision diagrams – BDD –, or by reducing the set of states using partial order reduction.
2.3.7 Testing

If we are not able to fully check and verify the system, which may often be the case, we have no other choice than to test the system.

Testing occurs when the system is (partially or fully) ready for its deployment, but instead of the real deployment, test sequences are fed to the system input in order to prove the correctness of the ECS reaction. Often, critical or rare situations are used as test cases, i.e. boundary test cases.

2.3.8 Real-time Supervisors, Redundant and Backup Systems

During the ECS operation the system needs to be monitored/tested. To do this we could replicate the system, i.e. make the system fully redundant or replicate some critical components [41]. The replication however is not a guarantee for fault-tolerance since a decision still has to be taken on which device is working properly and which is not. Usually a majority algorithm is used, but still this is not a guarantee.

Another approach is to have an external, independent entity, called a supervisor, which is able to control and coordinate the redundant systems. The supervisor coordination and monitoring is more complex than a simple majority algorithm, but is often simpler than the redundant ECSs.

Backup systems are simplified redundant systems. If the main ECS fails, for example when a supervisor notices a not allowable degradation, control is given to the backup system. The replacement however is only partial and temporary, since the backup system does not provide the full system functionality.

2.4 Case Study: The OLGA Embedded Control System

As mentioned in the introduction, during the helicopter project a dedicated ECS was developed [12]. In this section we will describe the ECS architecture and some implementation details, referring indirectly to the concepts described in the previous sections. The descriptions are an excerpt of the more detailed publications [64, 12, 78, 43].
2.4 Case Study: The OLGA Embedded Control System

2.4.1 OLGA Computer System

We have built a new computer system, tailored for our autonomously flying helicopter. The advantages of such a radical approach are manifold. It permitted a drastic reduction of the power consumption of the computer system, which is a very important aspect since it is powered by batteries. Furthermore, it allowed the footprint of the computer board to be adapted to the needs of the helicopter system (size: 23.2 cm by 16.0 cm). Last but not least, an important effort was put in re-configurable hardware. Research at the Institute for Computer Systems in the last few years has led to programming tools for such devices. By using them we proved that those tools are suitable for practical applications.

It would be unfair not to mention the risks and disadvantages of such an approach. First, the development time is a big issue, since we needed about six months to build a fully functional board. Second, the likelihood of producing errors is bigger, and should there have been errors found on the computer system, we could not have blamed the manufacturer.

The StrongARM SA-110 was chosen as processor, primarily because of its optimal performance vs. power consumption ratio. It delivers more than 100 MIPS at only 1 W. Also, it features an appealing regular architecture with a modern RISC structure. For the implementation of the interface to the various peripheral devices, a programmable gate array was chosen. It allows a high degree of flexibility for further extensions and modifications without any change of physical parts. Although serial data communication can be realized with this resource, an additional UART was included for this prototype. The memory consists of 8 MByte of RAM and 1 MByte of ROM. The ROM can be electrically programmed from the processor (Flash-ROM). Figure 2.2 shows a photograph of the board.

Figure 2.2: OLGA computer board
The extensive use of re-configurable (re-programmable) hardware has proven to be an outstanding technology in our project. The PLD technology was used for the memory control unit. This unit is responsible for the interaction of the processor with the memory and the I/O devices. It is clear that such a unit is central for the operation and overall performance of the computer system.

The field programmable gate array (FPGA) was chosen for handling all the digital I/O of the system. The only exception is the UART, i.e. the serial interface, mainly to simplify the development and bootstrapping process of the prototype. All the other digital interfaces to the helicopter system go through the gate array. That way we are neither bound to specific I/O devices nor to specific I/O signals. Moreover, the use of re-programmable hardware has demonstrated to be a cost effective solution for our prototype system since the modifications of the hardware architecture were made by changing the gate array configuration.

The re-programmable hardware circuits were developed with the CAD tools built at the Institute for Computer Systems. The description of all the digital circuits of the PLDs and FPGAs was done in the language Lola developed by N. Wirth [76]. The packages Hades [49] and Trianus [28] were then used for the routing, drawing and configuring of the gate array device. These tools have proven to be convenient for the correct development and fast implementation of digital circuits.

Figure 2.3 shows the actual circuits implemented in the FPGA. The realized circuits are pulse-width modulated signals (generation and decoding) and a rotor frequency counter. Obviously only a small portion of the FPGA is used, giving us still room for future extensions. The complete implementation on the FPGA takes 699 cells, which is about 17% of the available space.
2.4.2 OLGA Software

The software running on the OLGA computer is subdivided into two categories. The real-time operating system [64] (HelyOS) which controls and manages all the hardware and software resources of the system, and the autopilot software which implements the navigation and control algorithms [43]. The software is written using the OberonSA cross-compiler running on the development station.

OberonSA Cross-Compiler

The compiler was written by N. Wirth, and is tailored to our specific needs. This mainly because it is well known that embedded real-time applications must be based not only on effective hardware, but equally so on efficient software implementation. Therefore, the compiler must give the programmer a closer control over the program. The programmer must be fully aware of the code generation strategy; the compiling algorithm must be transparent. This can only be achieved by a reasonably simple language with clearly defined constructs whose representation in terms of the processor architecture is well understood. The language furthermore adds some features that enable the engineer to better exploit the resources of the underlying hardware. These new features are of course a trade-off between high-level abstraction and low-level implementation, but in terms of advantages and performance this approach is acceptable. The added features worth being mentioned are:

- **Riders** are a means to implement a fast and clear way to access arrays. In the usual implementation of the Oberon language the sequential access to array variables is slow, since each time the array is accessed the offset in the array structure has to be recomputed. With Riders the programmer can easily access (read or write) sequentially an array without recalculating the complete offset in the array structure anew, thus making the array access faster. The Riders concept directly mirrors the increment and decrement addressing modes of the SA processor.

```plaintext
(* without rider *)
VAR a: ARRAY 32 OF INTEGER; i, j: INTEGER;
BEGIN  j := 0;
   FOR i := 0 TO 31 DO
      j := j + a[i] (* Array access = Mem[a + i*4] *)
   END

(* with rider *)
VAR a: ARRAY 32 OF INTEGER; i, j: INTEGER; r : RIDER;
BEGIN  j := 0; SET(r, 0, a); (* r := ADR(a) *)
   FOR i := 0 TO 31 DO
      (* j = MEM[r]; r := r + 4 *)
      j := j + r;
```
Fast interrupt handlers are another important feature of the compiler. Interrupt handlers are parameterless procedures called when an external interrupt occurs. Since in a real-time environment a quick reaction on external events is very important, the compiler must give the programmer the ability to implement handlers with minimal time overhead.

Leaf procedures are procedures that do not call other procedures. Such procedures can be optimized eliminating some overhead in the calling convention mechanism. This allows the programmer to speed up code fractions that are often called.

Register variables are used to further speed up the code within leaf procedures. These variables are to be explicitly declared as register variables by the programmer. They are stored in the processor registers instead of in memory. The computation is accelerated since unnecessary memory accesses are avoided.

As mentioned above the language OberonSA is simple which leads to a compact implementation of the compiler. Compared to compilers for more complex languages, the OberonSA Compiler we use is much smaller (approx. 80 Kbytes). Due to its compact implementation the likelihood of error is small, and so is the compiling time needed for scanning, parsing, and code generation. This contradicts the current trend towards fancy, complex and bulky software packages. Moreover, the principles of separate compilation and load time linking even further reduce the time needed to implement modifications.

2.4.3 Efficiency

One of the bottlenecks in our computer system is the lack of a floating-point unit (FPU) in the StrongARM processor. While this lack seems to contradict our choice of StrongARM as the processor for the on-board computer, the reason is that StrongARM was still in development phase. A new version with FPU was planned to replace the StrongARM processor we used (SA-110), but this is still not available.

In the meantime we chose to implement a software emulator. It had to be completely integrated into the software system, and its execution transparent to the application. The floating-point (FP) instruction set we chose had to be the same as the existing ARM floating-point instruction set (IEEE 754 floating-point format).

The execution time of the emulated floating-point instructions must be reduced and optimized as much as possible. Therefore we adopted some well-known techniques to speed up the FP execution. One of them is the look-up optimization. Since any floating-point instruction is usually followed by more FP instructions, the emulator can reduce the overhead of the emulator's
context switch if, instead of returning to the running program, it directly tries to decode the following instructions. Such an emulator is slightly more complex (15 assembler instructions) but 26% faster than the normal one without the look-ahead feature. The compiled autopilot program consists of 17956 bytes with 2726 FP instructions of which 1395 are adjacent FP instructions, i.e. FP that are following another FP instruction. The performance achieved with the current look-ahead solution is about 0.5 MFLOPS.

We also implemented a second type of emulator where the FP instructions were not emulated, but rather implemented as procedure calls, thus reducing the context switch and the execution time drastically. This method is about three times faster than the emulation. Its performance is 1.5 MFLOPS.

**Real-time Aspects**

The real-time operating system HelyOS [64] controls all the real-time aspects of the system. In general real-time operating systems, the scheduling of the different system tasks has to be versatile, robust, and must run transparently with as little overhead as possible. In our case, however, we have an undeniable advantage: we know the types of jobs needed by the autopilot application. Therefore, we can focus our strategy on this application and optimize it accordingly. We are convinced that the implementation of a simple and clear strategy has the benefit of a faster, smaller and more robust implementation.

The strategies usually adopted are the use of co-routines or threads as multitasking entity. Instead, our approach uses subroutines as the multitasking entity [77]. The scheduler starts the subroutine tasks in a fixed and predefined order, according to their priority. Any task may be preempted by other tasks, i.e. suspended, but unlike in other approaches, it always runs to completion. The HelyOS real-time-aspects are described in detail in [64].

This simple approach reduces the code size of the operating system and the scheduler overhead, thus providing more computational power for the real-time tasks.

### 2.4.4 MICOLA

To program the helicopter mission a so called “mission control language” has been developed. The description of the language can be found in [65].

The mission control language (MICOLA) gives high flexibility for mission descriptions. The language makes a higher abstraction of the underlying hardware and software, allowing us to easily adapt the helicopter to different missions. The language allows even an inexperienced user to implement new missions, without any deeper knowledge of the system.
An example of MICOLA is shown below:

```plaintext
PLAN Demo;

EVENT Hight (z > 10.0);
BEGIN (* detected to high condition *)
  BROADCAST("ToHigh")
END Hight;

PROCEDURE TakeOff; (*TakeOff Procedure*)
BEGIN
  ACCELERATE(1.0, 0.0, 0.0, 0.0, -0.5);
  TRAVEL(9.0, 0.0, 0.0, 0.0, -0.5);
  ACCELERATE(1.0, 0.0, 0.0, 0.0, 0.5);
END TakeOff;

PROCEDURE Hovering(time: REAL); (*Hovering Procedure*)
BEGIN
  TRAVEL(time, 0.0, 0.0, 0.0, 0.0);
END Hovering;

PROCEDURE Landing; (*Landing Procedure*)
BEGIN
  ACCELERATE(1.0, 0.0, 0.0, 0.0, 0.3);
  TRAVEL(20.0, 0.0, 0.0, 0.0, 0.3)
END Landing;

BEGIN EVENTS(TRUE);
  TakeOff;
  Hovering(10.000000);
  Landing
END Demo.
```

The user can produce MICOLA code automatically via a graphical framework developed in Matlab, simplifying the task to plan/generate trajectories. The compiled code can be simulated off-line in order to check the planned trajectory.

### 2.4.5 Commercial Application

A commercially available system, called `wePilot1000` has been implemented based on the OLGA system. A new computer board has been manufactured by the ETH spin-off company `weControl`, based on the OLGA computer system. The ECS software structure and the control algorithms are similar. In Fig. 2.4 the `wePilot1000` system is shown.
Figure 2.4: wePilot1000
Seite Leer / Blank leaf
Part II

Embedded System Testing
– The Framework
Chapter 3

Hardware-in-the-loop Simulators

The best definition of an HIL simulator is given by Isermann [39]:

- The simulated process can be operated with the real control hardware.
- The simulated process replaces either fully or partially the controlled process consisting of actuators, physical process and sensors.

3.1 Architectures

HIL can be implemented in many different ways. The implementation strategy is usually a project and resource dependent matter. To discuss this we use the categorization given by Isermann in [39]. As can be seen from Table 3.1 not all the combinations are feasible, more precisely configurations 3, 7 and 8 are impossible, since a simulated actuator which delivers a physical output is a real actuator of the system, and similarly a simulated sensor which reads physical inputs is a real sensor. Figure 3.1 shows a graphical representation of the various configurations.

Configuration 4 is the real working system, since no simulation is involved, whereas 5 is a fully simulated system. Such a configuration is called Full Simulation. When a real component is in the simulation loop it is called Partial Simulation.

Note that during the development cycle we are able to move from configuration 5 to 4 replacing real actuators, real sensors and eventually the real process. These transitions however are very difficult to implement since at each step a new physical interface has to be implemented between simulation and real actuator/sensor.
In the following section we will refer to Table 3.1 again, and to simplify the numbering we will use the following coding: (actuator, process, sensor) were actuator, process and sensors can be R for Real, or S for Simulated. For example configuration 6 would be coded as (SSR). The parentheses specify that the interface with the system is hardware – analog or digital –, we write [SSS] when the ECS is communicating via software only, i.e. no hardware communication is involved. Obviously [R..] and [..R] are not possible, since to communicate with R we need some sort of hardware interface.

### Table 3.1: HIL categories

<table>
<thead>
<tr>
<th>Case</th>
<th>Actuators</th>
<th>Process</th>
<th>Sensors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Real</td>
<td>Sim.</td>
<td>Real</td>
</tr>
<tr>
<td>1</td>
<td>√</td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>2</td>
<td>√</td>
<td></td>
<td>√</td>
</tr>
<tr>
<td>3</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>√</td>
<td>√</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Impossible</td>
</tr>
</tbody>
</table>

![Figure 3.1: Architectures](image)

#### 3.2 Frameworks

Frameworks are a "set of constraints on components, on their interaction and on the benefits that derive from those constraints" [47]. This broader definition of a framework is better suited than the usual definition used for object-oriented software design [40] to describe our framework.
3.3 Frameworks for HIL

structure. To understand better the characteristics of a framework we orthogonalize its services, as defined by Lee in [47]:

**Ontology** The definition of the meaning of a framework component. For example a component could be a subroutine, a process, or a complex object.

**Epistemology** The knowledge that each component has about itself, about the other components of the framework and about how they communicate with each other.

**Protocols** This is the interaction mechanism of the components. How, when and in which sequence they share or transmit information.

**Lexicon** This is the vocabulary of the interactions. What can be sent, and the meaning of the messages.

These four aspects of a framework allow us to describe and compare frameworks precisely. The advantages in using a framework, besides its architectural reuse, is that adapting the framework to an application specific problem implies a reduction in source code size written by the programmer. The reduction can be up to 90% [24] if compared to the same software written with conventional libraries.

The drawback is that frameworks are hard to develop since it is difficult to find the trade-off between generality and application specific needs. This requires a thorough domain and application software understanding.

Last but not least it takes considerable effort to learn the framework, but once the framework is fully understood the programmer will be able to implement applications based on the framework much faster and more reliably. Examples of software frameworks and studies about their impact in software development are the ET++ [73] for Editors, and the Trianus framework for FPGA CAD Tools [28].

### 3.3 Frameworks for HIL

Frameworks are well suited for domains where numerous similar applications are built from scratch over and over again. The HIL simulation domain is a good candidate, for each new ECS an HIL simulation platform has to be implemented.

Despite having been a widely proven and applied concept in many application domains, frameworks for HIL simulation have been avoided in the HIL simulation community. This since the people who designed HIL simulators were often the same people who also developed the ECS.
Moreover, the implementation of an HIL from scratch was often not feasible or more complex than the ECS itself.

To develop a framework for HIL simulations the following steps have to be made. We have to choose an HIL architecture which suits our scope best, thereafter we have to identify the tasks that the HIL simulation has to implement and extract the common parts, called hot-spots by Pree in [24, 63].

3.3.1 Chosen HIL Hardware Architecture

We chose to implement the Full Simulation configuration since it has the major advantage that the interconnection between the ECS is via analog/digital interfaces. To allow Partial Simulation the implementation of some kind of physical interface is necessary. This is obviously very difficult and expensive to build.

3.3.2 HIL Simulation Framework Requirements or Finding the Hot-Spots

Which is the minimal common requirement that all HIL simulators have to implement? And is it possible to implement them as framework components? We analyzed different HIL simulators and came to the conclusion that the minimal set of functions that are present in all the HILSs are: Numerical simulation, hardware interface, and real-time constraints. These properties have to be implemented in a generalized way in the framework. Furthermore, we added a fourth function, which allows a generalized way of testing the completed ECS.

The framework is subdivided into 4 independent functionalities – we borrow the term from the AOCS framework [58] or sub-frameworks. These simplify the task of using them since their are decoupled – see Fig. 3.2.

The functionalities:

- Fault Generation, Chapter 4.
- Real-time Scheduler, Chapter 5.
- Hardware Interface, Chapter 6.
- Numerical Simulation, Chapter 7.
3.3.3 Building an Application Specific HIL Simulation

To implement an application specific HIL simulator the programmer can use the four independent functionalities, coding the missing parts and re-using the framework code and structure. The drawback of this strategy is that the structure of the customization is limited in its degrees of freedom by the framework.

Additionally, it is possible to use the functionalities separately in case one of them is not needed. For example for non-real-time simulations no real-time scheduler is needed or if no external communication is required then no Hardware Interface generation is needed. In Fig. 3.3 a representation of two different customized HIL simulations is shown, where only a part of the framework is being used.

3.4 Research and Commercial HIL Simulators

In this section we briefly present some dedicated HIL simulators used for research as well as for commercial purposes. We do not restrict HIL in respect to their architectures – as defined in 3.1.
3.4.1 Motion Simulator

This HIL Simulator system developed by Acutronic AG (Switzerland) is a five-axis Flight-Target Motion System mainly used for missile system integration. The configuration of the HIL is of the type (SSR) – see 3.1 for the definition.

3.4.2 TVE

The TVE – Test and Verification Equipment for Spacecraft Attitude & Orbit Control (AOC)– was developed by the NLR, the Netherlands’ National Aerospace Laboratories. It is based on a prototype TVE developed for the European Space Agency (ESA), test equipment has been
3.4 Research and Commercial HIL Simulators

developed for AOCS subsystems and system level testing of the XMM (X-ray multi mirror) and INTEGRAL (International Gamma Ray Laboratory) scientific satellites.

The structure of the system is (SSS), similarly to our framework, the main difference is that the simulator is tailored to the ECS for the satellite attitude controller, called AOCS subsystem. Besides, the simulator and the AOCS are connected, not via the actuator and sensor interfaces, but directly to the VME bus used in the AOCS, the actuators and sensors. The simulator therefore implements the bus protocol for the actuator and sensor, and is limited to systems deploying the same bus system.

3.4.3 HIL Simulators based on Matlab/Simulink and DSPACE

Many ad-hoc HIL simulators developed nowadays are based on the Matlab/Simulink or DSPACE systems. These systems are mature and widely accepted in the engineering community as the de-facto standard tools in control design and simulation. These tools are used for implementing rapid prototypes, but sometimes also to implement HIL simulators. These tools compared with our HIL framework lack two features which are in our opinion central in the HIL simulation implementation.

First, the interface with the ECS is not flexible enough. Such tools do not allow the implementation of dedicated circuits which are able to decode/generate custom signals. This is not possible because it is difficult to implement the hardware, like a PCI card, and to write drivers. Furthermore, they are limited in the set of I/O supported. Although the number of supported cards is massive, they are not flexible enough to adapt themselves to the requirements of specific ECSs.
Second, such tools do not embed in their simulation core a methodology for analyzing the simulation results. Every simulation run is visualized and then analyzed manually. Testing of faults is also done manually. Therefore, error reproductions cannot be formalized nor can the ECS reaction be automatically checked.

They are however, appealing to the control community for their clear and simple user interface (Simulink), but also for the very large amount of available computational libraries, so called toolboxes.

### 3.5 HIL in Education

HIL is not only a useful tool for testing ECSs but can also be used as valuable help in teaching control theory to students. The benefits of HIL are that students can try controllers on many different real control problems without large investment costs, since the plant is simulated. The controller and system are equivalent in complexity to the real controller, and it is possible to implement the control system for hazardous and expensive experiments. This would be impossible to do on a real platform.
In this Chapter we focus on methods for testing embedded control systems. We observe that in many catastrophic failures of real systems, e.g. the Ariane 5 rocket explosion, the Mars Polar Lander etc., the cause of failure was a simple fault or a constellation of simple faults which the system was not able to handle correctly. These faults then propagated through the system bringing it in an unforeseen and faulty condition.

The objective of the sub-framework described in this Chapter is to provide a set of building blocks which simplify the development of a dedicated simulator. This sub-framework implements a method for generating faults, for scheduling their generation and for the automatic verification of a correct system response to the fault. Such tests can be run during a simulation and also used as documentation that the system was correctly implemented for the specified test cases. This also allows us to do regression tests on the system, simulating for every new version of the system the old fault test cases in order to verify that the new system is a real extension and improvement of the previous one, neither invalidating nor introducing new errors on the system.

The first part of this Chapter will explain what faults are and how we can reproduce faults on the simulator. The second part illustrates how a correct answer of the system is specified and checked at run-time. In the third section we introduce the meta language called FAUSEL, which allows us to define in the framework how faults are generated and checked. In the fourth section we compare the testing approach to a formal verification approach. The last section compares this testing approach with other similar approaches.
4.1 Failures, Errors, and Faults

In this section we give a short overview of the basic concepts of failures, errors and faults [41].

![Fault, error and failure diagram](image)

**Figure 4.1: Fault, error and failure**

4.1.1 Failures

A failure is the visible and tangible effect of an error or fault. As defined in [41] and in [5] failures can be classified according to their nature, their perception, their effect and their frequency of occurrence. A correctly implemented ECS should not produce any failures, in the sense of deviating from its specification. The HIL simulator goal is to inject faults into the system and to detect if the response of the ECS to defined faults, and faults configuration is correct, i.e. did not lead to a failure.

**Failure Nature:** This is how the failure is presented to the outside system, it can be either a value failure, i.e. a wrong value is presented to the outside, or a timing error, i.e. the correct value is not presented to the outside after a specified time interval.

**Failure Perception:** This is how the user sees the failure. The perception can be either fail-silent, i.e. the perception of the error is not trivial, or a crash failure, i.e. the system stops working, or a two-face failure (also called Byzantine failure) if the error perception changes. Interesting to note are the theoretical results by [61] which found that to tolerate k failures of a certain type, we need:

- k+1 components for fail-silent failures
- 2k+1 components for crash failures
- 3k+1 components for Byzantine failures

**Failure Effect:** This is the real problematic characteristic of a failure. The effect can be either benign, i.e. the effect has a non-dramatic effect on the overall system, or malign, i.e. the failure deteriorates the system significantly.
4.1 Failures, Errors, and Faults

**Occurrence** This is how a failure repeats in time. It can appear only once, also called *single* failure, or it can be periodical, also called *transient* or *intermittent*. If a single failure is persistent it is called a *permanent* failure.

We note that the simulator will only be able to perceive the failure based on the observation of the inputs to the simulator, i.e. the actuator values and on the state of the simulated system. It is not able to analyze internal states of the ECS.

4.1.2 Errors

These are incorrect internal states. Errors can also have different forms, i.e. be transient, single or permanent. In our case we are not interested in these kind of errors since the HIL simulator can neither control, nor observe them. Errors will generate failures and are therefore only indirectly observable by the simulator.

4.1.3 Faults

Faults are the cause of failures and errors. Kopetz [41] classifies faults similarly to the failures:

**Fault Nature**: A fault may happen by *chance*, if the cause of the fault is a random event, or it can be *intentional* if the fault is generated intentionally by somebody or by something, like a software virus or software error.

**Fault Perception**: This is how the fault is generated, it can be either *physical*, like a cable brake, or can be *design*, i.e. when the fault is in the system specification itself.

**Fault Boundary**: This describes were the fault originated, i.e. were the fault came from. It could be *external* if the fault originated outside of the system, or *internal* if generated inside the system.

**Fault Origin**: Faults can be distinguished by their generation by an incorrect system development or during system operation.

**Fault Persistence**: Faults can occur only once, also called *single* faults, these faults are the most dangerous since they are difficult to track and therefore like time bombs in the system. The *recurrent* faults, are faults that appear quite often. *Persistent* faults are continuously active. In Fig. 4.2 the three fault types are represented.

A more detailed description on faults can be found in the book by Laprie [45]. For a better understanding of these fault characteristics we use a real example. In Fig. 4.3 we have two real
measurements of the same signal: the left presents faults, the right does not. In this case the fault is recurrent, it is originated internally, and it is of physical nature.

The simulator must be flexible enough to reproduce all these kinds of faults and propagate them to the ECS. Furthermore, the fault generation/perception has to be decoupled from the fault nature and persistence, this in order to allow the test of the same fault generation under different frequencies.
4.1 Failures, Errors, and Faults

4.1.4 Fault Tree Analysis

In this Section we shortly describe a technique used to make a fault-hypothesis. This because even a perfectly designed fault-tolerant system will fail if a fault happens which is not covered by the hypothesis.

Fault tree analysis – FTA – is a technique of reliability and safety analysis generally applied to complex systems. Fault tree analysis provides an objective basis for analyzing system design, common mode failures, demonstrating compliance with safety requirements, and justifying system changes and additions.

Therefore, FTA is a tool for the methodological analysis of faults, whereas the fault generation scheme which will be described in the next Section is a tool which allows us to test the system for the analyzed faults.

The concept of fault tree analysis was developed by Bell Telephone Laboratories as a technique to evaluate the Minuteman Launch Control System in 1961. Later the Boeing Company modified the concept for computer utilization. It is now widely used in many fields, such as the nuclear, chemical, and aviation industries. A fault tree is a graphical and logical representation of the various combinations of possible events occurring in a system that lead to the most undesirable event. This event is obviously called the undesired event because we do not desire its realization. It is the root (top) of the fault tree and the analysis aims to determine all its possible causes. The application of analysis of the top undesired event can include a complete factory, as well as a system or a subsystem. The failures of elementary components are called terminal events of the fault tree.

The analysis process consists of decomposing the considered system into sub-systems, then decomposing each sub-system into its sub-systems and so on until the desired granularity in the description is reached. The possible failures of each component are synthesized as a boolean function of the possible failures of its sub-components. A fault tree is thus a boolean formula that describes the various ways failures of elementary components induce a failure of the whole system. Terminal events are assumed to fail randomly and independently according to certain probability laws.

This technique is a starting point and can help in the specification of the design, but also for the test cases. In our case FTA can help in the implementation of the different tests using FAUSEL.
4.2 Specification of the Correct System Response

Once a fault is generated and is propagated to the output of the simulator, the controller has to react on the fault. This reaction is what we want to check. We can categorize the reaction types as:

**Non observable:** the system continues its normal operation, or detects the errors but we cannot observe any change in the system response of the controller.

**Soft:** the system changes its control strategy in an observable but bump-less way. This means that no abrupt change in the control output is noticeable.

**Hard:** the system changes its control strategy in an observable, abrupt way, like for example opening a parachute, stopping the system, etc.

The goal of the system is to analyze the ECS response to faults, detecting incorrect responses automatically. To allow for the automatic detection we need a specification which is expressive enough in describing, what is an acceptable response and what is not. We will compare different possible specification methods and justify our choice for the simulator framework.

The implementation code of the controller can be used as the reference specification. This is usually called analytical redundancy [13, 38, 56], since two identical systems are working in parallel: the embedded system and the simulator. This approach is not able to discern any errors in the implementation and therefore cannot be used as a reference, i.e. as a specification.

Another specification method is the *time forbidden state*. From the controller design we extract a state machine, called supervisor, which is able to check the controller responses in relation to time. This approach [7, 3] gives a technique for characterizing the controller of the embedded system, however not the complete embedded system. Moreover, it is limited to discrete event systems.

Boolean expressions are also appropriate specifications, they are relatively simple to understand and to specify, but their major advantage are the simple algorithms needed for the run-time check of the system responses. Boolean expressions cannot, however express the temporal characteristics of the system. To add temporal information, the introduction of memory elements is needed, but this would be too cumbersome as a description of a system response.

The straight forward solution is to use *Temporal Logic*, which was introduced by Pnueli [62, 50] specifically as a specification for such systems, i.e. reactive systems. *Temporal Logic* can be defined as an extension to Boolean algebra adding time dependent operators to it. A *Temporal Logic* expression differs from a Boolean expression in the fact that it becomes True or False depending also on the past and/or on the future values of the propositions in the expression.
4.2 Specification of the Correct System Response

4.2.1 Linear Temporal Logic - LTL

There are different kinds of Temporal Logic with different characteristics and expressive power, but we can subdivide them into two categories: branching time logic, i.e. CTL – the C in CTL stands for computational – and linear time logic, i.e. LTL [14, 22, 6]. In the branching time logic, the time is represented as a tree of possible future paths, whereas in the linear time logic, the time is a single possible future path since the goal of the fault generation and specification is to test the response of the system, a single path approach is expressive enough. We like the definition of time given by Edward Lee [47]: “a relentless, measurable march”. LTL gives us a method to describe such a march.

Syntax and Semantic

A formal definition of LTL appears in [29]. For a more detailed understanding of the temporal logic refer to [22]. LTL formulas are Boolean formulas with the addition of temporal operators: X, the next operator, and U, the until operator. More precisely, given a set of propositions \( P \) we can define an LTL formula inductively as follows:

- every member of \( P \) is an LTL formula
- if \( \alpha \) and \( \beta \) are LTL formulas then the following expressions are also LTL formulas:
  - \( \alpha \lor \beta \)
  - \( \alpha \land \beta \)
  - \( \neg \alpha \)
  - \( X \alpha \) (this is called the next operator)
  - \( \alpha U \beta \) (this is called the until operator)

The semantics of the Boolean operators are defined as usual. So we merely have to define the semantics of the next and until operators (i.e. \( X \) and \( U \)). An informal definition is that the next operator asserts that the expression \( \alpha \) has to be TRUE at the next discrete time step, and the until operator asserts that \( \alpha \) will be TRUE at the current time and in all the following times until the operator \( \beta \) will finally become TRUE. More formally defined: we define \( P \) as a finite set of propositions, \( S \) as a final set of states, \( L : S \rightarrow 2^P \) as a function that labels each state with a set of propositions TRUE in that state, and \( \omega = s_0s_1s_2... \) as an infinite sequence of states \( s_i \in S \). Moreover, we write \( \omega_i \) for the sequence starting at \( s_i \).

- \( \omega \models q \) iff \( q \in L(s_0) \)
- \( \omega \models \neg \alpha \) iff \( \omega \not\models \alpha \)
• \( \omega \models \alpha \land \beta \) iff \( \omega \models \alpha \) and \( \omega \models \beta \)

• \( \omega \models \alpha \lor \beta \) iff \( \omega \models \alpha \) or \( \omega \models \beta \)

• \( \omega \models X\alpha \) iff \( \omega_1 \models \alpha \)

• \( \omega \models \alpha U\beta \) iff there is an \( i \geq 0 \) such that \( \omega_i \models \beta \) and for all \( 0 \leq j < i \), \( \omega_j \models \alpha \)

We also introduce some other operators: the *imply* operator \( \alpha \rightarrow \beta \equiv \neg \alpha \lor \beta \), the *repeat* operator \( \alpha R\beta \equiv \neg \neg(\alpha U \neg \beta) \), the *finally* operator \( F\alpha \equiv \text{TRUE} U\alpha \) and the *global* operator \( G\alpha \equiv \text{FALSE} R\alpha \).

### Types of LTL Formulas

With LTL formulas we can easily describe a correct answer of a system. Any LTL formula specifies how the system reaction should be, more precisely any LTL formula checks the system for a specified response characteristic.

For example we can assert that a signal will never be greater than a maximum value: \( G(\text{signal} < \text{MAX}) \). These kind of formulas are called *global assertions*. Another kind of LTL formula are the *liveness assertions*, for example if we specify that sometimes a signal will reach a desired value in the future: \( F(\text{signal} = \text{value}) \). The *causal assertions* specify that if something happens then something else has to happen eventually: \( G(x) \rightarrow F(y) \).

As we can see from the previous examples, time is not explicitly defined in the formulas, to do this we need to use the next operator (i.e. \( X \)) more extensively. For example to describe a *bounded-response requirement*, which states that a signal \( s \) has to be active within a given period, for example 4 time periods: \( G(a \rightarrow (s \lor X(s \lor X(s \lor X(s)))))) \).

### 4.2.2 Metrical Temporal Logic - MTL

Metrical temporal logic (MTL), also called integer time logic [1, 2, 33] is an extension of LTL. The LTL description is extended with explicit time specification, called *constrained temporal operators*. For example for the previous case we would simply write \( F(a \rightarrow (F[4]_i(s))) \). This however does not change the expressiveness power of MTL, since as shown in [33] every MTL formula can be re-written using LTL and vice versa.

Formally: we add to the temporal operator \( F,G,U,R \) an optional *constraint* with the following semantic:

- \( F_{[a,b]}(x) \) with \( a \leq b \) iff \( \omega_i \models x \) for an \( a \leq i \leq b \)
4.2 Specification of the Correct System Response

- \( G_{[a,b]}(x) \) with \( a \leq b \) iff \( \omega_i \models x \) for all \( a \leq i \leq b \)
- \( yU_{[a,b]}x \) with \( a \leq b \) iff there is an \( a \leq i \leq b \) such that \( \omega_i \models x \) and \( \omega_j \models y \) for all \( 0 \leq j < i \)
- \( xR_{[a,b]}y \equiv \neg(\neg xU_{[a,b]}\neg y) \)

4.2.3 Testing

The inherent difficulty to test the validity of an LTL formula, i.e. that an observed sequence of events satisfies the definition from the LTL formula, is deducible by the large number of papers and publications about this topic. The best known approach to prove the validity is to use a transformation of the LTL formula into an automaton, more precisely into a generalized Büchi automaton [16]. This transformation uses a tableau-construction [29] which will shortly be described. This approach can be optimized (see [17]); this is, however, not a topic of this dissertation.

LTL to Büchi Automaton

We start by defining a Büchi automaton.

**Definition 1** A Büchi automaton is a tuple \( A = (\Sigma, S, \rho, s_0, F) \) where

- \( \Sigma \) is an alphabet
- \( S \) is a set of states
- \( \rho : S \times \Sigma \rightarrow 2^S \) is a nondeterministic transition function
- \( s_0 \) is the initial state
- \( F \) is the set of accepting states

This definition is similar to a finite state automaton, with the difference in the definition of the accepted sequences. A run of \( A \) over an infinite word \( \omega = a_0a_1..a_\infty \) is an infinite sequence \( s_0, s_1, ... \) such that \( s_i \in \rho(s_{i-1}, a_i) \) and \( s_0 \) is the initial state. A run is accepted if there is some accepting state, i.e. an \( s \in F \), that is repeated infinitely often. The set of accepting runs of \( A \) is defined as \( L(A) \).

The construction starts with the translation of the LTL formula in the negated normal form. In this form no negation is allowed on the temporal operators. It is only allowed on the propositions. Furthermore, Boolean operators cannot have any temporal operator as an argument. This
transformation is done by applying recursively the De Morgan law for the Boolean expressions and the temporal operation definitions. Moreover, we eliminate all the $F$ and $G$ by using their definition with the $U$ and $R$ operators.

Example: $\neg F(x)$ is transformed to the $\neg (\text{TRUE } U x)$ using the definition of $F$, and then transformed to the normal form $\text{FALSE } R \neg x$ using the definition of $R$.

With the formula in normal form we then construct the Büchi automaton. The algorithm is based on the use of the fixpoint property of the $U$ and $R$ operators. The formula $xUy$ is equivalent to $y \lor (x \land X(xUy))$, and the formula $xRy$ is equivalent to $y \land (x \lor X(xRy))$.

As example we translate the LTL formula $FG(a)$, which states that a proposition $a$ eventually remains true. The formula is transformed to the negation normal form $\text{TRUE } U (\text{FALSE } Ra)$, then transformed into the automaton. Figure 4.4 shows the resulting automaton. Traversing this automaton at each time step we will be able to prove the correctness of the ECS response.

MTL to Alur-Dill Automaton

We extended the algorithm [29] in order to generate an automaton from an MTL description. To do this we need to use an extended variant of the Büchi automaton, the Alur-Dill automaton [2], in some literature this form of automaton is also called Timed Büchi Automaton.

We start by defining an Alur-Dill automaton.
4.2 Specification of the Correct System Response

**Definition 2** An Alur-Dill automaton is a tuple $A = (\Sigma, S, s_0, F, C, E)$ where

- $\Sigma$ is an alphabet
- $S$ is a set of states
- $s_0$ is the initial state
- $F$ is the set of acceptable states
- $C$ is a set of clocks
- $E : S \times S \times \Sigma \times 2^C \times \Phi(C)$ is the set of transitions. An edge $< s, s', a, \lambda, \phi >$ represents a transition from state $s$ to state $s'$ on input symbol $a$. The set $\lambda$ gives the clocks to be reset with this transition, and $\phi$ is a clock constraint over $C$.

The construction is similar to the Büchi automaton, but we modify the fixpoint in the following way. The formula $pU_{[1,h]}q$ is equivalent to:

$$(((\text{time} - u) \in [1..h]) \land q) \lor [p \land X(pU_{[1,h]}q) \land (\text{time} - u) \leq h],$$

where time is the actual system time and $u$ is a timestamp assigned to the operator $U$. The formula $pR_{[1,h]}q$ is equivalent to:

$$((\text{time} - r) > h) \lor (((\text{time} - r) < 1) \lor q) \land [p \lor X(pR_{[1,h]}q)].$$

Remark: The fixpoint definition of $pR_{[1,h]}q$ is computed from the inverted fixpoint definition of $\neg pU_{[1,h]} \neg q$.

In Fig. 4.5 an example of a small Alur-Dill automaton is shown. The automaton specifies a bounded-response time; the response time to an event $a$ has always to be less than 2 time units. This is described in MTL as: $F(a \geq F_{[0,2]}b)$.

**Testing for Inclusion**

Automata of this kind have two properties which make them difficult to test at run-time if a sequence of propositions is included in the language $L$ defined by the automaton.

The first is that the Büchi automata are nondeterministic. In a finite state automaton this can be overcome since the automaton can easily be transformed into an equivalent deterministic automaton, for a proof see [23]. For the automata on infinite words, this is not applicable, for
the proof see [15], and therefore another approach needs to be chosen. To solve this we have no other way than to check all the active states during a run, since we have multiple active paths on the automaton. At the beginning only the first state $s_0$ is checked, but consequently as soon as the checker encounters nondeterministic states the checkers will have to check all the different parallel paths. The maximal number of active paths is the maximal number of states, which is finite and related exponentially to the complexity of the temporal expression.

In Fig. 4.6 we illustrate an inclusion test for two proposition sequences against the automaton generated by $bUa$. As we see in this example, this is a special case since the problem is finite and decidable, e.g. if a sequence arrives at the state $s_1$ it will never leave it and therefore the sequence is accepted by the automaton.

The other problem, already mentioned before, is the undecidability at run time if a sequence belongs to the $L(A)$ set. This because the definition of the automaton states that a sequence must traverse infinitely often the acceptable states. In some special cases acceptable states have an outgoing edge with the proposition true, these states are also called final states, since once we arrive at this state we finished the inclusion test. If the automaton does not have any final state we will have to traverse the automaton infinitely, obviously impossible at run time. We are therefore only able to check if a response is not included in the language at the actual time.

This can be illustrated with the formula $G(a)$, in this case we are not able to check if a sequence belongs to $L$ but we are able do detect if a sequence does not belong to it. See Fig. 4.7.
4.3 FAUSEL

FAUSEL is an acronym of Fault Specification Language and enables the simulator framework to check on-line for some specification. More precisely, the simulator is able to determine if some specified reactions are to be considered acceptable or not. The EBNF syntax description of FAUSEL and the interface definition of Faults are in Appendix C. This sub-framework is
implemented in three modules *Fault*, *LTL* and *FAUSEL*. *Faults* implements the back-end for the fault generation and the fault objects, the *LTL* does the LTL/MTL to automata translation, and *FAUSEL* is the front-end which implements the meta language (i.e. scanner, parser and fault scheduler). The LTL/MTL translation has been described in the previous sections, now we focus more on the back-end and front-end, i.e. how faults are executed, programmed and scheduled.

In *FAUSEL* we are able to describe when and how faults have to be simulated on the HIL simulator, and we are also able to specify what a correct answer coming from the ECS has to be like. The fault itself has to be programmed as an object-oriented extension of the Fault Object – this will be described shortly in the next section. In the *FAUSEL* program we describe the fault nature and persistence, i.e. when it will start, which period it has and when it will stop. The execution of all the specified faults is scheduled as discrete events which will perturb the normal HIL simulation execution. We will explain the scheduling more in detail in the next few sections.

### 4.3.1 Fault Objects

The fault object is the main actor in the fault generation and testing. To implement a specific fault in the dedicated simulator, the fault object has to be implemented overloading the methods of the base object. The main interfaces are dedicated to the following tasks: variable declaration and fault generation. The variable declaration is needed by the tester, since it needs a method to access the variables of the fault and of the simulation. This is done during fault object initialization: the object itself declares the variable which will be visible by the *FAUSEL* language. *FAUSEL* can then access them via the *Enum*, *Get*, and *Set* methods. Every fault object defines at least the variables *firststart*, *start*, and *period*. where *firststart* is the time when the fault was started for the first time, *start* the next fault execution time, and *period* is the interval between two fault executions.

To generate the fault the object must implement the methods *FirstDo*, *Do*, *LastDo*. These methods are called during the execution of the fault. *FirstDo* is called at the first execution, *Do* at every period, *LastDo* at the last fault execution.

### 4.3.2 Compilation

Once the desired faults are implemented, extending the *Fault* object, we are able to describe how a simulation test should be done. This is done by writing a *FAUSEL* description in which we describe when, how long and how often faults have to be activated. Moreover, we have to specify the frequency in which we check the ECS response.
During compilation a data structure is constructed which will be interpreted during execution. In the example which will be given at the end of this section we will show the generated data structure.

4.3.3 Execution

After the FAUSEL compilation the simulation can be started. During simulation the FAUSEL scheduler will generate faults as described by the FAUSEL program and test for the correct response as specified by the LTL/MTL specification.

The execution of the faults is controlled by a scheduler which chooses from the data structure the faults which need to be activated and also triggers the checking of the Büchi automata for each fault specification.

4.3.4 Example

We simulate a very simple system, a spring which has the following motion equation:

$$m\ddot{x}(t) = -kx(t) + u(t),$$

where $x$ is the position of the spring, $u$ is the controller input, $m$ a mass and $k$ the spring constant.

We then assume that an embedded system measures the value of $x(t)$ and generates the force $u(t)$. Now we would like to test the embedded system for the following error conditions:

1. the sensor measuring $x$ sometimes does an automatic re-calibration procedure which generates errors in the read value $x$, the controller has to stop its control till the normal sensor operation is restarted after maximal 10 seconds or it sends an alarm to the operator, and

2. the actuator $u$ sometimes blocks, and can be unblocked only if a backward motion is done. The embedded system controlling the spring must consider these two known error conditions, observe the system and implement a correct strategy to overcome these problems.

With FAUSEL it is possible to check the correct response of the system with the following description:
SIMULATE SPRING;

FAULT A IS Spring.SensorCalibration;
START 10;
STOP firsttime + 10.0; (* stopped after 10 seconds *)
PERIOD 0.5; (* we check the system every 0.5 s *)
SPEC
  G[0,10](u = 0) (* the system stops any active control *)
END A;

FAULT B IS Spring.Block;
START A.firsttime + RAND({); (* started between 0..1 s after A *)
STOP firsttime + 2.0; (* stopped after 2 seconds *)
PERIOD 0.2; (* we check the system every 0.2 s *)
SPEC
  F[0,1](u < 0) (* at least one backward movement *)
END B;

END SPRING.

The Spring.SensorCalibration and Spring.Block extend the Fault base object. Their implementation stops the generation of the sensor values, and stops the generation of position values. The data structure generated by the FAUSEL compiler is shown in Fig. 4.8. In Fig. 4.9 the execution of the faults generation and verification during a simulation run is shown.

This example has been inspired by a real controller platform used for an automatic anaesthesia project [25] where the sensor calibration are called artifacts and the spring stands for a mechanical syringe. The embedded system in this case has much more sophisticated fault diagnose procedures, and in case of a fault the controller computes a mathematical model of the system and continues the control in a blind mode, using the simulated value as the actual position.
4 Fault Generation, Specification and Test

4.4 Specification Testing versus Formal Verification

LTL was introduced as a specification language for reactive systems by Pnueli [62] in 1977. Since then this logic has been extensively used by the verification community as a formal specification. The following Table 4.10 summarizes the differences between formal verification and specification testing.
4.4 Specification Testing versus Formal Verification

<table>
<thead>
<tr>
<th>Formal Verification</th>
<th>Specification Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>static view</td>
<td>dynamic view</td>
</tr>
<tr>
<td>symbolic execution</td>
<td>concrete execution</td>
</tr>
<tr>
<td>complete cover</td>
<td>partial cover</td>
</tr>
</tbody>
</table>

Figure 4.10: Verification vs. testing

The advantages of a test are the following:

- The test is done on the real system, not a model nor a subset implementation.
- It is possible to test the real system in the field, where the system eventually will be deployed.

The main disadvantage is that the testing cannot cover all the execution sequences.

The differences between formal verification methods and testing methods do not make them antithetic. These methods are important in different phases of a project. The formal verification rather at the beginning where a complete model description of the system is still verifiable due to the reduced complexity, and in a final stage the testing ensures that critical parts of the system are checked on the real final platform [4].

4.4.1 Model Checking

We would like to show the differences between formal verification and specification testing concretely using a small example from the model helicopter system to be explained in Chapter 8. Since the HIL simulator is described in the cited Chapter, we will focus on a possible formal verification of the state machine using a Model Checker. The model checker used is the SMV – Symbolic Model Checker [52]. The model we want to check is the part of the helicopter system that controls the interaction between the human pilot and the auto-pilot system as well as the critical take-off phase. The system has three inputs: two switches controlled by the pilot and a ground station which can send the message takeoff to the helicopter. This model is a simplification of the real system, but complex enough to show an example of a formal verification.

The following SMV description specifies the user interface and interaction with the helicopter. At the end of the SMV description the specification parts are defined, using CTL, a variant of Temporal Logic. Two different specifications are verified: the first asserts that if the helicopter motor is idle the helicopter must be on the ground (this is a trivial assertion, but ensures that our implementation models the helicopter correctly). The second one asserts that if the helicopter is
flying the engine will never be controlled manually. Using SMV we can prove that the following automata implementation will never invalidate these two assertions.

The main problem here is that this implementation is not a one-to-one implementation on the real system and therefore, it is not possible to guarantee their equivalence. However, it is possible to prove that the design of a solution of the problem is correct, and it is then a matter of implementing it correctly.

```
MODULE main
VAR
  switch1: off, on;
  switch2: off, on;
  grdmsg: none, takeoff;
  motor: idle, ramp0, ramp0done, ramp1, run;
  helipos: ground, air;
  controller: bypass, engine, full;
ASSIGN
  -- initial inputs
  init (switch1) := off;
  init (switch2) := off;
  init (grdmsg) := none;

  -- initial state of the helicopter
  init (motor) := idle;
  init (controller) := bypass;

  next(motor) := case
    (motor = idle) & (switch1 = on) : ramp0;
    (motor = ramp0) & (switch1 = on) : ramp0done;
    (motor = ramp0done) & (switch1 = on)
      & (grdmsg = takeoff) : ramp1;
    (motor = ramp1) & (switch1 = on) : run;
    1 : motor;
  esac;

  next(controller) := case
    (controller = bypass) & (switch1 = on) : engine;
    (controller = engine) & (switch1 = on) : engine;
    (controller = engine) & (switch2 = on) : full;
    (controller = full) & (switch2 = off) : engine;
    1: controller;
  esac;

  helipos := case
    motor = run : air;
    1: ground;
  esac;
```
4.5 Other Similar Approaches

SPEC AG (motor = idle -> helipos = ground)
SPEC AG (helipos = air -> ((controller = engine)
| (controller = full)))

4.4.2 Hybrid Automata

There is also some research done in the use of Hybrid Automata. These automata are used to describe the system and the controlled environment together. Since the environment is normally a continuous process, the automata are characterized by a discrete and a continuous description. This is why they are called Hybrid Automata. The goal of such systems is to verify formally the complete system, the controller and its interaction with the environment. This type of verification is, however, still affected by the problem that the final system and the description verified could present some difference and therefore also some undiscovered errors.

One method to overcome this pathological problem is to have an automatic translation from the description verified to a full implementation or a skeleton of it. This could reduce the likelihood of an implementation error but cannot guarantee its absence. Moreover, such tools – for example the research projects of Masaccio [34] and Giotto [35] – are still in development.

4.5 Other Similar Approaches

There are applications used for testing a system at run-time which use the same specification. The Time Rover commercial software package [20] uses LTL/MTL specification added as comments in the source code of the real platform. The compiled code will then be added with code sequences which check the LTL/MTL specifications on-line. To reduce the computational overhead the software delegates the checking of the temporal logic assertions to a host system whereas the target system includes only code fragments which send the relevant information needed by the host for the testing. This is a more elaborated way to describe assertion than the assertion statements found in many programming languages.
Seite Leer / Blank leaf
In this Chapter we describe the sub-framework responsible for the real-time execution of the simulation. We start by analyzing the features of available real-time operating systems, and then describe what is required by our HIL framework.

5.1 Real-time Execution

Real-time systems and real-time operating systems (RTOS) often offer a large set of functionalities and are thus very complex. The range of functionality is very wide allowing the implementation of many different real-time application types from the microwave oven to the Mars polar lander system. Therefore, the same system has to furnish a wide set of functions. These functions can be subdivided in: Scheduler, Task Priority Assignment and Resource Management, which are described in the following subsections.

5.1.1 The Scheduler

The scheduler controls the execution of each thread or task. It is responsible for selecting the appropriate tasks, and to start its execution on the processor (or processors). There are many different scheduler strategies suited for real-time applications. However, if the task execution sequence does not change during the real-time execution the scheduling can be done off-line; the order in which the tasks will be executed in real-time is defined statically.
On the other hand, if the set of activated tasks changes at every step, a dynamic scheduler has to be provided. The scheduler decides at run-time which tasks are ready to be executed and in which order. To simplify this decision, for each task a priority is assigned.

### 5.1.2 Task Priority Assignment

The priority of a task can be assigned statically or dynamically. The dynamic assignment is based on the following criteria: the worst-case execution time needed by the task, the deadline time of the task, the start time of the task and the resources needed. There are very different algorithms used for the dynamic priority assignment. The most widely used algorithms are the earliest task first, and earliest deadline first [9]. These algorithms change the task priority at run-time, this means that an off-line analysis of the execution is very difficult, i.e. NP-complete. Moreover, such schedulers suffer from the priority inversion problem.

#### Priority Inversion

When two different tasks share a common resource, they somehow have to guarantee that the resource accesses do not collide. The two tasks will therefore have to observe the critical region of the resource access to guarantee collision-free resource handling.

Since these two tasks can block each other during their critical execution, it is possible that the priority inversion phenomenon happens. This is the case when a task $A$ with high priority is waiting for a resource which is blocked by a task $B$ with a low priority. The task $A$ can therefore be preempted by a third task $C$ (with a medium priority). This is an error: In such a situation a high priority task could be preempted and its execution delayed by a lower priority task. The naive solution is to allow a sort of priority inheritance: task $B$ inherits the highest priority of the blocked task, which will assure that task $A$ will not be preempted by a task with lower priority.

If the scheduler uses static priority assignment, the scheduler becomes very simple, since the chronological execution sequence is decided off-line, allowing an easier off-line execution analysis.

The dynamic case is a must for all systems which do not know the set of tasks executed on the system a priori. This is true for servers which have to perform client specific tasks in real-time. For systems were the set of tasks is known a priori there is no need for complex priority assignment.
5.2 Real-time Scheduler Features for an HIL Simulator

5.1.3 Resource Management

Since different tasks can preempt each other a full featured RTOS has to provide mechanism, like Semaphores [18] and Monitors [36] to protect the access to shared resources like Input/Output devices and memory. These mechanisms are well known but they are also the cause of many failures since they are complex to use and may cause deadlocks.

5.2 Real-time Scheduler Features for an HIL Simulator

We start to identify the features that an HIL simulator needs. The HIL simulator has to reproduce the outside environment as closely as possible. This means that the interaction with the outside is a critical part of the system. The outside is the embedded system which is responsible to control both the simulated and the real system correctly.

These are obviously real-time constraints to the simulator, the inputs have to be read in the rate needed by the simulator and the outputs must be ready to be read by the embedded control system on time.

HIL Priority Assignment

The involved tasks are known a priori, and this implies that the scheduling of the tasks can be easily solved by a simple cyclic executive, i.e. the tasks are started in a fixed given order.

The tasks do not have to do complex computations and can be simple procedure calls, this is the concept introduced and implemented in the Oberon 7 system [77] and also in the HelyOS system [64]. The tasks are execution entities which run to completion. They simplify the scheduler since no complex context switching is necessary.

We can categorize the real-time task on the HIL simulator: signal acquisition, signal generation and model simulation. How these functions are implemented is explained in Section 7. Here we focus on the cyclic execution of the tasks.

Worst-Case Execution

The executed real-time task must have a worst-case execution time less than the minimal period of the task. To ensure this we must be able to compute the worst-case execution of a task. This
computation is goal of many research projects, since the computation of the worst-case execution has to take everything into account, the architecture of the system, the memory, the caches and so on.

In the HIL Framework a more conservative approach can be applied since typically the algorithms involved in real-time are simple in complexity. Consequently, it is simple to estimate their worst-case execution time.

The Framework Scheduler

The tasks are registered in the scheduler during initialization. To be registered a task needs to define its starting time, its stopping time, the up-call procedure and the period of execution. The scheduler frequency is fixed and therefore the period of execution defined by the task will be approximated by the scheduler. In Fig. 5.1 an execution sequence of three tasks is shown. The task may need to have private data which has to be invisible to the other tasks. To do this we can exploit the usual information hiding [57] mechanism inherent in modules to guarantee the invariant of the data, or the task can extend the task object with records which are accessible only by the task itself.

DEFINITION Scheduler;

TYPE
  ActionHandler = PROCEDURE (me: Task);
  Task = POINTER TO TaskDesc;
  TaskDesc = RECORD
    dT-: REAL;
    running-: BOOLEAN;
    time-: REAL;
    timestart-: REAL;
    timestop-: REAL;
    timewarp-: REAL;
  END;

VAR
  dT-: REAL;
  running-: BOOLEAN;
  time-: REAL;
  timestart-: REAL;
  timestop-: REAL;
  timewarp-: REAL;

PROCEDURE GetTime (): REAL;

PROCEDURE Install (t: Task; start, stop, period: REAL; action: ActionHandler);
PROCEDURE Remove (t: Task);
PROCEDURE Run (start, stop, warp: REAL);
PROCEDURE Stop;

END Scheduler.
5.2 Real-time Scheduler Features for an HIL Simulator

Figure 5.1: Task scheduler

- Task1: period = 5ms, start = 1000ms, end = inf
- Task2: period = 10ms, start = 1000ms, end = 1010ms
- Task3: period = 0, start = 1010ms

5.2.1 HIL Resource Access

Since task execution is purely sequential, the only protection needed for the shared resources is between background activity and the tasks. The tasks preempt the normal background system only and therefore a sort of access mechanism has to be performed. Possible implementations are: Mailboxes and the Non Blocking Write Protocol.

In the Native Oberon implementation of the real-time scheduler we have a garbage collector background task which interferes with the scheduler. To protect the scheduler from the garbage collector is rather complex and therefore we chose to disable it during the real-time activity. Mailboxes and the Non Blocking Write Protocol cannot be used to solve this resource conflict since the garbage collector changes all pointer references on the heap during its mark phase (a pointer rotation algorithm is used), and therefore invalidates all the pointers.

Mailboxes

As stated before, the framework does not allow any blocking on any task, which means that the synchronization cannot be based on the mutual exclusion. A mailbox is a unidirectional channel which allows the real-time task and the background to exchange information. The holder of the mailbox can get messages, and is noticed by a flag which states that something is ready to be read from the mailbox.

Non Blocking Write Protocol

Another approach for a lock-free protocol is the non blocking write protocol [42]. In this protocol a resource has one writer and many readers. The writer will never be blocked by the reader,
whereas the reader might be blocked by the writer. The resource is bound to a counter which is
incremented atomically by the writer both at the beginning and at the end of the write access,
i.e. during the write operation the counter will have an odd value. The reader can start to read
the data as soon as the resource counter is even. If at the end of the read operation the reader
reads a different counter value an inconsistent data was read and therefore has to restart the read
operation.

If the time between writes is significantly longer than the duration of a write or read operation, we
are able to compute an upper bound for the number of read attempts. The worst-case execution
time of a typical task with read retries is only a few percent of the worst-case execution of the
task without read retry [42]. To note is the fact, that the opposite, i.e. a Non Blocking Read
Protocol, is not possible.

5.2.2 HIL Frequency

We still have to analyze the update frequency of the HIL simulation, i.e. the frequency at which
the HIL simulator needs to work.

We can identify two cases: one in which the time between the two systems is synchronized, in
this case the frequency can be the same as the update frequency of the controller.

In the second case we assume that no synchronization between the controller and the framework
is available. In this case we have to guarantee that from the controller side it is impossible to
discover any difference between the real and simulated system. This means that if the controller
writes something to the simulator, the next time the controller reads the input, the input must be
already updated by the simulator.

If the controller update has a frequency $f_{controller}$ then from Shannon's theorem we know that
the maximal signal frequency detected by the controller is $f_{system} < \frac{f_{controller}}{2}$. On the other
hand in order to generate/simulate a signal, we need to apply Shannon's theorem backward,
i.e. $f_{simulator} > 2f_{system}$. Therefore, using $f_{simulator} = f_{controller} > 2f_{system}$ we are able to generate
signals that are correctly detected by the controller.

This however does not solve the generated delay problem. This effect can be reduced if a correct
controller frequency is used. The best way to show it is to make an example of an HIL simulation
of a wave with a frequency $f$, a controller and the HIL simulator with frequency $g = 2f$. The
controller and the simulator are not synchronized and therefore they have an offset $k$.

In Fig. 5.2 we see the example, the simulator generates the wave with frequency $f$, but the
controller reads the value generated by the simulator $k$ seconds after, shifting the read wave by $k$.
The same problem arises in the opposite direction, i.e. when the controller writes to the actuator
and the simulator reads the value. The only way to reduce this problem is to have an update rate
which is much bigger than the simulated process frequency, this ensures that the maximum delay will be small relative to the simulated system frequency.

If the two independent clocks of the ECS and of the HIL simulator are not precise, we can also have a clock skew problem. This is when the offset $k$ can vary with the time. Therefore, the read wave will not only be phase shifted but also the frequency of the read wave will be changed.

**Time Warping**

We are able to implement a time warping mechanism. This allows us to speed up or slow down a simulation. In case of external signal generation which is dependent of time, like PWM or simple serial interfaces, care must be taken since the time warping is only valid for the local time of the simulator and controller software and not for the communication signals.

### 5.2.3 Hard Real-time Scheduler Implementation

The hard real-time scheduler for the Native Oberon System is implemented by using the time interrupt handler. This handler, called by the Oberon System every 5ms, calls the scheduler.
The scheduler searches the tasks which have to be activated and starts them sequentially. In this implementation re-entrant tasks are not allowed, because interrupts are disabled during the execution of the real-time tasks. This does not cause any problem if the sum of all task execution times is not more than 5ms. Since no pre-emption is allowed, we also do not have to assign priorities to the different tasks. To permit re-entrant tasks we would need the following steps: a) re-enable the interrupts after having saved all critical data into the local stack of the scheduler, and b) just before the exit disable the interrupts again, restore the critical data and re-enable the interrupts again. The critical data are: the floating point registers, since they are not stored automatically by the interrupt handler, the time variable and the task data structure. For HIL simulation, however, we do not need any priority, we usually only need few tasks. In some cases we need tasks for writing, or reading data with other frequency for actuators and sensors, but no re-entrance is needed.

Scheduler Pseudo Code:

```plaintext
Disable Interrupts;
Save Floating Point Registers;
IF time < timestop THEN
   time := time + timewarpfactor * 0.005;
   FOR all Tasks DO
      IF task.exec_time <= time THEN
         execute task;
         IF (task is periodic) and (task stoptime > time) THEN
            next task execution time = old task execution time + task period;
            reschedule task;
         ELSE
            remove task;
         END;
      END;
   END;
ELSE
   Terminate Simulation;
Re-enable Garbage Collection;
END;
Restored Floating Point;
Re-enable Interrupts;
```

5.2.4 Soft Real-time Scheduler Implementation

The scheduler can be also implemented using a soft real-time mode. This can be the case for very slow processes, like the hydro-powerplant barrage – see Chapter 9 – or for simulations where no real-time is required. The implementation is a simple background task which uses the system time as the clock. If the ECS works at a much lower frequency than the maximal frequency of
the soft real-time scheduler, it can be used without loss of precision. The Barrage simulation in Chapter 9 was done using the soft real-time scheduler since the ECS frequency is 5Hz and the scheduler maximal frequency around 7Hz.

The implementation is done trying to get the maximum of execution possible from the Oberon System. In the implementation for Windows Oberon we were able to achieve a maximum of 100Hz, on a Pentium III 700MHz. This however depends heavily on the complexity of the scheduled tasks, and on the activity of the System. Surprisingly with this approach the jitter between execution was relatively small. Since we cannot guarantee the period between two scheduler executions, we adopted a dynamic delta. This means that the actual time is not updated like before with fixed steps, but adapted to the actual execution of the background task. In the tests done we achieved in the most complex execution cases a maximal frequency of 7.6Hz, i.e. the scheduler was called after maximal 131ms.

Scheduler Pseudo Code:

```plaintext
IF time < timestamp THEN
    tasktime := Time();
    dT := timewarp*(tasktime - lasttasktime);
    lasttasktime := tasktime;
    time := time + dT;
    FOR all Tasks DO
        IF task.executetime <= time THEN
            execute task;
            IF (task is periodic) and (task stoptime > time) THEN
                next task execution time = old task execution time + task period;
                reschedule task;
            ELSE
                remove task;
            END;
        END;
    END;
ELSE
    Terminate Simulation;
END;
```
Seite Leer /
Blank leaf
Chapter 6

Signal Acquisition and Generation

The HIL simulator must be able to communicate with different external actuators and sensors. To achieve this goal different approaches may be used. In this Chapter we will summarize them and illustrate the possibilities and our selection.

6.1 HIL Simulation and Signal Generation-Acquisition

To be able to define a common strategy to communicate with the external system we analyze what kind of actuator and sensor communication systems exist.

Here we do not discuss the analog interface, since we will devote a special Section 6.4 to it. Moreover, we can handle actuators, and sensors similarly, because the only difference is the direction of the information exchange.

**Digital Signal:** In this case the actuator input, or sensor output is an encoded digital signal. For example a PWM (Pulse With Modulation) or a PFM (Pulse Frequency Modulation). To be able to read such signals a dedicated hardware component has to be implemented.

**Point-to-Point Communication:** In this case the communication is based on a standard interface, like RS-232, RS-485 etc. The information is then transmitted in a dedicated protocol.

**Bus System:** In this case various actuators communicate via a shared communication interface, like Ethernet, VME, SpaceWire etc.

In most known HIL systems the interface with the target system has been solved by using a *Bus*
System. In this way only a single hardware interface is needed since all the actuators and sensors can be implemented via software-simulation.

This approach has the disadvantage of making the HIL simulator too application specific. An example is the TEV [72] simulator – discussed in 3.4.2. In such systems the bus used is the VME bus used in satellite systems, making the simulator suitable only for applications where such a bus is used.

Therefore, we focused on a solution which is more general. By integrating re-programmable hardware devices in the HIL simulator framework we achieved this, since there is no theoretical limitation in the FPGA implementations. All signals can be generated and decoded by a dedicated FPGA digital circuit.

Obviously, FPGA is not the only possible medium of communication, the use of other devices is still possible. In short the following requirements had to be solved by this sub-framework:

1. Easy to implement different types of interfaces.
2. Easy to write the software drivers.

In order to achieve these requirements we used the following techniques:

1. Use of the HDL Lola to program the re-programmable HW.
2. Use of Field Programmable Gate Arrays (FPGA) for the digital interface.
3. Use of a DAC/ADC card for the analog interface.

6.2 The Language Lola and its Extension LolaX

The Lola – Logical Language – [21, 76] is a Hardware Description Language (HDL) developed at the ETH by N. Wirth. The language was originally used for teaching digital design at ETH. The language is simple but descriptive enough to implement real hardware problems. The Lola compiler was extended with different tools [79] for PLD design – Mach and GAL22V10 devices – and for FPGAs – Xilinx XC6000 [49, 28] and Atmel 6000. We used Lola also in the helicopter project for the design and implementation of the memory controller on a Mach PLD, and for the signal generation and acquisition for the PWM signals on a Xilinx XC6000 FPGA.

LolaX, i.e. Lola eXtended, extends the original Lola language with the Interface Definition. Using such definition it is possible to translate the generated code into a standard definition file format for Xilinx FPGAs and to write generic device drivers. A simple introduction to the Lola and LolaX languages can be found in Appendix A.
6.2 The Language Lola and its Extension LolaX

6.2.1 The Interface Definition

In order to be able to automatically generate an interface definition, the LolaX compiler needs more information from the Lola definition. First LolaX needs to know which signals are used to communicate with the PC, and second it needs to know where the data is located in the logical address space implemented by the FPGA.

To define signals which communicate with the PC the new keywords DATA, ADDRESS and WRITE are used. To define the logical location of the variables we use the same syntax as the original Lola, i.e. the position assignment. As in the original Lola, signals which are connected to pins are still defined with the position assignment. The exact syntax will be clarified by the example illustrated in Section 6.2.4. We also added the possibility to import other LolaX programs, making it possible to implement digital circuit libraries.

The LolaX compiler translates the code into a data structure which is saved as a file. The storage of the data structure is done using a single pass recursive traversal of the data structure. This technique is explained in [30] and eliminates the need of a second pass. The definition of the symbol file structure is given in Appendix A. The compiled code is translated to an FPGA “program”, i.e. a bitstream, by an external tool, which is not part of LolaX compiler. The generated symbol file is used by a Front-end which is the interface for the software driver. The Back-end handles the communication and configuration of the device itself. This subdivision allows us to change easily the FPGA hardware specific component, i.e. the interface with the PC. The LolaX structure is illustrated in Fig. 6.1.

The idea of an interface definition is not new, a first attempt was done by S. Ludwig in his dissertation [49]. In his implementation a device driver was automatically generated from the Lola program. This was possible since the FPGA used – the XC6000 – has a fixed interface with the PC, i.e. the address and data bus were already defined in the FPGA architecture itself. The logical address of the variables is equivalent to the physical location of the variable registers, i.e. the position of the variable in the FPGA cell space itself. This made it possible to implement the driver directly, starting from the Lola description. For other FPGAs this is not possible and therefore our need to add information to the Lola description.

6.2.2 The Back-end

The back-end implements the communication with the re-programmable FPGA system. It has to program the device itself, i.e. downloading the FPGA code to the device, but also to access the address space implemented on the device. This because the hardware interface PC/FPGA is fixed and cannot be modified, from the PC side. The FPGA configuration is device dependent, and has to be implemented for each new device type. The access of the address space can be simplified by using a simple general RAM interface access, illustrated in Fig. 6.2. Remember that every
accessible variable is associated with a logical address. In case of a **read** the FPGA program has to decode the address and copy the value from the desired address into the data bus. In case of a **write** the value present on the data bus has to be copied into the desired address. This kind of communication is very simple to implement into any FPGA. The overhead for implementing the interface is also relatively small. The time delay is FPGA dependent and therefore has to be considered carefully.

### 6.2.3 The Front-end

The Front-end reads the LolaX symbol file and determines which variables are accessible on the device. Each variable is identified through its name, and its logical address. This information is used to access the variable in the FPGA.
6.2 The Language Lola and its Extension LolaX

6.2.4 Example

To clarify this concept, an example is given. A four-bit counter (variable \( x \)) and a constant (variable \( y \)) are implemented as a LolaX program. Both variables are mapped to a logical address which is used by the driver to read them. The implementation is done translating the LolaX code into the XNF – Xilinx Netlist Format [81] – description and then compiled and downloaded to a Xilinx Spartan Chip.

The LolaX program:

```
MODULE Example;

IN ck: BIT; pc: [6] BIT;
OUT stat: [4] BIT;
VAR x, y: [4] BIT;

CLOCK ck;
```
ADDRESS pc[1..2]; (* defines the address space *)
DATA stat[0..3]; (* defines the data space *)
WRITE pc[0];

BEGIN
(* physical positions *)
ck :: 13;
pc[0] :: 44; pc[1] :: 45; pc[2] :: 46;
stat[0] :: 70; stat[1] :: 77;

(* logical position in the address space*)
x :: 0;
y :: 1;

(* implements the read interface with CPU *)
stat[0] := MUX(pc[2]:MUX(pc[1]: x[0], y[0]), MUX(pc[1]: '0', '0'));
stat[1] := MUX(pc[2]:MUX(pc[1]: x[1], y[1]), MUX(pc[1]: '0', '0'));
stat[2] := MUX(pc[2]:MUX(pc[1]: x[2], y[2]), MUX(pc[1]: '0', '0'));
stat[3] := MUX(pc[2]:MUX(pc[1]: x[3], y[3]), MUX(pc[1]: '0', '0'));

(* y is a constant *)
FOR i := 0 TO 3 DO
  y[i] := '1';
END;

(* x is a counter *)
x[0] := REG(-x[0]);
x[1] := REG(x[0] - x[1]);
x[2] := REG(x[0]*x[1] - x[2]);

END Example.

The resulting FPGA circuit generated by the Xilinx place and route software is shown in Fig. 6.3.

The driver can access the x and y variables of the FPGA circuit as follows:

.... LolaResources.Get("x", i); LolaResources.Get("y", j);

6.3 The XNF Translation

Using the Interface Definition mechanism we are now able to access the data implemented into the FPGA in a simplified way; we still need a method to implement the device efficiently. Dif-
There are two possible approaches: to implement an FPGA placer/router or to relay in a sort of translation from LolaX to a standard definition which will be used as input for external tools. This last approach has been successfully used by E. Oertli for the Switcherland project [55]. In his implementation the Lola program was translated to a XNF [81] description for the XC5000 FPGA family. For our project we did not want to restrict the translation to a single FPGA family. Therefore, we rewrote the translation using a subset of the XNF definition which is standardized for all FPGAs. In this way all Xilinx FPGAs can be programmed starting from a Lola description.

Moreover, this decoupling allows us to implement the whole system in a CAD Tool not compatible with Lola, and then simply use LolaX to simplify the device driver implementation.
6.4 Is the Analog Case Resolved?

The undisputed advantage of re-programmable digital circuits, like FPGAs or PLDs is clear. This, however, is not the case for re-programmable analog circuits, since such devices have not yet been as successful as re-programmable digital circuits.

In the case of digital re-programmable devices, the only disadvantages compared to dedicated digital designs, is the reduced speed and their relatively large chip size. These are relatively small penalties to pay compared with the flexibility of re-programming the device.

For analog devices such penalties are much higher and therefore only few such re-programmable analog devices are available today. More precisely the speed, power consumption and tuning of the analog design are more important than the flexibility to re-program them. Most of them are arrays of operational amplifiers.

In our framework we decided to use a simple ADC/DAC card which covered our need for generating and reading analog signals. In addition, we noticed that there is a trend with sensor and actuator manufacturers to use digital signals for the interfacing. Therefore, we concluded that the effort to embed re-programmable analog devices in the framework would not bring any relevant advantages.
Chapter 7

Numerical Simulation

In this Chapter we describe the sub-framework for numerical simulation. The sub-framework is subdivided into three different functions. Such functions are dedicated to computing the mathematical model of the actuators, the process models and the sensors.

The actuators gather the information from the embedded system and simulate the physical actuators. The model is the mathematical representation of the simulated process, and the sensors simulate the measurements of the physically sensed information and send the results to the embedded system.

The model is therefore connected to one or more actuators and sensors. Whereas the actuators and the sensors are asymmetrical, the actuators get data from the ECS, and the sensors send data to the ECS. For a better understanding of the concept we will use the same simple spring mass system used in Section 4.3.4.

7.1 Description of a Process

A process can be described in many different ways, but for a numerical simulation a formal specification is obviously more appropriate. A formal description of a process holds all the information needed by the simulator in order to allow the system to mathematically compute the process behavior from the embedded system response. Therefore, we assume that given a starting condition and a sequence of embedded system commands we are able to compute a trajectory, i.e. the state of the process, for all time sequences [44].

Actually there are only two ways to describe a system: a graphical representation, and through mathematical formulas. Many simulation tools have chosen the first strategy, which is the most
intuitive way of building complex systems based on simple building blocks. The blocks, which are themselves mathematical descriptions of a process, can be easily joined together, forming a complex graphical system representation. This intuitive approach was inspired by the graphical interfaces used in CAD systems. However, there is a big difference: in the case of CAD systems you build a new block based on smaller blocks with a smaller set of functions. In the simulation case you are trying to simulate a known process. Therefore, you divide the system in smaller sub-blocks with known mathematical models. This does not mean that it is a difficult approach, but it is different.

The textual approach using mathematical formulas is more general since you are not bound to the constraints of the building blocks. But it is not as intuitive to read as a graphical representation. Studies have been conducted on this problem, see Ganz [27], trying to find the advantages and disadvantages of these methods.

We chose the representation of the system in form of mathematical formulas expressed in a procedural language formalism.

### 7.2 System Identification

In this dissertation we are not concerned with the problem of finding the mathematical representation of the process. This mainly because such a representation is already needed in the previous stages of the development – see Chapter 2, e.g. the implementation of the controller. The sys-
7.3 The Class Model

*System identification* consists in gathering data from the process, measuring the system response to different kinds of inputs, and mathematically find the closest mathematical description of the system [48].

### 7.3 The Class Model

The model is the mathematical representation of the process. In the spring system example the model will consist of an object with two states: the position and the velocity of the spring, and its dynamic. In order to implement a new model we have to extend it and override the following methods:

```plaintext
PROCEDURE (s: Spring) Init();
PROCEDURE (s: Spring) SetIntegrator(i :Integrator);
PROCEDURE (s: Spring) SetDynamic(i : PROCEDURE (ydot, y, u, x: Vector));
PROCEDURE (s: Spring) Limit();
PROCEDURE (s: Spring) Stop();
```

In the Init() method we have to build the data structure describing the state vector of the process – called StateDir, which will be explained later.

If we assume that the controller of the spring system runs at a rate of 50Hz, the simulator must run at a rate greater then 50Hz. The simulator has to read the force applied by the actuator, compute the new states, i.e. the new position and velocity of the spring and write the sensor information. The following code is how this is implemented in the base class Model.

The pseudo code:

```plaintext
PROCEDURE (s: Model) Step(dt: REAL);
BEGIN
  s.GetInputs;
  s.UpdateInputs;
  s.integrator.Step(dynamic, b.state, b.state, b.input, b.error, dt);
  s.Limit();
  s.SetOutputs;
END Step;
```
7.4 The Class Actuator

The actuators have two main tasks, one is to gather the data from the ECS, and the other is to simulate the behavior of the actuator itself. In case of asynchronous inputs, the actuator has simply to read the data as soon as it is available, if the simulation rate is too slow it has to install a task with a higher rate in order to assure that all incoming inputs are received.

PROCEDURE (a: SpringActuator) Init();
PROCEDURE (a: SpringActuator) Get();
PROCEDURE (a: SpringActuator) Stop();

7.5 The Class Sensor

The sensor gets the actual state of the model and computes the sensor behavior. Thereafter, it generates the output on the specified port. In case of asynchronous outputs, the sensor can install an asynchronous task on the scheduler in order to simulate the asynchronous behavior or it can simply approximate the synchronicity in units of the simulation steps.

PROCEDURE (s: SpringSensor) Init();
PROCEDURE (s: SpringSensor) Set();
PROCEDURE (s: SpringSensor) Stop();

7.6 Binding Model, Actuators and Sensors

The model, actuators and sensors must exchange data. This happens through the binding of the actuator with the model and with the sensors. The model reads the actuator values from the bounded actuators, similarly the sensors read the values from the model and write them to the output.

7.6.1 Static versus Dynamic Binding

To implement the binding mechanisms a dynamic approach has been used. This means that the actuators, model and sensors are bound during initialization. We could bind them at compile-time, but this would require a recompilation if some sensor or actuator is changed. The dynamic
approach however has the problem of consistency. Somehow we have to prove that the connected actuator and sensors are delivering or gathering the correct information, and in the correct format.

All intercommunication is done using vectors. Vectors are open arrays of REAL values. To allow the dynamic binding we use a description index, called the StateDir. This structure describes the values of the array, see Fig.7.2. Using the description the model is able to collect the needed values from the actuators. The collection is done by searching in all the bound actuators an output with the same name. Only if all the needed names are found by the model the system can start a simulation. To efficiently implement this mechanism, the collection is done during initialization and the indexes of the needed actuator outputs are stored. During the simulation the stored index is then used to copy the value from the actuator to the input vector of the model.

![State Vector Diagram](image)

**Figure 7.2: Representation of the description index (StateDir)**

### 7.7 Integration

The main problems in the integration of the differential equations are the time needed in the computation and the precision. The framework allows the user to use the predefined integration routines or to add new and more precise integration algorithms. The only constraint is that the routines must be based on fixed integration steps. Therefore, no algorithm with adaptive integration can be used. This because of the real-time constraints, which determine the duration of the integration step.

The framework offers only two standard integration algorithms – Newton-Euler and Runge-Kutta –, but new algorithms can be easily implemented and added to the framework. For discrete systems, no integration step is needed.
7.7.1 Discrete Systems

If the model would have been specified using *difference equations*, the model would not need any integration step. The computation is much simpler but the problem is that difference equations are only valid for the specified sampling time. Therefore, if a change in the sampling time is needed the difference equation has to be modified accordingly. For continuous differential equations the change does not make any difference.
Part III

Case Studies and Their Implementation
Seite Leer / Blank leaf
Chapter 8

A Simulator for a Model Helicopter

In this Chapter we will present in detail the helicopter system simulator for the OLGA ECS. The OLGA ECS was described in Chapter 2 as a case study. We will test the system using the HIL simulator framework. Due to the complexity of the ECS we subdivided the tests into two parts, the first tests the engine and main rotor system, the second the hovering of the helicopter.

8.1 The Helicopter Dynamics

There are many different mathematical models for helicopters. All of them are very complex and non-linear. This, of course, because the real helicopter is a very complex system. However, there are reduced models, which take into account some kind of simplification.

For this case study two simplified mathematical models were analyzed. The first called OLGA and the second called R-50 will be described in the next section. For a complete mathematical model of a full-scale helicopter we refer to [32].

For the HIL simulator implementation we chose the OLGA mathematical model which was used for developing the OLGA ECS, i.e. for tuning and computing the controller system. Simulating this model we discovered, however, that flying the simulated helicopter by hand gave the impression to the pilot not to be flying a real helicopter, this because these two models were restricted to hovering. Therefore, to use the simulator as a real flight simulator we should change the model to a more complex one which simulates also other kinds of flight behaviors.
8.1.1 Yamaha R-50 Based Model

This model was developed at Carnegie Mellon University in Pittsburgh. The motivation behind this model was the design of an autonomously flying helicopter based on the Yamaha R-50, an unmanned helicopter with a rotor diameter of 3 meters \[74\]. This model based on physical properties of the helicopter was identified using real flight data, and is limited to the helicopter hovering state. The description of the dynamics and the value of the identified parameters can be found in \[53\]. The linear mathematical description of the helicopter system has 14 states and 5 inputs.

The state of the helicopter is \((x, y, z \text{ position}; u, v, w \text{ velocity}; p, q, r \text{ rotation velocity}; \phi, \theta, r_{fb} \text{ attitude}; a_{1s}, b_{1s} \text{ rotor flapping})\), the state equations are:

\[
X = \begin{bmatrix} x & y & u & v & p & q & \phi & \theta & a_{1s} & b_{1s} & z & w & r & r_{fb} \end{bmatrix} \\
U = \begin{bmatrix} \delta_{lat} & \delta_{lon} & \delta_{ped} & \delta_{col} \end{bmatrix} \\
\dot{X} = AX + BU
\]
8.1 The Helicopter Dynamics

8.1.2 OLGA Based Model

This model was developed at ETH Zurich for the OLGA project, the helicopter has a 35ccm engine and a rotor diameter of 1.80 m. For a full description of the model we refer to [10]. The model is nicely subdivided into 4 decoupled systems, one for the x-direction, one for the y-direction, one for the z-direction and one for the yaw angle. Similarly to the R-50 model, it was identified using real flight data, and it is limited to the helicopter hovering state.

\[ X = [ x \ u \ \theta \ q ] \]
\[ Y = [ y \ v \ \phi \ p ] \]
Also in this case we refer to [10] for the identified parameters. To note is that, with difference to the R-50 model, in this model also the rotation speed of the main rotor $\omega$ is computed. This will be used to test the engine controller; for the hovering test it will not be computed, since it is assumed to be constant. Moreover, in some cases the engine is controlled open-loop, this means that the value of the throttle is fully coupled to the value of $A_m$. In this case the simulator has to
8.2 The Actuators

8.2.1 Manually Actuated Servos

This is the most simple type of actuator, the user can change the values of the inputs manually by pressing buttons on the screen. This rather simple actuator is useful as a test for the system reaction.

For example, in Fig. 8.3 we excited the system and were able to visualize the helicopter response. This allows us to check if the system is working as expected.

8.2.2 RC Servos

This actuator simulates the servos that control the movements of the cyclical pitch, the rotor blades, the collective pitch, the tail rotor pitch and the engine throttle in the real model helicopter.

The servo inputs are fed by a 0-5V signal which encodes the position of the servo. The servo can be positioned within 90°. The encoded signal uses a pulse width modulation, i.e. the width of the pulse is proportional to the position of the servo. The signal has a variable frequency between 40Hz and 50Hz. Figure 8.4 shows a PWM signal and a servo.

In order to control the helicopter we need to decode 6 servos: 4 are used to control the main rotor, 1 for the tail rotor and 1 for the throttle. The values of the servos have to be transformed to the \(U\) vector which is the input to the mathematical model of the helicopter. This translation is very helicopter dependent.

To do the translation we compute the pseudo-inverse of the translation matrix \(M\) and the offset matrix \(O\):

\[
\begin{bmatrix}
    sl & sr & sf & sb & st & sg
\end{bmatrix}^T = M \begin{bmatrix}
    A_{1s} & B_{1s} & A_m & A_t & GAS
\end{bmatrix}^T + O
\]

The matrix \(M\) has a fixed structure which is dependent on the swash-plate configuration, in the
case of a 4-point configuration the matrix has the following form:

\[
M = \begin{bmatrix}
x_{la} & 0 & x_{lm} & 0 & 0 \\
x_{ra} & 0 & x_{rm} & 0 & 0 \\
0 & x_{fb} & x_{fm} & 0 & 0 \\
0 & x_{bb} & x_{bm} & 0 & 0 \\
0 & 0 & x_{tm} & x_{tt} & 0 \\
0 & 0 & 0 & 0 & x_{g}
\end{bmatrix}
\]

To measure the coefficients we simply need to measure the PWM servo values for the fully open (=1), middle (=0) and fully closed (= -1) positions. Using these values we are able to compute all the coefficients.
8.2 The Actuators

In case of a 3-point swash-plate the $M$ matrix becomes simpler, and reducing the number of the needed servos to 5: $[ sf \ sr \ sc \ st \ sg ]^T$: 

Figure 8.4: PWM signal and a servo

Figure 8.5: Swash-plate
In order to compute the $U$ vector we need to use the pseudo inverse $P = (M^T M)^{-1} M^T$. We are then able to translate the servo encoded values to the input vector as follows

$$U = P[s_{0}s_{1}s_{2}s_{3}s_{4}s_{5}]^T - PO.$$

To read the signals we need to build a dedicated hardware in order to decode the PWM signals. To do this we used an FPGA card connected to the parallel port of the PC. The FPGA card, shown in Fig. 8.6 is a prototype card manufactured by XESS corporation, we modified it in order to decode the PWM signals. The next section illustrates the FPGA interface for the decoder.

![FPGA card](image)

Figure 8.6: FPGA card

**Hardware-Spartan XS10**

The customized hardware is implemented on a Xilinx Spartan XS10 FPGA. The digital circuit is implemented using the Xilinx software, the schema is in Appendix D. The FPGA implements the decoding of the PWM signals generated by the RC receiver or by the ECS. In the first case we are able to fly the simulated helicopter manually by using an RC sender, in the second case we are able to fly it via the ECS, i.e. the autopilot.

The decoding of the signals is implemented using a sequential mechanism in order to reduce the needed counters. In a naive approach, we used for each of the 6 incoming signals one 12-bit
8.3 The Sensors

counter to count the width of the pulse – the main clock is 1MHz giving a maximal pulse width of 2,048ms. The used resources were too big for the small XS10 device. We optimized the circuit in order to use only one counter for all signals. This is possible since all the incoming PWM are sent sequentially and not in parallel. This was discovered by Niklaus Wirth during the implementation of the decoders for the stability augmentation system Horla [12], a simplification of the full autonomous OLGA project.

Moreover, the digital circuit generates pulse frequency modulated signals – PFM – to simulate the rotor hall sensor. This will be described later in the relative sensor section.

The software interface of the FPGA with the simulator is implemented using the LolaX compiler. The LolaX code is also in Appendix D.

8.2.3 Replay Servos

We are also able to read the value of the servos from a file, this allows us to replay a flight on the simulator. This can be very useful for comparing the simulated behavior and the real behavior of the helicopter. To read the data we use the data format presented in Appendix B. Care must be taken in this case since the simulator will most likely not behave like the real helicopter, because the real helicopter movements are generated by external events like wind etc. and cannot be simulated. The reaction of the servos on the real helicopter will therefore generate a different movement on the simulated helicopter.

In case the helicopter is moved actively by the ECS, the simulated helicopter will give similar results like the real helicopter. In Fig. 8.7 a short sequence is shown were the measured data of a real flight, i.e. \([A_{1x}, B_{1x}, A_{f}, A_{mn}]\) are used as inputs for the simulator. The shown output is the simulated output of the \(p\) gyroscope, i.e. the gyroscope measuring the rotation in the \(x\)-direction. The similarity of simulated versus real system is evident. The simulator does not simulate the vibrations generated by the main rotor, which are the cause of the oscillations in the real measured signal.

8.3 The Sensors

8.3.1 Analog Gyroscopes and Accelerometers

These sensors generate an analog value proportional to the rotational rate (gyroscopes) and acceleration (accelerometers). The simulated sensors generate 6 analog values: 3 gyroscopes and 3 accelerometers. The main problem is to handle correctly the scaling factors used by the real gyroscopes and accelerometers. Moreover, the accelerometers have to be augmented by the ac-
celeration generated by gravity. To do this we need to know the actual pitch, roll and yaw angles, i.e. the attitude of the helicopter, and add the gravity vector, to the simulated accelerations. This can be done multiplying the $g$ vector $[0 \ 0 \ 9.81]$ with the rotation matrix $C_{e}^{b}$, which transforms the vector from earth-fixed coordinates to helicopter-fixed coordinates.

$$C_{e}^{b} = \begin{bmatrix}
\cos \psi \cos \phi & \sin \psi \cos \phi & -\sin \phi \\
\cos \psi \sin \phi \sin \theta - \sin \psi \cos \theta & \sin \psi \sin \phi \sin \theta + \cos \psi \cos \theta & \cos \phi \sin \theta \\
\cos \psi \sin \phi \cos \theta + \sin \psi \sin \theta & \sin \psi \sin \phi \cos \theta - \cos \psi \sin \theta & \cos \phi \cos \theta
\end{bmatrix}$$

Multiplying $C_{e}^{b}$ with $g$, this is simplified into:

$$C_{e}^{b} g_{\theta} = 9.81 \begin{bmatrix}
-\sin \phi \\
\cos \phi \sin \theta \\
\cos \phi \cos \theta
\end{bmatrix}$$

To generate the analog values we used a six-channel digital analog converter (DAC). The implementation of the driver is very simple and therefore not very important to discuss.

## 8.3.2 Rotor Hall-Effect Sensor

This sensor simulates the pulses generated by a hall-effect sensor which generates 3 pulses per rotor revolution. It is implemented by the FPGA. The sensor generates a pulse of fixed width with the rotor revolution period. This is not a real reproduction of the real sensor, since the pulse
width should be proportional to the rotor period. We simplified this generation, since we know that the rotor signal decoder does not take into account the pulse width, but only its rising edge. In another autopilot system [11] the decoder counted the low period of the signal, taking indirectly into account the high pulse period. The software decoder had to add a value proportional to the computed period. This to show that in order to implement an HIL simulator it is important to know how the ECS is implemented. Moreover, implementing an FPGA circuit with a variable pulse width is more complex than one with a fixed pulse width.

Figure 8.8: Rotor pulse at 666Hz
($\frac{1}{3 \cdot 0.00055}$)
8.4 3D OpenGL Visualizer

This virtual sensor helps to get a more realistic view of the flying helicopter. This is necessary in case of manual flight. In case of ECS controlling the system, no 3D visualization would be necessary. Even in this situation, however, it is nice to get a 3D feedback of the helicopter behavior in order to analyze how the ECS is controlling the system, without interpreting the off-line data.

The visualizer is developed using a standardized 3D library, i.e. OpenGL [80]. It is implemented as an independent viewer which gathers the data from the sensor.

Moreover, the visualizer was extended in order to be able to replay recorded flights from a real helicopter flight or even from a simulated recorded flight. This is possible since both systems use the same file format to store the data (see Appendix B for the definition of the file format).

We also used the same visualizer as a training platform. The visualizer allows us to learn how to use the ECS, i.e. to see the behavior of the helicopter.

An example of the 3D view is shown in Fig. 8.9.

8.5 Testing and Results

We used the simulator to HIL simulate two different things: first the main rotor controller part of the helicopter system. The second simulation was used for testing the hovering controller part.

8.5.1 Main Rotor Controller Testing

This is a single input, single output simulation – SISO. The input to the system is a PWM signal which controls the throttle of the gas servo. The output of the system is a PFM signal which encodes the rotor speed $\omega$.

The simulator is structured as shown in Fig. 8.10. All the input signals are read from the ECS, but only $\omega$ is computed as output. The model used is a subset of the model presented in 8.1.2. The model has two inputs, i.e. $A_m$ and GAS, in our simulation we set $A_m = 0$, since no vertical movement is simulated.

In Fig. 8.11 the results of a test are shown. The spikes generated by the rotor signal on the left are generated using FAUSEL, which was used to reproduce spikes on the signal. As you notice on the right side, the controller detects the errors. This kind of errors were discovered on the real
OLGA helicopter – in Fig. 8.12 you see a real measured data set. We were able to reproduce them on the simulator in order to test whether the ECS could filter out such spikes correctly. Using
FAUSEL we were able to start the error sequences during a simulation run and automatically prove the ECS correctness.

Figure 8.11: Simulated rotor signal and ECS response

Figure 8.12: Real measured rotor signal with spikes
8.5 Testing and Results

SIMULATE SpikeTest;

    FAULT Spike IS Rotor.Spikes
    START 90.0;
    STOP 120.0;
    PERIOD 0.02;
    SPEC
       G((GAS <= 1700) & (GAS >= 1200));
    END Spike;

END SpikeTest;

8.5.2 Hovering Controller Testing

In this case the helicopter is in hovering modus, the controller is switched on and the ECS will control the simulated helicopter.

The simulator reads the input from the ECS, i.e. $A_{15}, B_{15}, A_m, A_t$. The simulator can choose different types of actuators as described in the previous sections. The simulator computes the helicopter state, i.e. position and velocity and generates the value of the sensors. Different sensor signals can be used. The structure of the ECS is shown in Fig. 8.13.

![Signal flow for control attitude testing](Figure 8.13: Signal flow for control attitude testing)
The user interface of the simulator is quite simple, and allows us to configure the actuators and sensors to be used by the simulation run. Furthermore, the GUI allows us to compile the FAUSEL description that will be used in the next simulation run.

![Graphical user interface for attitude simulation](image)

**Figure 8.14:** Graphical user interface for attitude simulation

In Fig. 8.15 the response of the system with respect to errors generated by simulating wind gusts acting on the tail rotor is shown. The errors were generated using FAUSEL and the specified correct response is that in a maximum 10 seconds the yaw angle has to be brought back to its original yaw position (with a maximum 1% error, i.e. \( \pm 0.03 \) rad).

```plaintext
SIMULATE WindGusts;

FAULT Gust IS Helis.Wind
START 10.0;
STOP 30.0;
PERIOD 0.02;
```
8.5 Testing and Results

Figure 8.15: ECS response to wind gusts

```
SPEC
F[0,10]|{(yaw <= startyaw + 0.03)&(yaw >= startyaw - 0.03)});
END Spike;
END WindGusts;
```

8.5.3 Conclusion

During these two HIL tests on the OLGA ECS we were able to assure the correct implementation of the ECS. These two tests also demonstrate the ease of simulating faults like spikes and special conditions like wind gusts. There are other advantages given by the use of such tests. They can be summarized briefly as:
Added safety: For example it is impossible to test in flight that the spike detector algorithm works. This because the only way to test it is to artificially generate those spikes. During the real development of OLGA, the tests were done keeping the helicopter on the ground letting the main rotor rotate at an almost take-off rotation speed. In this way real-spikes were generated, and the system was tested. This was obviously a dangerous test. Using HIL we were able to prove that also during flight such spikes are correctly detected.

Deterministic testing: We are able to repeat deterministically the same test, over and over again. This allows us to test the system after each ECS tuning and after correction. This is impossible using the real platform.

Rare condition testing: Using such tests it is possible to reproduce conditions and faults that are rare in reality, but for safety and completeness have to be handled by the ECS. Testing such faults on the real platform is very difficult, since the rare event has to be reproduced artificially and could be catastrophic. For example in the hovering condition we could easily simulate a power loss condition of the main engine. Such a fault condition could be caused by many factors, for example by an abrupt change in air temperature and density. These events are very rare but could happen if the helicopter is used in hostile environments, and the ECS would need to be able to handle such conditions.

Dangerous testing: We are able to test faults that are very dangerous for the user and for the platform itself. The spike detection test is exactly such a test. If the handling were incorrect, the helicopter would crash. The spike would then be interpreted as a very high rotational speed which would cause the controller to lower the engine throttle drastically, or as a very low rotation speed which would cause the controller to increase the engine throttle drastically. In both situations the helicopter has a high probability of crashing.
In this chapter we present the implementation of an HIL simulator for the barrage of a hydro-powerplant. The simulator models a barrage system which was inspired by a real hydro-powerplant installed by the Rittmeyer company, Switzerland.

The goal of this simulation is to prove that our concept is general enough to be applicable also to other real existing problems, i.e. not only to the helicopter system.

9.1 The Barrage

The hydro-powerplant barrage (see Fig. 9.1) contains two moving parts, the segment and the flag. The segment is moved by a motor controlling angle $\alpha$, which moves the barrage upward or downward. The obtained aperture is $h_1$. The water from the reservoir can flow below the barrage. The flag is located on the top of the segment and is moved by a motor controlling angle $\beta$. The flag changes the height $h_2$. The water can thus also flow over the barrage.

9.1.1 Simulation Software and Dynamic Model

The software is divided into two parts: the barrage simulation and the faults generation for the testing. The differential equation of the barrage is:
were $\alpha$ is the angle of the segment, and $\beta$ is the angle of the flag. The parameters $\omega_s, \omega_f$ are the angular velocity of segment and flag, and the parameter $u_s, u_f$ are the motor input commands, $f_s, f_f$ are the mathematical modelling of faults. The values of $u_s, u_f$ are limited to $-2, 2$, and in normal operation the controller limits its output to $-1, 1$. 
9.1 The Barrage

9.1.2 Fault Simulation

The possible faulty conditions we want to test on the ECS are:

- **Leak**: an oil leakage from the motor \( \alpha \) forces the barrage to close, reducing the height \( h_1 \).
- **Stuck**: the segment cannot be completely closed because something is blocking the segment.

These two faults must be detected and a fault reaction procedure has to be started by the control system. In the leakage case the controller will simply maximize its output power, in order to compensate the position error, and release an alarm. In this way the controller is able at least to stop the segment from closing. In the stuck case the controller will try to open the segment in order to clean the reservoir ground, and thereafter will re-try to close the segment. In this discussion we focus only on the motor \( \alpha \).

The simulator and the ECS are running at a 5 Hz frequency, and communicate via a simple serial connection. After starting the simulation and compiling the FAUSEL description (see the code of the used FAUSEL description in Fig. 9.2; the generated automata are shown in Fig. 9.3) the ECS will control the barrage to a given reference position for the motors \( \alpha \) and \( \beta \). The simulator will analyze the ECS response, traversing the automata generated by FAUSEL and will compute the new actual position of the barrage.

```plaintext
SIMULATE Barrage;

FAULT W0 IS Barrages.Stuck;
START 0.0;
STOP 50.0;
PERIOD 0.2;
SPEC (* open the barrage to clean*)
   F((alpha <= 1.48) & F(alpha>=1.65))
END W0;

FAULT F1 IS Barrages.Leak;
START 10.0;
STOP 30.0;
PERIOD 0.2;
SPEC (*detect the alarm *)
   F(G[0,5](u0 > 1))
END F1;

END Barrage.
```

Figure 9.2: Barrage FAUSEL description
In Fig. 9.4 we show the user interface of the simulator. The user interface also shows a simplified representation of the barrage and the actual simulated position of the segment and flag.

Figure 9.4: Barrage simulator interface
9.1.3 Results

In this section we show the results of two simulation runs, in the first we tried to open the segment to 110.0 degrees. After 10 seconds the simulator starts to generate a leakage fault and the correct ECS response are shown in Fig. 9.5 and in Fig. 9.6. Figures 9.7 shows the ECS response when no faults are generated. The correct ECS response is specified in FAUSEL as $F(G[0,5](u0 > 1))$, meaning that after the fault generation the motor $\alpha$ input eventually will augment its output over the normal limit of 1 for at least 5 seconds. This specification is automatically verified by the simulator.

![Figure 9.5: Leak-fault response](image)
Figure 9.6: ECS control output

Figure 9.7: No-leak response
In the second simulation run we started at time 0 the stuck fault for 50 seconds, or till the specified response was verified. In this case the specification is $F((\alpha <= 1.48) \& F(\alpha >= 1.65))$ – the specification uses radiant $\&$, meaning that eventually after the fault started the segment will go below 84.7 degrees, i.e. it will touch the obstacle, and after that eventually will go higher than 94.5 degrees. We could also add a temporal limitation to specify a maximal response time of 5 s and 0 s minimal reaction time; the specification would be $F((\alpha <= 1.48) \& F[0,5](\alpha >= 1.65))$. In Fig. 9.8 you see the response of the ECS in presence of a stuck fault, in Fig. 9.9 the ECS response in the normal case.

![Figure 9.8: Stuck-fault response](image)

Figure 9.8: Stuck-fault response
Figure 9.9: No-stuck-fault response
Part IV

Conclusions
Seite Leer / Blank leaf
Chapter 10

Conclusions

10.1 Summary

In this work the available approaches for developing embedded control systems – ECSs – are presented. Thereafter we explain why hardware-in-the-loop simulation is a valuable testing tool for ECS during implementation and in the final stage of the development. After these two important introductory digressions we present the framework concept and its implementation. The four sub-frameworks are explained and analyzed. They are:

- a real-time scheduler for the real-time simulation
- an object-oriented abstraction for implementing the mathematical model, actuators and sensors necessary for the simulation
- a generalized method for acquisition and generation of customized signals, and last but not least
- a language and object-oriented abstraction for generating faults and testing for the ECS response correctness

To demonstrate the concepts and the framework we utilize it to implement two very different simulations. The first implemented simulation is used for the testing parts of the OLGA system which is also presented as a case study for the development of ECSs. The second simulation, a simulator for a hydro-powerplant barrage system was inspired by another real existing problem.
10.2 Outlook

The HIL simulator concept is obviously not new but we think that adding a temporal logic prover makes this testing method more valuable, and we hope this will be also used for further developments.

The HIL simulator however cannot be described as finished since several additions could be implemented. One possible improvement could be to couple the simulator with existing computer aided control design software – CACD – systems like Matlab. This would allow us to define the tests already during the design of the controller systems.

Moreover, we think that FAUSEL could also be used for other topics and not only for HIL simulation. It could be used as a more advanced supervisor for controlling an ECS during real-time operation. For this reason we started a project which is porting the HIL software and more precisely the FAUSEL system to a more open platform which will then be simpler to deploy as a general ECS supervisor.

The implementation of the simulator for other platforms has to be studied. More precisely, it could be interesting to study if the implementation of HIL simulation using Matlab and a real-time Toolbox would result in the same capability, or if it would be easier to use state chart diagrams to check the ECS response instead of using FAUSEL and linear temporal logic.
Part V

Appendix
A.1 LolaX EBNF Syntax

The syntax and semantic of LolaX differs from the original Lola only in the addition of the keywords DATA, WRITE and ADDRESS. They are used to define which signals are used for interfacing the FPGA with a PC. Moreover, the position assignment :: is used also for internal variables in order to assign a logical position to the variable, i.e. its position in the address space defined by ADDRESS and DATA.

The complete syntax description can be found in [21, 76].

```
identifier = letter {letter | digit} ["'"].
integer = digit {digit}.
LogicValue = "'0" | "'1".
SimpleType = BasicType | identifier ["(" ExpressionList ")"].
BasicType = "BIT" | "TS" | "OC".
ExpressionList = expression {"," expression}.
type = [ "(" expression ")" ] SimpleType.
ConstDeclaration = identifier ":= expression ";".
VarDeclaration = IdList ":= type ";".
IdList = identifier {"," identifier}.
selector = [ "." identifier | "." integer | ["(" expression ")]").
factor = identifier selector | LogicValue | integer | 
"-" factor | "+" factor | "+" expression ")" | 
"MUX" ["(" expression ":" expression "," expression ")"] | 
"MUX" ["(" expression ":" expression "," expression "," expression "," expression "," expression "," expression ")"] | 
"REG" ["(" expression "," expression ")"] | 
```
"LATCH" "(" expression "," expression ")" |
"SR" "(" expression "," expression ")"

term = factor {{"*" | "/" | "DIV" | "MOD"} factor}.
expression = term {{"+" | "-"} term}.
assignment = identifier selector ":=" [condition ":"] expression.
condition = expression.
relation = expression {{"=" | "+" | "<" | "<=" | ">" | ">="} expression}.
IfStatement = "IF" relation "THEN" StatementSequence
{"ELSIF" relation "THEN" StatementSequence}
{"ELSE" StatementSequence}
"END"
ForStatement = "FOR" identifier ":=" expression "DO" StatementSequence "END"
statement = [assignment | UnitAssignment | IfStatement | ForStatement].
StatementSequence = statement {";" statement}.
module = "MODULE" identifier ";" {TypeDeclaration ";"}
{"CONST" {ConstDeclaration}}
{"IN" {VarDeclaration}}
{"INOUT" {VarDeclaration}}
{"OUT" {VarDeclaration}}
{"VAR" {VarDeclaration}}
{"CLOCK" identifier ";"}
{"DATA" IdList ";"}
{"ADDRESS" IdList ";"}
{"WRITE" identifier ";"}
{"BEGIN" StatementSequence}
"END" identifier ";".

FormalType = "[" [expression] "]" "BIT".
FormalBusType = "[" [expression] "]" ("TS" | "OC").

TypeDeclaration = "TYPE" identifier ["**"] ["(* IdList ")] ";"
{"CONST" {ConstDeclaration}}
{"IN" {IdList ":" FormalType ";"}]
{"INOUT" {IdList ":" FormalBusType ";"}]
{"OUT" {VarDeclaration}]
{"VAR" {VarDeclaration}}
{"BEGIN" StatementSequence}
"END" identifier.

UnitAssignment = identifier selector "(" ExpressionList ")".

A.2 LolaX Symbol File Format

LolaX symbol files are stored with the extension Syl.

LolaXSymbol = 0F0X name key:2 {lolanode} 0X {adrnode} 0X {datanode} 0X
{wrnode} 0X {possnode} 0X.
A.2 LolaX Symbol File Format

name = {ch:1} 0X.
lolanode = node.
adrnnode = node.
datanode = node.
wrnode = node.
possnnode = node.
node = ref [ simplenode | varnode | typenode | imptype | module].
simplenode = val:2 node node.
varnode = val:2 name node node node.
typenode = val:2 name node mno:2 tno:2.
imptype = mno:2 tno:2.
module = name key:2 mno:2.
Seite Leer / Blank leaf
Appendix B

Replay Data Format

This format is used to store the sequences of data on a target system. This format is also used, for example, by the helicopter control system [12] to store the measured and computed value during a flight and to do the post-processing of the signals. This file format has proven to be very flexible and versatile. Moreover, the files size can be optimized reducing the precision of the stored data. This is very important since the file is transmitted from the target to the host via a slow serial connection.

Furthermore, it is possible to configure the stored data very easily. In the helicopter system this was done using a textual configuration file, that allowed us to describe the values that had to be stored during the flight. The only drawback of this format is that the values are stored synchronously, and therefore it is difficult to store asynchronous values, like GPS messages.

In Fig. B.1 the off-line tool Watson of the helicopter project is shown. This tool was implemented on Matlab by Markus Kottmann [43]. In our HIL simulator we used the replay dataformat to do replays of real flown flight data, but also to store the state of the model during simulation.

```
bbxfile = header len:4
       {int32 | int16 | int8 | float32 | fix0 | fix2 | fix3} OFFX
       {string} starttime:4 {data:4 | data:2 | data:1 }.
string   = {char} 0X.
header   = char:32.
int32    = 0X.
int16    = 1X.
int8     = 2X.
float32  = 3X.
fix0     = 4X.
fix2     = 5X.
fix3     = 6X.
```
Figure B.1: The Watson analysis tool
C.1 FAUSEL EBNF

The EBNF description of the FAUSEL language:

```
FAUSEL = "SIMULATE" ident ";" {fault } [spec] "END" ident ";".  
fault = "FAULT" ident "IS" qualident [startpar];" 
    [exec] [spec] "END" ident ";".  
startpar = "(" expression ";=" expression  
    
exec = START expression ";" [STOP expression ";"] [PERIOD number ";").  
spec = "SPEC" [simplexpression [";"]].  
simplexpression = expression {("<" | "=" | ">" | ">=" | ">=") expression}.  
expression = ["+"] | ["-"] term  
    
   term = factor ["*" | "/" | "&") factor.  
factor = ["-" | ["F" [time] | "G" [time] | "X") factor | qualident |  
    [" simpleexpression "]" | number | "TIME" | "FALSE" | "TRUE").  
time = ["[" number [".", number])].  
number = ["0".."9"]{%["0".."9"]} ["." ["0".."9"]} ["0".."9"]}.  
ident = ["a".."z"] | ["A".."Z"]} ["a".."z"] | ["A".."Z"] | ["0".."9")}.  
qualident = ident [":" ident].
```
C.2 Fault Object

DEFINITON Faults;

    TYPE
        Fault = POINTER TO FaultDesc;
        FaultDesc = RECORD
            name: Ident;
        END;

        Variable = POINTER TO RECORD
            name: Ident;
            fault: Fault; next: Variable
        END;

    PROCEDURE (f: Fault) Get (name: ARRAY OF CHAR): REAL;
    PROCEDURE (f: Fault) Enum (VAR link: Variable);
    PROCEDURE (f: Fault) Init;
    PROCEDURE (f: Fault) FirstDa (time: REAL);
    PROCEDURE (f: Fault) Da (time: REAL);
    PROCEDURE (f: Fault) LastDa (time: REAL);

END Faults.
Appendix D

FPGA Schema

D.1 PWM Decoder and Rotor Signal Generator

This is the circuit used for the acquisition of the PWM signals. The circuit was implemented using the Xilinx CAD software.
D.2 LolaX Interface

MODULE PWMDecoder;

IN ck: BIT; pc: [6] BIT;
OUT stat: [4] BIT;
VAR
    pwm0, pwm1, pwm2, pwm3, pwm4, pwm5: [12] BIT;
CLOCK ck;
DATA stat[0..3];
ADDRESS pc[1..2]; (* defines the address space *)
WRITE pc[0];

BEGIN
(* physical positions *)
    ck :: 13;
    pc[0] :: 44; pc[1] :: 45; pc[2] :: 46;
    stat[0] :: 70; stat[1] :: 77;

(* logical position in the address space *)
    pwm0 :: 0;
    pwm1 :: 1;
    pwm2 :: 2;
    pwm3 :: 3;
    pwm4 :: 4;
    pwm5 :: 5;

(* NO IMPLEMENTATION NEEDED *)
FOR i := 0 TO 11 DO
    pwm0[i] := '0;
    pwm1[i] := '0;
    pwm2[i] := '0;
    pwm3[i] := '0;
    pwm4[i] := '0;
    pwm5[i] := '0;
END;
FOR i := 0 TO 3 DO
    stat[i] := '0;
END;

END PWMDecoder.
Seite Leer / Blank leaf
The following tables list the modules of the HIL framework and the two simulation examples illustrated in this dissertation together with a short description and the size of the Oberon source code given by the source analyzer. The system is available from the world wide web [82].

<table>
<thead>
<tr>
<th>Fault Generation Modules</th>
<th>Description</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Faults.Mod</td>
<td>Base type for faults handling</td>
<td>13720</td>
</tr>
<tr>
<td>Fausel.Mod</td>
<td>Parser and fault generator</td>
<td>18864</td>
</tr>
<tr>
<td>ISets.Mod</td>
<td>Sets used by LTL module</td>
<td>2565</td>
</tr>
<tr>
<td>LTL.Mod</td>
<td>Linear temporal logic parser/tester</td>
<td>39246</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Models Modules</th>
<th>Description</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integrators.Mod</td>
<td>Base type for integrations</td>
<td>1539</td>
</tr>
<tr>
<td>RungeKutta.Mod</td>
<td>Runge-Kutta integration extension</td>
<td>3386</td>
</tr>
<tr>
<td>WatsonLogger.Mod</td>
<td>Storage of the model state</td>
<td>4662</td>
</tr>
<tr>
<td>SimulationBase.Mod</td>
<td>Base type for simulation</td>
<td>15872</td>
</tr>
<tr>
<td>ModelGadgets.Mod</td>
<td>Interface to the gadget system</td>
<td>6650</td>
</tr>
<tr>
<td>Vectors.Mod</td>
<td>Base type for state information</td>
<td>4376</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scheduling Modules</th>
<th>Description</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BGScheduler.Mod</td>
<td>Background scheduler</td>
<td>5919</td>
</tr>
<tr>
<td>Scheduler.Mod</td>
<td>Real-time scheduler</td>
<td>6270</td>
</tr>
<tr>
<td>NBW.Mod</td>
<td>Non blocking write protocol</td>
<td>924</td>
</tr>
</tbody>
</table>
## Module List

### LolaX Modules

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lolax.Mod</td>
<td>Lola Compiler</td>
<td>9465</td>
</tr>
<tr>
<td>LSBx.Mod</td>
<td>LolaX Base module</td>
<td>7465</td>
</tr>
<tr>
<td>LSCx.Mod</td>
<td>LolaX Compiler</td>
<td>32379</td>
</tr>
<tr>
<td>LSSx.Mod</td>
<td>LolaX Scanner</td>
<td>5521</td>
</tr>
<tr>
<td>LolaBrowserx.Mod</td>
<td>LolaX Browser</td>
<td>7370</td>
</tr>
<tr>
<td>LolaResources.Mod</td>
<td>Lola Resource handler</td>
<td>3455</td>
</tr>
<tr>
<td>LolaXNF.Mod</td>
<td>XNF Translator</td>
<td>18831</td>
</tr>
<tr>
<td>Resources.Mod</td>
<td>Base type for all resources</td>
<td>2057</td>
</tr>
<tr>
<td>FPGAInterface.Mod</td>
<td>FPGA interface</td>
<td>38888</td>
</tr>
<tr>
<td>XSSInterface.Mod</td>
<td>XS interface</td>
<td>13457</td>
</tr>
</tbody>
</table>

### Helicopter Simulation Modules

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Helis.Mod</td>
<td>Main helicopter model</td>
<td>14031</td>
</tr>
<tr>
<td>HeliCompass.Mod</td>
<td>Compass sensor</td>
<td>1705</td>
</tr>
<tr>
<td>HeliIMU.Mod</td>
<td>IMU sensor</td>
<td>1984</td>
</tr>
<tr>
<td>HeliGLSensor.Mod</td>
<td>OpenGL visualizer</td>
<td>39020</td>
</tr>
<tr>
<td>HeliCrossbow.Mod</td>
<td>Crossbow IMU sensor</td>
<td>1705</td>
</tr>
<tr>
<td>HeliViews.Mod</td>
<td>Simple visualizer</td>
<td>9020</td>
</tr>
<tr>
<td>HeliRotors.Mod</td>
<td>Rotor PFM generator</td>
<td>2234</td>
</tr>
<tr>
<td>HeliRC.Mod</td>
<td>RemoteControl actuator</td>
<td>2339</td>
</tr>
<tr>
<td>HeliWePilotRC.Mod</td>
<td>wePilot RemoteControl</td>
<td>2522</td>
</tr>
<tr>
<td>HeliReplay.Mod</td>
<td>Replay actuator</td>
<td>4482</td>
</tr>
<tr>
<td>HeliController.Mod</td>
<td>Replay actuator</td>
<td>4686</td>
</tr>
<tr>
<td>HeliTrajectory.Mod</td>
<td>Trajectory model</td>
<td>4474</td>
</tr>
</tbody>
</table>

### Barrage Simulation Modules

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
<th>Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BarrageViews.Mod</td>
<td>Simple visualizer</td>
<td>6620</td>
</tr>
<tr>
<td>Barrages.Mod</td>
<td>Barrage model</td>
<td>9787</td>
</tr>
</tbody>
</table>
Bibliography


127


Curriculum Vitae
Marco A.A. Sanvido

Born in Sorengo the 3rd of January 1972

- September 1978 - August 1983
  Primary school.
  Viganello, Switzerland.
- September 1983 - August 1987
  Secondary school.
  Lugano, Switzerland.
- September 1987 - August 1991
  College, with scientific curriculum.
  Liceo Cantonale 1, Lugano, Switzerland.
- November 1991 - November 1996
  M.S. in Computer Science Engineering.
  Swiss Federal Institute of Technology (ETH) Zurich, Switzerland.
- November 1995 - April 1996
  Practicing Engineer.
  STMicroelectronics, San Jose, California, USA.
- April 1996 - November 1996
  Diploma Thesis: Development of a timing analyzer for the XC6200 FPGA.
  Swiss Federal Institute of Technology (ETH) Zurich, Switzerland.
- November 1996 - April 1999
  Research and Teaching Assistant with Prof. N. Wirth.
  Institute for Computer Systems
  Swiss Federal Institute of Technology (ETH) Zurich, Switzerland.
- April 1999 - April 2002
  Research and Teaching Assistant with Prof. W. Schaufelberger.
  Automatic Control Laboratory
  Swiss Federal Institute of Technology (ETH) Zurich, Switzerland.