A Hardware Abstraction Layer for Model-based Design of Embedded Systems

Julius Roob
University of Kaiserslautern, Embedded Systems Group
julius@juliusroob.de

Contents

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, keine anderen als die angegebenen Quellen und Hilfs- mittel benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Danksagung

An erster Stelle verdienen die Betreuer meiner Bachelorarbeit, Manuel Gesell und Prof. Dr. Schneider meinen Danke für ihre geduldige, stets freundliche und sehr motivierende Betreuung. Meiner Familie möchte ich hier dafür danken, dass ich mich stets auf ihre Hilfe verlassen kann und sie mir einen Ort bieten, zu dem ich mich zum Arbeiten und Entspannen zurückziehen kann. Auch meinen Freunden gebührt ein besonderer Dank dafür, in meinen schwierigen Phasen deutlich mehr meiner Launen ertragen zu haben, als ich jemals hätte verlangen können.

Zusammenfassung

Der Einsatz von eingebetteten Systemen in alltäglicher Elektronik nimmt stetig zu. Eine Konsequenz hieraus ist, dass diese in gleichem Maße wichtiger für die Zuverlässigkeit und Sicherheit dieser sind. Die hohen Kosten die Fehler in Software verursachen können, die Gefahr die solche auch für Menschenleben bedeuten können, sowie der große Aufwand einer manuellen Verifikation motivieren auch weiterhin die Erforschung modellbasierter Softwareentwicklung und automatischer Verifikationsmethoden.

Die synchrone Sprache Quartz, Teil des Averest Frameworks, ist eines der Resultate und Werkzeuge dieser Forschung. Sie erlaubt die Beschreibung von Modellen, welche sich durch Verifikation und Testsimulation prüfen lassen. Das ConceptCar, ein ferngesteuertes Modellauto welches die Datenbus-Struktur der Komponenten in modernen Fahrzeugen nachahmt, wurde als Plattform für Tests und Experimente zur Fahrzeugtechnik entwickelt. Es eignet sich somit hervorragend als Testgegenstand für die Werkzeuge des Averest Frameworks.

Die auf dem ConceptCar verbauten Mikrocontroller werden in C programmiert, welches eine der Sprachen ist, für welche Code aus Quartz Modellen generiert werden kann. Da dieser generierte Code keinen Zugriff auf die zugrundeliegenden Hardwarekomponenten hat, werden Treiber benötigt, die die benötigten Schnittstellen zur Verfügung stellen. Um diese Lücke zu schließen, wird diese Arbeit einen Hardware Abstraction Layer (HAL) vorstellen, welcher sowohl die benötigten Treiber als auch sonstige Infrastruktur und Werkzeug zur Verfügung stellt, um lauffähige Firmware aus dem generierten Code zu erzeugen.

Als Ergänzung zu diesem wird außerdem eine Transformation vorgestellt, die Modelle und für sie spezifizierte Testfälle zu ausführbaren Tests zusammenführt, welche sich zum Prüfen der Korrektheit der C Synthese verwenden lassen.

Abstract

Because of the widespread use of embedded systems in everyday electronic applications, and their importance for the safety and reliability of those, the demand for verification continues to provide a motivation for research on the topics of model-driven development.

The synchronous language Quartz, which is part of the Averest framework, allows the description of models which can then be subjected to verification and test-driven simulation. As a platform for testing and experimentation, the ConceptCar, a remote-controlled model car, emulates the bus-oriented structure of components in real automotive applications. This makes it an ideal test case for the features of the Averest framework.

The controllers of the ConceptCar are programmed in C, which is one of the languages available for synthesis from Quartz, but the generated C code lacks functions to access to the hardware on the ConceptCar. This gap will be bridged by the Hardware Abstraction Layer (HAL) defined in this thesis.

Also, a transformation on an intermediate representation of software models is introduced to merge test cases with programs, providing self-executing unit tests to check the correctness of the C synthesis.

1 Introduction

Electronics control many aspects of our everyday live. Nowadays, due to their flexibility and cheap prices when manufactured in large quantities, almost every device is controlled by one or more micro processors that have their function defined by software.

The inherent complexity of software makes it prone to mistakes, which, while causing only nuisance or financial damage in many cases, endangers human lives when happening in applications like aviation or automotive control systems. These require a formal verification of components to be approved for public use to alleviate the issue. Manual verification being very time-consuming and expensive, which, together with the cost of mistakes happening in cases where it is neglected, is the motivation for the research in automated verification of software.

One result of, and tool for this research is the synchronous language Quartz used for describing models of embedded systems. It is developed as a part of the Averest framework that features tools to compile models written in the language, generate hardware circuits or procedural programs from these, apply model-checking for verification and simulate the execution of models for test cases.

The hardware platform used for this thesis is the ConceptCar, a remote-controlled car. It is designed to provide a platform for experimentation and testing of concepts, and is thus modelled to come close to real automotive application. Featuring several Electronic Control Units (ECUs), individual boards driven by micro controllers and communicating through a common data bus, the structure of a real car employing drive-, break- and steer-by-wire is emulated. These ECUs have been programmed in C and C++, and, as Quartz can be synthesised to C, models defined in the synchronous language can be used to describe the software driving the ECUs. This will produce a set of models that allow for verification of both individual components and the ConceptCar as a whole, providing an ideal test for the model-checking capabilities of the Averest framework. This generated C code has no access to the hardware of the micro controllers and thus requires drivers to interface with these, which is the gap that will be closed by the Hardware Abstraction Layer (HAL) defined in this work. It will consist of drivers for the hardware, setup and glue code that is required to provide a working environment and interface for the Quartz modules as well as a build environment to automate the whole process of turning these modules into firmware images that are ready to be written to the micro controllers.

Finally, as an extension to this and for testing the C synthesis, a transformation will be described to merge programs written in Quartz with test cases to produce self-executing unit tests.

The content of this thesis is structured around the development process of the HAL. After this introduction and a section on related work, Section 3 will explain the given environment of this work, i. e., a detailed description of the Averest framework including the Quartz language and the hardware involved. Following that, Section 4 will list the functionality required, as well as reflecting the challenges involved with, and decisions made for, the Quartz API design. Section 5 will then focus on various aspects of the implementation, such as the makefile used for building the firmware, code required to run the code produced by the Quartz to C synthesis and methods for efficient interface access. It is followed by a report on the testing of individual components and the whole system in Section 6, which includes a collection of examples that show the usage of individual components of the HAL API from within Quartz. Prior to the suggestions for future work in Section 8, a summary and conclusion of this work is given in Section 7.

2 Related Work

This Section will be split into related work concerning model-driven development in general - and prior publications written in the context of the ConceptCar.

2.1 Model-Driven Development in General

Model driven development and verification has been, and still is, an important topic of research. As such, there are various previous publications on the topic regarding the development of software for embedded systems.

[10] describes an approach using Unified Modelling Language Testing Profile (UML-TP) to generate a testing framework for resource constrained real-time systems.

[11] proposes a framework using the Architectural Analysis Description Language (AADL) to describe platform-dependent code which interfaces synthesised software with, for example, the FreeRTOS embedded operating system. Focusing on common requirements of multithreaded software, it is not applicable to code generated by the Averest framework, but may be a subject for future work (Section 8.6).

In [19], a tool for the compilation of Java code to domain specific software is described, which utilises transformations on an intermediate Abstract Syntax Tree (AST). This is an approach that can not be applied to the Averest framework, because it is neither applicable to synchronous languages nor the guarded actions used as an intermediate representation thereof.

[12] gives a detailed description of the calculation of Worst Case Execution Time (WCET) and Best Case Execution Time (BCET), important characteristics of software in embedded systems, using models described in the Quartz language and applying temporal logics.

2.2 Publications Related to the ConceptCar

Being an experimental platform for research on embedded automotive applications, there have been publications made in the context of the ConceptCar, both in regards to software synthesis and security, as well as special applications.

The diploma thesis of Jonas Mitschang [13], describes both the architecture of the ConceptCar platform and an embedded application development process that utilises model-based design and C code generation. Models are defined in Matlab/Simulink and Stateflow, compiled to C by Real-Time Workshop, Embedded Coder and integrated with platform/operating system code by a java application called SimulinkTarget. Similar to the topic of this thesis, it does, however, not include any aspect of verification and model-checking and focuses on the application of tools that can not be applied to Quartz.

Another diploma thesis, by Marcel Zimmer [20], focusing on the implementation of security patterns in embedded systems, he analyses the possible causes of failure in the ConceptCar platform and describes the challenges of translating common security patterns to hardware-software codesign.

The master thesis of O. Rafique [15] is focused on the task of implementing the hard- and software for controlling the ConceptCar with an Android cellphone. This is achieved by designing and implementing a new ECU, the WirelessBoard, a Bluetooth interface and the accompanying Android application. While it does not mention software or hardware synthesis, it provides a comprehensive description of the architecture of the ConceptCar platform.

In [16], the ConceptCar is used as an example platform to describe the different levels of abstraction that can be applied when interfacing code generated by the Averest framework with hardware or

operating systems. While giving examples for possible implementation details, no complete system was produced, and it serves as a motivation for this thesis.

To conclude the related work, while there has been a lot of research on the topic of model-based software development and code generation for embedded systems, none of these is able to supply the C synthesis of the Averest framework with a HAL able to interface it with the specific hardware available.

3 Prerequisites

This section documents the platform for which the HAL is built, and the tools used are described. The Averest framework and Quartz language are discussed first, after which the ConceptCar platform followed by the architecture of the embedded micro controllers are described.

3.1 Averest Framework

The Averest Framework [1] is a toolchain developed by the Embedded Systems Group at the University of Kaiserslautern to provide a set of tools for the development and verification of reactive systems.

While the Quartz language described in Section 3.1.1 is used for formulating the models in a human-readable way, it is compiled into the Averest Intermediate Format (AIF) (described in 3.1.3) to produce a representation that is suitable for further processing.

There is a variety of AIF to AIF transformations (Section 3.1.4), which are used to prepare models for hardware/software synthesis and verification.

For each of these steps, the Averest framework provides an executable tool. These are explained in Section 3.1.2, and are part of the toolchain available on the project website [1], which also is a source for example code and documentation.

3.1.1 Quartz

```
1 pause;
2 a = 2;
3 b = 7;
4 c = a + 2;
5 pause;
```

Listing 1: Macro Step Example

```
1 module a(bool ?a, !b) {
2     ...
3 } drivenby testcase1 {
4     assert(!b);
5     a = true;
6     pause;
7     assert(b);
8 }
```

Listing 2: drivenby Example

The synchronous programming language Quartz[17] is the basis of model definition. It is a strongly typed language with formal semantics that is optimised for describing embedded systems and reactive systems.

Unlike procedural languages, programs written in Quartz do not consist of a sequence of instructions. They are comprised of macro steps, which represent logical steps in time and contain a set of instructions that are assumed to be executed in parallel. This is done explicitly by the programmer, in contrast to compiler optimisation done for out-of-order execution and VLIW processors.

An example is shown in Listing 1, where three assignments are executed in one macro step, separated from other macro steps by the **`pause;`** statement. Intuitively, these statements are comparable to a block of gates in a logic circuit set between clocked synchronisation elements. Variables can have boolean, numerical or data types compound of those, and assignments to these can be either instantaneous (`a = 3;`) or delayed (**`next`**`(b)= 4;`), in which case the value for the following macro step is set. They can be stored as either events, which are reset to a default value at the start of every macro step, or memorised variables that keep their value until a new one is assigned.

There are syntax elements known from procedural languages like **`if`**`(S1)B1` **`else`** `B2` and **`for`** loops, but have a behaviour that is highly dependent on the placement of **`pause;`** statements. For example, loops that do not contain a **`pause;`** are effectively unrolled, i. e., have all iterations executed in parallel in one macro step. In Quartz, parallelism is supported not only at the macro step level, but also for threads of execution. They can be either synchronous (`B1 || B2`), i. e., have their macro steps be executed in unison, or asynchronous (`B1 ||| B2`), which corresponds to concurrent threads with unknown schedule known from non-synchronous languages and systems.

Code is structured into modules that represent functional units, and packages, which are hierarchical collections of modules and sub-packages. Modules have an interface that contains variables which are declared as output, input or inout. Inputs are only readable by the module itself and have their values determined by the environment or the calling modules, while outputs can only be written by the module itself. Inout interface variables are reduced to outputs at link-time. Unlike procedural functions, the content of interface variables is not determined at the point of module instantiation, but can be assigned new values every macro step, depending on the type, by either the instantiating or the instantiated module.

For model checking, simulation and testing purposes, the Quartz language provides methods to define temporal logic constraints, which are explained in Section 3.1.7, and drivenby test cases on each module. Drivenby tests have the same structure as regular Quartz code, but have the module input variables for output and to provide inputs and check the correctness of output the module they test. This is done by placing assertions that fail when output expectations are not met, a basic example for which is given in Listing 2.

3.1.2 Invoking/Using the Averest Framework

The Averest framework provides several easy to use tools to translate Quartz to AIF, apply AIF to AIF transformations and synthesise C and Verilog, among others.

- *qrz2aif*: Translation from Quartz to AIF is done as follows:

  ```
  qrz2aif example.qrz
  ```

  After this, there is a file example.aif containing the guarded actions.

- *aif2aif* is a command-line interface for the transformations explained in Section 3.1.4. The list of supported transformations can be displayed by issuing the following command:

  ```
  aif2aif --help
  ```

- *aif2smv* creates input files suitable for the SMV symbolic model checker from AIF.
- *aif2c*: A call to aif2c with the intermediate file produces the `.c` and the `.h` files that contain the code and interface of the Quartz module. Their exact structure is explained in 3.1.6.
- *aif2txt* converts a Quartz module to a textual, human-readable representation, which can be used mainly for understanding and debugging the content of the generated AIF.

- *aif2trc* is another basic tool for debugging, which allows running Quartz modules in a simulator and printing all state information like variables and program flow at either macro or micro step granularity. To run a module that requires input in the simulator, at least one drivenby test case has to be written.

  By default `aif2trc` produces traces for all drivenby blocks of the specified module. To run only a single test case, the `-t <testcase name>` parameter can be used.

3.1.3 Guarded Actions & Averest Intermediate Format

Modules are written in Quartz and are compiled into the Averest Intermediate Format (AIF) before they can be used by the tools provided by the Averest framework. AIF is an XML format, into which the program behaviour as well as all information needed for verification and further processing are translated.

The basic representation of program behaviour in AIF is called guarded actions. Each guarded action is a conditional statement. As defined in [17], these statements can be either assignments, delayed assignments or assertions, which have been described in Section 3.1.1, or assumptions used by developers to give the model-checker explicit hints about program behaviour.

As the conditions and assignments include both the content of variables and the currently active program labels, they describe the data flow and the program flow of a model in a uniform way. Allowing the entire behaviour to be modelled as an equation system, this is very beneficial for the formal point of view.

When compiling Quartz to AIF, first a `.aifm` file is produced that contains the behaviour of the module, and which is then linked with other required modules to produce a finished `.aifs` file that contains all information necessary for execution. Because linking is done automatically and there is no reason to consider the unlinked modules in this context, AIF will refer to the AIFS throughout this thesis.

3.1.4 Transformations

The Averest framework features a set of AIF to AIF mappings which are used both to test transformations and as a basic tool for software and hardware synthesis. There are transformations to convert all data types used to scalar, boolean or bitvector types, which is required when a target component does not support the data types used natively.

For the synthesis of procedural code, the Extended Finite State Machine (EFSM) transformation is utilised to reduce the amount of guarded actions that have to be executed/evaluated in a macro step. This fact is, for example, reflected in the structure of the C code produced by the C synthesis, as can be seen in 3.1.6.

3.1.5 Hardware and Software Synthesis

Besides C, which is used in this thesis, there are other target languages to which Quartz modules can be translated. Two examples for those are the Verilog synthesis for hardware descriptions and a synthesis for Multiprocessing that generates C++ code that relies on the OpenMP framework to distribute parallel threads of execution to multiple processors.

3.1.6 Structure of Synthesised C Code

Each Quartz Module, when synthesised to C, provides exactly one method to be started with the number of Quartz macro steps, which is defined as follows:

```
1 || void adc(int __steps)
```

Synthesised Modules are not reentrant, they loose their state whenever they terminate or have executed the given number of macro steps.

As mentioned before, the C synthesis relies on the EFSM transformation to reduce the amount of guarded actions that have to be evaluated for each macro step. The detailed process is described in [7], resulting, if applicable, in a set of states for each of which only those guarded actions are evaluated that can fire. This leads to the C code having one loop containing a switch statement that embeds all code of the module that chooses the state, and thus code to execute, by a state variable.

For each of the interface variables given in Quartz, it expects to have access to a `WRITE_VAR(VAR)`, `READ_VAR(VAR)` or both macros defined in $\langle module \rangle\_io.h$ [1]. Also, there is a macro `STEP_FINISHED()`, which is called at the end of each macro step. Those - aside from the module being started by a function call - are the main interface between C and the Quartz module.

A short example, taken from the "Quartz Introduction" presentation slides available on the website of the Averest framework, is "ABRO". Both the Quartz and synthesised C sources are shown in Appendix C.

3.1.7 Verification and Model Checking

One main goal of the Averest framework is to provide a framework for the verification of model descriptions written in Quartz. To this end, the language features proof goal specification through LTL, CTL and the μ calculus, the latter is a low-level language to which the prior two are compiled.

For instance, taken from the examples delivered with the Averest framework download and available for browsing on the website, Listing 3 specifies constraints for the boundedness of a buffer:

3.1.8 Averest Version

The Averest version used by this thesis is 2.2.4.5, which was not the newest at the beginning of implementation, because of a faulty automated transformation that introduced mistakes into descriptions synthesised with the more recent version.

There is another bug in the Averest framework for this and the more recent versions, namely a mistake in determining the surface of a loop, which leads to problems with the "schizophrenia problem" explained in [17]. This leads to variables declared in a loop context to have values from the previous iteration instead of those assigned in the current one, easily circumvented, at the cost of one additional macro step per iteration, by adding a **`pause;`** statement at the beginning of each loop. Being a recommended way to simplify compilation and synthesis, the workaround also keeps separate variable instances from being needed.

3.2 Concept Car Platform

The ConceptCar [2] is a platform for testing various embedded applications, one of which is being a target for software synthesis through the Averest framework. It was initially developed at the Fraunhofer

[1] this might change in future releases of Averest

```
1  module BoundedBuffer (
2      int ?guess_max_buffer,
3          !rate_f, !rate_g, !rate_h,
4          buffer, !max_buffer,
5      bool guess, state_f, state_g, state_h
6  ) {
7     // Implementation...
8  } satisfies {
9      bounded_buffer : assert E G (buffer <= max_buffer);
10     rates_for_bounded_buffer: assert A ((G (buffer <= →
           max_buffer))
11                                          -> (G F(6*rate_f +
12                                                  3*rate_g +
13                                                  2*rate_h == →
                                                       0)));
14 }
```

Listing 3: BoundedBuffer Proof Goal Example

IESE institute, where the first revisions for both hard- and software were created.

The ConceptCar is an ideal platform for model-based software development and verification, hence it was chosen as an experimental platform for the Averest framework.

3.2.1 Electronic Control Units

It features several Electronic Control Units (ECUs), boards with micro controllers and periphery, which are, with the exception of an emergency board, connected through a common CAN bus for communication.

While all the periphery is connected to and controlled by Atmel AVR AT90CAN128 micro controllers, there is a ”ControlBoard” for more computation-intensive tasks featuring a more powerful 32bit ARM7TDMI processor. It is however not relevant for this thesis because there are existing Quartz applications and hardware abstraction for it.

The Emergency Board features a AVR ATmega88 controller, and provides a reliable tool for the user to shut down the car in cases of system failure.

The detailed operation of each ECU relevant to this thesis is described in Section 4.

3.2.2 Controller Area Network

The Controller Area Network (CAN, [18]) is a two-wire differential bus commonly used in automotive applications.

It features a packet-oriented protocol, in which each packet has a identification (ID) number that is associated with the data being transmitted and does not necessarily indicate the nodes involved. Depending on the version of CAN, the ID can have a length of either 11 or 29 bits.

The bus is multi-mastered, using an arbitration scheme that resolves conflicts without interrupting messages. When transmitting a message on the physical layer, the bits sent are either dominant(”0”)

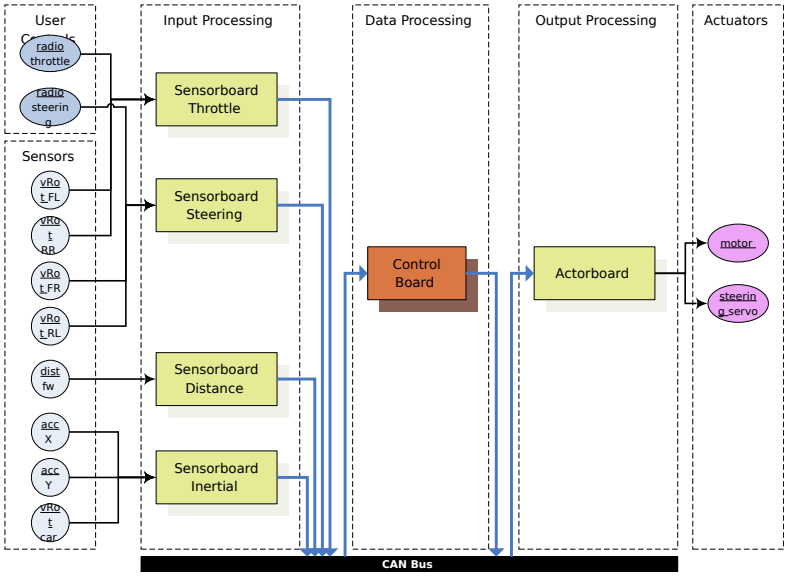

Figure 1: Overview of the ConceptCar Electrical Architecture

or recessive(”1”). In cases where two senders are transmitting different symbols at the same point of time, only the dominant bit is received by the other participants, which allows a node sending a recessive bit to detect when another node sends a dominant bit and retreat from arbitration. This happens without interfering with the transmission from the node who sent the dominant bit, and thus resolves the collision without destroying the message. When multiple nodes start sending a packet at the same point in time, the arbitration is decided by the different message IDs, which are the first to be transmitted after the start-of-frame delimiter. Since the ”0” bit is dominant, this is a prioritisation mechanism, in which packets with a numerically lower ID get a higher priority on the bus.

The ConceptCar architecture uses a very simple application protocol on top of CAN, in which each sensor value, switch state and controller output is assigned a CAN message ID and is transmitted as a 32 bit unsigned integer. Examples for those are: Steering orientation and throttle, wheel speed as time between optical encoder crossings and the setting of the ActorBoard source switch.

Every sensor, actor and the remote control receivers on the ConceptCar are connected through a common bus system on which the sensor readings, received signals and other state information are periodically broadcasted by the different ECUs.

3.2.3 Remote Control and Actor Signalling

Being common components for remote-controlled models that are designed to work without intermediate controllers, the servo motor for steering and motor controller receive settings in the form of Pulse Width Modulated (PWM) signals, which are given by the remote-control wireless receivers. The frequency of the signals is 50 Hz and information is encoded by the length of an impulse, which is described as the duty cycle, being the percentage of the duration of each period in which the signal is in the high state. Valid duty cycles for the ConceptCar remote and actors range between 5% and 10%, which is mapped to the ranges available for steering and motor speed.

To filter and transmit this signal through the CAN bus, the ECUs of the ConceptCar sit in between the remote control receivers and actors and, after measuring the length of incoming pulses, broadcast

these on the CAN bus before they are are reproduced by the ActorBoard. Having an intermediate stage in software and on the CAN bus allows for further processing to take place.

3.3 AVR Micro Controllers

There are two different types of AVR micro controller embedded on the ConceptCar ECUs, the ATmega88 and the AT90CAN128.

Because the ATmega88 is only included in a deprecated hardware revision of the EmergencyBoard and its features are a subset of the AT90CAN128, only the latter will be regarded in this section.

The AVR micro controller family features a 8-bit processor core with a custom instruction set and 32 registers. All connected memory and peripherals are mapped in a uniform 16bit address space. It receives and handles interrupts from various sources, some of which are explained in the following sections. Each controller has an internal flash memory from which the firmware is executed, integrated RAM for temporary and EEPROM for permanent data storage.

3.3.1 General Purpose Input/Output

While pins, depending on the specific controller, are connected to various special functions like the peripherals explained in the following sections, each pin that is not used for power supply or voltage reference is usable as a digital input or output. They are managed in IO ports, groups of 8 pins addressed through a set of common registers.

Being usable as both input and output, pins can be individually switched between a high-impedance state for input and an output state, able to drive loads of up to $20mA$ by setting and clearing a bit in the data direction register of the corresponding port.

3.3.2 External Interrupts

The AT90CAN128 features 8 external interrupt sources, which are pins that trigger processor interrupts on either a low level, all changes in state or rising as well as falling edges on their input.

3.3.3 Timer

There are four timers that count cycles and external in parallel to the normal program execution. Each of those has a counter, two of those are 8 bit, the others 16 bit wide, that are incremented either on external events by configurable internal clock sources. Additionally, prescalers exist that divide the clock by up to 1024 to allow measurement over larger spans of time.

The timers trigger interrupts on counter overflow or user defined values. Each timer incorporates up to three compare registers, allowing to measure periodic intervals and interrupt at specified counter values. These registers are connected to output pins on which they allow the generation of PWM signals.

3.3.4 Controller Area Network

The embedded CAN controller is responsible for the data link and physical layer operation of the CAN bus. The latter requires an external driver to provide sufficient power for signal transmission as well as bus termination.

Interfacing with the software is done through configuration registers as well as a set of 15 ”Message Objects” (MOb), each of which is a buffer that is configured for either the transmission or reception

of CAN messages. For reception, a MOb is assigned both an ID and a mask used to match incoming message IDs with.

Also, if enabled, an interrupt is triggered on reception, completed transmissions and error conditions.

3.3.5 Analogue to Digital Converter

While there is only a single Analogue to Digital Converter (ADC) integrated, there are eight input channels connected to separate pins from which the input to be measured is chosen. Analogue to digital conversions run in parallel to the program execution of the processor and, depending on the configuration, either automatically restart when finished or require the software to start them individually. When finished, the ADC either triggers an interrupt or sets a flag to indicate this to polling software.

The measurements have a 10-bit resolution, which reflects values between either the externally supplied or the internal voltage reference of $2.45V$ and Ground.

3.3.6 Programming

There are two mechanisms for programming integrated in the AT90CAN128. The AVR In-System-Programmer (ISP) is a serial interface that requires a 6-pin connection and is used to upload and download the content of the flash, EEPROM and fuse settings to and from the controller. JTAG [6] is a standard that, besides firmware upload and download like the ISP also features debugging that include access to the processor and internal peripheral state, being able to set software breakpoints on various conditions and reading the content of the RAM as well as the state of the controller pins.

3.3.7 Firmware Development Environment

While Atmel provides a development environment to write firmware for micro controllers of the AVR architecture, all code written in this thesis is targeted at the AVR-GCC compiler suite. This is due to both the previous ConceptCar implementations being written for AVR-GCC and it being available prepackaged for the Linux distributions run on the development computers. The AVR-GCC is bundled with an open-source libC that provides a basis and library for software and the AVR binutils, which provide a linker and tools to, among other functions, analyse the resulting binaries and translate them to a format readable by the programmer.

For programming we chose avrdude, because of the availability on every platform supporting the AVR-GCC, absence of external dependencies and Firmware images are taken as raw binaries or in the Intel hex format and written to the flash and EEPROM using a variety of hardware programmers.

3.4 Quartz Descriptions

The motivation - and basis - of my work are existing Quartz descriptions for the AVR-based ECUs, namely the ActorBoard, EmergencyBoard and the SensorBoards for Acceleration (SensorBoardInertial), Throttle and Steering. Those were a work in progress including a model of the CAN bus to verify the whole system behaviour.

Because of several shortcomings and mistakes of the API and implementations, these descriptions serve only as a rough outline and reference point for the required interfaces.

4 Requirements Analysis & Design Decisions

This section will describe the specific functionality required to produce a working replacement for the firmware controlling the hardware of the ConceptCar, and the challenges involved in the design of a HAL for Quartz modules. Aside from its purpose of documentation, it also serves as a point of reference during design and implementation and to keep track of the tasks that remain to be completed.

First, the existing non-Quartz firmware implementations for the ECUs are described in Section 4.1. The possible mechanisms to receive hardware events are discussed in Section 4.2, followed by a description of the interface decided for drivers required by all or most modules, namely the AVR GPIOs in Section 4.3, CAN bus in Section 4.4 and the timekeeping in Section 4.5. After these, a comprehensive list of the ECUs involved and their requirements as well as design decisions for those is given in Sections 4.6 to 4.11, followed by the real-time constraints and the test case transformation.

4.1 Existing Firmware Implementations

The existing firmware for the ConceptCar ECUs is split into two folders: There is the ”legacy” firmware written in C with no dependencies besides the AVR libC and the new C++ implementations that use FreeRTOS [3] for scheduling as well as Simulink [4] models for higher-level description.

4.1.1 Legacy Barebone C Implementation

The initial implementation of the ECU firmware was done in plain C for the AVR-GCC compiler, which is the same platform the Quartz HAL is developed for. Because most of the libraries of the first implementation have almost no dependencies, they provide an ideal basis and source of drivers for my implementation.

4.1.2 FreeRTOS/Simulink

The legacy C implementation has been declared deprecated and all boards now have firmware that is written in C++ instead of C and features advanced techniques for task-driven and model-based design and implementation.

The FreeRTOS [3] is used for scheduling. It allows the creation of tasks with different priorities as well as periodic time-triggered execution and basic synchronisation primitives like mutexes, queues and semaphores. Also, it contains a software timer framework that allows the execution tasks after specific delays or at regular intervals.

Some of the ECUs are now driven by Simulink models from which C code was generated. These are thoroughly explained in [13], but are more complex and too difficult to translate to be included in the HAL. Also, since Quartz has language-level parallel execution/threading, there is no need for an external scheduler. Thus, it is a requirement for the Quartz HAL that there are neither FreeRTOS dependencies nor Simulink models, although FreeRTOS will be supported by future versions of the Averest framework as described in Section 8.6.

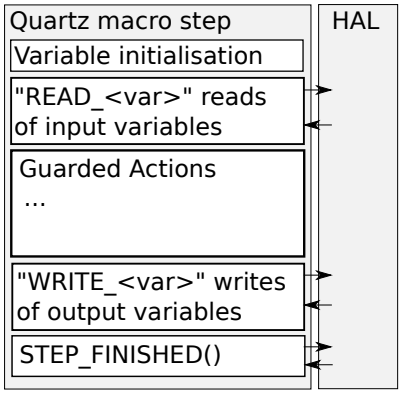

Figure 2: Structure of a single macro step

4.2 Interrupts, Polling and Quartz Macro Steps

There are generally two methods to receive events from hardware: Interrupts and Polling. Interrupts are events generated by hardware that interrupt the flow of regular execution and jump into a previously defined function - called an interrupt service routine. The state of execution of the processor - its instruction pointer, registers[2] and flags are saved and the Interrupt Service Routine (ISR) is called. After it is finished, the previous state is restored and the program continues. While this mechanism ensures that the program flow continues undisturbed after an interrupt occurred, changes made by the ISR need to be synchronised with the main program flow to prevent inconsistencies.

Polling is the term for asking/checking the hardware for new events in regular intervals, usually done in the main loop that polls for all possible events each iteration.

The code produced by the Quartz to C code synthesis favours the usage of polling. Synthesised C consists of a loop, each iteration of which executes one macro step of the corresponding Quartz module. The execution of each macro step is structured as shown in Figure 4.2.

At the beginning of a macro step, all input interface variables are read. Afterwards, the guarded actions are evaluated. The output interface variables are written back, and finally `STEP_FINISHED` is executed.

Using an interrupt to change multiple Quartz interface variables requires synchronisation because reading the input variables is a non-atomic operation. One way to keep the interface variables in a consistent state is to introduce a set of buffer variables that store all changes until the beginning of the next macro step.

4.3 General-Purpose Input & Output

Parts of the functionality required for the HAL can be reduced to digital input and output on controller pins. Examples for these are the Light Emitting Diodes (LEDs) used on all ECUs to indicate state and operation, e.g., flash on CAN message transmission and reception, and reading hardware switch states.

To make the AVR GPIO pins accessible from within Quartz, one boolean interface variable per pin will be introduced per pin and automatically switch the hardware between digital input or output

[2]Saving registers is done by software. AVR-GCC does this when necessary.

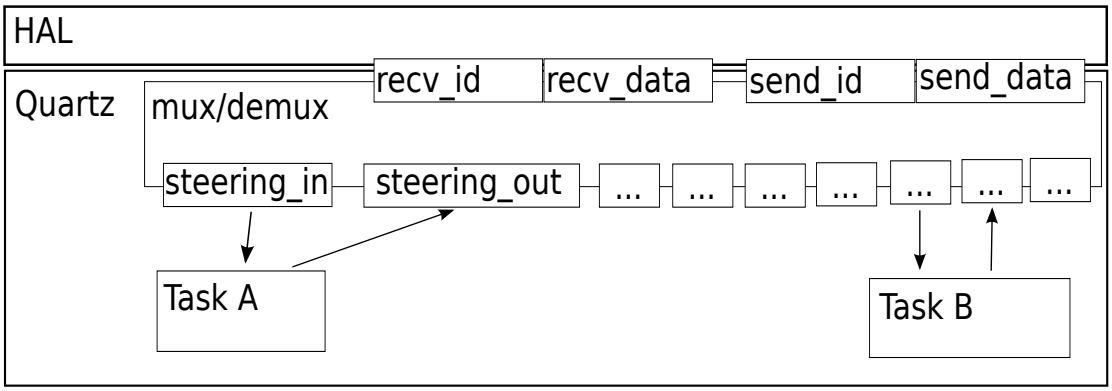

Figure 3: First draft for CAN interface, not implemented

depending on the variable type. Analogous to the naming of the pins given by Atmel, these variables will be named `AVR_PIN<port><number>`, for example `AVR_PINA0`.

4.4 Controller Area Network

Nearly all ECUs of the ConceptCar communicate through a common CAN bus. The only exception is the EmergencyBoard, which is isolated from the other components. As such, being able to send and receive CAN messages is an important requirement for the hardware abstraction layer.

Each CAN message transports one unsigned integer encoded in 4 bytes. The format for those is given in the lower level CAN bus interfaces, they are decoded like this:

```
1 // Read message, data is a uint32_t
2 *(((uint8_t*) &data) + 0) = CANMSG;
3 *(((uint8_t*) &data) + 1) = CANMSG;
4 *(((uint8_t*) &data) + 2) = CANMSG;
5 *(((uint8_t*) &data) + 3) = CANMSG;
```

There have been several interface concepts taken into consideration to work with the CAN controller, being distinguishable mostly by the amount of logic required in the Quartz implementation.

The approach used in previous implementations is to have exactly one interface variable for each CAN message ID that is received. It has the benefit of being the easiest to use, but requires every new message to be added to the C API and makes it difficult for the underlying library to determine which messages the Quartz module requires to be listening to.

There are two other concepts taken into consideration to have each Quartz Module register the CAN message IDs it listens to and receive through the library. Both approaches provide Quartz with arrays each for registering listening IDs, receiving data and status information.

My first consideration was to keep the interface variables as small as possible, and having a Quartz sub-module take care of reading from and writing to FIFO buffers the library provides and multiplex the values passed on those to arrays that allow other components to access received values directly.

To keep the interface simple, still retain the possibility to configure the receiving and sending message IDs from Quartz and keeping the memory footprint as low as possible, I decided to pass the previously mentioned arrays to the C API without prior processing. While a naive implementation would require a lot of memory and processing time, passing pointers and defining proper constraints makes the interface very lightweight.

```
1  module TestBoard{
2     event     bool                  !CAN_RESET,
3           [RECV_SLOTS] nat{NUM_CANID_BITS} !CAN_RCV_IDS,
4           [RECV_SLOTS] nat{NUM_CAN_BITS}   ?CAN_RCV_DATA,
5           [RECV_SLOTS] bool                ?CAN_RCV_RDY,
6           [SEND_SLOTS] nat{NUM_CANID_BITS} !CAN_SEND_IDS,
7           [SEND_SLOTS] nat{NUM_CAN_BITS}   !CAN_SEND_DATA,
8     event [SEND_SLOTS] bool                !CAN_SEND_REQ)
9  {
10    // Implementation
11 }
12 satisfies {
13    // Make sure the Quartz module never changes sending and
14    // receiving IDs without issuing a reset signal.
15    // Used to keep the hardware abstraction layer simple.
16    s1: assert G ((CAN_RCV_IDS != X(CAN_RCV_IDS)) -> →
         X(CAN_RESET))
17    s1: assert G ((CAN_SEND_IDS != X(CAN_SEND_IDS)) -> →
         X(CAN_RESET))
18 }
```

Listing 4: CAN Interface Verification

The first constraint is that the arrays which tell the HAL the CAN IDs for message reception or sending are only allowed to be modified in the same macro step in which a CAN reset is issued. This - together with updating values and executing CAN actions only at the end of each macro step - allows me to use the local variables of the synthesised C code from within my library.

Proof goals specified in the Quartz description allows the first constraint to be verified against, which shows that model-checking can be used for HAL optimisation as well as safety and reliability. The proof goals required for the implemented CAN library are shown in Listing 4.

4.5 Timekeeping

As a generic interface for the hardware timers of the AVR micro controllers would have gone beyond the scope of this thesis and nearly all libraries of the legacy C implementation depend on the `runtime.c` library written by Jonas Mitschang, I decided to use the functions provided by that library.

While the library itself measures time in ”ticks”, which are increased by one every 32 processor cycles, the Quartz interface has access to human-readable units, i. e., milliseconds, seconds, minutes and hours. These are, due to their limited range of values, immune to the integer type problem described in Section 8.1, and allow to track time at a granularity that is optimal for regular-interval tasks required for ConceptCar operation.

4.6 Sensorboards Steering and Throttle

These sensor boards are responsible for receiving the signal of the remote control, each being responsible for the signal components indicated in their name. Precisely measuring the PWM duty cycle is not possible because the execution time required for a macro step is too long. Thus, a low-level driver taken from the previous firmware implementations will provide the Quartz modules with exact timing information.

Also, together, the SensorBoards measure the speed of all 4 wheels by counting impulses from optical encoders, which again is solved in a dedicated driver and made accessible by one interface variable per wheel to alleviate timing constraints.

4.7 ActorBoard

The ActorBoard generates the PWM signals for steering and throttle. It takes the values to output for both from the CAN bus, either the raw measurements from the Sensorboards mentioned previously, or the processed values generated by ARM7 ControlBoard. The signal source is chosen with a hardware switch.

Like the decoding done on the sensor boards Steering and Throttle, generating PWM signals suitable for the actors can not be implemented in Quartz on this platform, but requires a low-level driver that can utilise the precise timing possible with hardware timers and interrupts. Because the actors measure the duration of the pulses at their input, only these have to be handled at a lower level and periodic repetition can be taken care of by the model. Thus, the interface available allows Quartz code to set the duration and trigger a pulse, an example for which is the steering test shown in Section 6.2.

4.8 EmergencyBoard

Being the only ECU without a CAN interface, the EmergencyBoard is responsible for providing galvanic isolation between the actors, namely the motor and steering servo, and the ActorBoard which is responsible for throttle and steering control. It is also able to disconnect those in case of an emergency or ECU malfunction and control the actors to make the car stop as fast as possible to avoid damage. This is done through a separate multiplexer that is switched by a single pin and thus accessible using the GPIO interface from Section 4.3.

Being controlled by a separate SHR-7 remote control, decoding the messages that tell which buttons are pressed is done in software. The legacy firmware features a hand-written state machine, the newer implementations use a Simulink model to describe the protocol and generate a decoder. The reference Quartz description for the emergency board reduces those signals to either ”emergency signal” and ”no emergency signal”, but the HAL Quartz interface will distinguish between individual button presses to allow more fine-grained control.

4.9 SensorBoardInertial

The SensorBoardInertial is the ECU which allows users and developers to test and debug applications both at runtime through a LCD, which is configurable by a set of hardware switches, and do post-hoc analysis through a log file written to an SD card that contains all CAN messages received by the board. The built-in SPI controller is used to connect the integrated microcontroller to a ADIS16006 accelerometer, a ADIS16100 gyroscope and the logging SD card.

System voltage is monitored by an ADC, the driver for which will be written to support measurements on all eight possible input channels of the controller. These will be accessible through the variables `ADC0` to `ADC7`.

4.9.1 LCD

The SensorBoardInertial is equipped with a 2x8 character LCD display to display various sensor values and other system state information. Formatting of the output is done through the format strings of the `printf()` function known from C. Because Quartz has no notion of strings, all of these need to be stored either as constants in the source of the driver or read from the SD card available in the SensorBoardInertial.

The LCD interface for Quartz is an example for the possibility of different levels of abstraction as explained in [16]. The functionality implemented in the native C firmware, allowing the user to switch between different display values, can be implemented both in Quartz and the underlying library.

An interface that leaves the switching between output values to the Quartz module might realised as follows:

```
1  macro FORMAT_NUM = 256;
2  macro VALUE_MAX = exp(2, 16);
3
4  module source_lcdsimple(
5     (nat{FORMAT_NUM} * int{VALUE_MAX}) LCDISPLAY
6     event boot SOME_EVENT) {
7
8     // Display the value 1024 with the formatstring number 1.
9     LCDISPLAY = (1, 1024);
10
11    await(SOME_EVENT);
12
13    // Formatstring number 2 contains event notification
14    LCDISPLAY = (2, 0);
15 }
```

While this approach keeps the interface very simple and efficient, it requires a lot of Quartz code to implement the functionality described above. The more complex LCD interface may look like this:

```
1  // Number of predefined format strings
2  macro LCD_FORMAT_NUM = 256;
3
4  // Maximum value one parameter might take
5  macro LCD_PARAM_MAX = exp(2, 16);
6
7  // Number of different values
8  macro LCD_VALUE_NUM = 3;
9
10 module source_lcdcomplex(
11    nat{LCD_FORMAT_NUM} [LCD_VALUE_NUM] LCD_FORMATS,
12    int{LCD_PARAM_MAX} [LCD_VALUE_NUM] LCD_VALUES,
13    int{LCD_PARAM_MAX} SENSOR1,
14    int{LCD_PARAM_MAX} SENSOR2,
15    int{LCD_PARAM_MAX} SENSOR3 ) {
16
17    // Initial format string choice
18    LCD_FORMATS = [13, 7, 3];
19
20    loop {
21       // Update LCD Values from sensor readings.
22       LCD_VALUES = [SENSOR1, SENSOR2, SENSOR3];
23       pause;
24    }
25 }
```

Discussing the issue with my supervisor lead to the decision that implementing both interfaces provides a flexibility that allows Quartz modules to use the LCD for both indicating events and displaying a variety of sensor and debugging values as well as other aspects of the system state. I choose to call the ”complex” interface ”diagnostic” - because it is most usable for sensor and state monitoring.

4.10 ControlBoard

While all ECUs required for normal operation are equipped with AVR micro controllers, the ControlBoard is included for experimentation with signal processing which requires more computing power. To this end, a board featuring the AT91SAM7A2 processor is employed. It is based on the ARM7TDMI embedded processor and provides a 32bit RISC architecture and 16kBytes of internal SRAM.

This board will not be supported by the HAL developed in this thesis, because the fundamental architecture is different to that of the other ECUs and there are existing Quartz applications for the ARM processor.

4.11 WirelessBoard

In [15], an application for Android smartphones was developed that permits remote debugging and control of the ConceptCar via Bluetooth. To this end, the WirelessBoard was implemented, which translates

between CAN and the Universal Asynchronous Receiver Transmitter (USART), which can be connected to Bluetooth and WiFi modules as well as other serial interfaces.

The Bluetooth module is soldered to a separate board which is connected to the WirelessBoard by a plugged cable that contains the USART transmit and receive lines as well as power supply. Messages are transmitted through the serial connection in packages of up to 16 frames, which correspond to the individual CAN messages with identifier and 4 byte data as defined in Sections 4.4 and 3.2.2.

This board will not be supported by the HAL.

4.12 Real-Time Constraints

While automotive applications have strict real-time constraints, which are especially important in brake- and steer-by-wire scenarios, they are not subject of this thesis. To provide functioning drivers, timing has to be considered for some cases, especially the decoding and generation of PWM signals and their transmission through the CAN bus. These are shown to work by testing instead of a formal WCET analysis.

4.13 ”drivenby” Test Cases

To check the correctness of the Quartz to C synthesis, it is possible to compile drivenby test cases to C source that handles the input and output of a module.

Unfortunately, the compiled tests are - as far as they are working at all, not suitable to be run on a AVR microprocessor and produce useful output. For the version (2.2.4.5) of the Averest framework I am working with, the testbenches `aif2c` produces consist of a (`*_main.c`) C file and the header file, the first contains the boilerplate and input traces for the module, the latter does the actual tests/assertions in `STEP_FINISHED()`.

It is possible to transform those test cases to a firmware for the AVR microcontroller, but transformation on C code would be prone to errors and lead to incompatibilities with future versions of the Averest framework because it has to rely on the current layout of synthesised modules.

Another approach is to use the intermediate AIF files to merge a drivenby test case with the tested module. Since the structure of both sets of guarded actions is the same, the transformation can combine those without further processing except for those containing assertions. These will be rewritten to assignments to a `ASSERTION_ERROR` interface variable that is then used by the HAL to indicate the failed test to the user using an LED.

5 Implementation

In this section, a report on the implementation of various parts of the HAL will be given, both to document the work done, and to provide a reference for future implementations.

The build system required for automatic firmware compilation and linking is described in Sections 5.1 to 5.3 and forms the foundation for the development of drivers, for the implementation of which examples are given in Section 5.5 through 5.7.

Because porting existing drivers into the HAL follows a repeating pattern, the steps involved are described in Section 5.8, which is intended as a guide for future extension of the HAL. Finally, Section 5.9 explains the implementation and function of the transformation used to merge models and testcases.

5.1 Boilerplate

Both initialising the AVR Hardware and starting the Quartz module requires a boilerplate to be written that defines the `main()` function being executed at startup.

Since C compilers allow the definition of macros through command-line parameters, I used one file of C source code which calls a Quartz module with the name given by make.

```
1  #include "avr.h"
2
3  int main(void) {
4    // Initialize AVR library
5    avr_init();
6
7    // Run our Quartz module infinitely
8    QRZ_MODULE(-1);
9
10   return 0;
11 }
```

Listing 5: Quartz Module Boilerplate

Compiling this, for example, with

```
1 gcc -DQRZ_MODULE=lcd_simple -c avr/startup.c \
2  -o examples/lcd_simple.o
```

creates an object file containing the `main()` method that - after initialising the AVR library - starts the synthesised `lcd_simple` module.

For every Quartz Module compiled against my library - this is done automatically by the makefile discussed in the following section.

5.2 Makefile

To make the process of Quartz-to-C synthesis, compiling and linking against the library easy and reliable to use, I chose GNU make.

The dependency-based build system makes it easy to define all steps required for compiling the final AVR firmware.

Each module requiring a HAL library has to have this dependency listed in the makefile. Because the weak symbol library configuration explained in Section 5.3 makes it possible to configure the HAL at link-time, it suffices to specify all non-standard object files to be linked into the firmware as follows:

```
1 examples/adc.elf: avr/drivers/adc.o \
2                   avr/drivers/lcd.o \
3                   avr/drivers/dipswitch.o
```

The makefile rule responsible for the generation of firmware files (`.elf`) then takes care of linking in all specified and required object files.

Possible targets for end users are:

- `make` (default) or `make all`: Builds all firmware images for the AT90CAN128 Controllers.
- `make BOARD=EMERGENCY all`: Builds firmware for the ATmega88 controller used on the EmergencyBoard.

- `make program FLASH=<firmware>`: Takes care of both building and flashing the `<firmware>` image using the avrdude programming tool and a connected USBasp ISP.
- `make backup`: Is responsible for making backups using the same tools as the previously mentioned programming.

  Backups of both the microcontroller flash and EEPROM are placed in the `backups/` subfolder and named with a timestamp.
- `make clean`: This removes all automatically generated code and binaries from the source tree. Because the HAL generates C sources and headers for each Quartz module, all files of this type are deleted using wildcards on the ConceptCar subfolder.
- `make <quartz module>.png` creates an image of the EFSM graph using graphviz, which can be used for debugging.

5.3 Link-Time Library Configuration

Because most of the drivers available require calls to both their setup function and per-macro-step tasks, the HAL needs a mechanism to determine which drivers are used in the current module.

To make the build process as simple as possible, this configuration occurs at link-time, i. e., drivers required by a module only have to be included in linking the firmware. This is achieved by using GCC-specific weak symbols, that allow, at runtime, to check if certain symbols have been defined, and, for example, call the corresponding functions only when they are available. Requiring proper interfacing for each driver, the implementation of this mechanism is further explained in Section 5.8.

5.4 Efficient Array Handling

There are Quartz interfaces that rely on arrays to pass information from and to the library. Because both memory and processing power on the AVR micro controllers are limited, passing those by value would have an impact on the general performance.

A solution, passing only pointers to those arrays to the hardware abstraction layer, requires details of the C synthesis to be taken into consideration. First, it is advisable to access those arrays only from within the `STEP_FINISHED()` macro and not in `READ_<var>` or `WRITE_<var>` - because the order of reads and writes is not defined. Handling non-trivial communication between the library and the synthesised module in-between macro steps guarantees a consistent interface.

Also - it is important to note that while regular interface variables are declared once and the pointers to those can be considered valid for the entire time a module is running, event variables are re-declared in a code block for every macro step and thus are reset and may change location between macro steps. Output event variables are still valid during the `STEP_FINISHED` after they are written, input events have to be set in their `READ<var>()` macros to be usable from within the Quartz environment.

5.5 AVR GPIO Access

The HAL provides an interface to read and write digital values from and to every general-purpose input/output (GPIO) pin available. Each of those is available as a boolean variable from within Quartz. Because the electrical configuration of pins differs between input and output mode, the input mode having a high impedance while the output mode equals a connection to either the supply voltage (high) or ground (low), they have to be configured by setting a specific bit in the data direction register of the

```
1
2  #define PIN_WRITE_DEF(port, number, high) \
3    DDR ## port |= 1 << DD ## port ## number; \
4    if(high) {\
5      PORT ## port |= 1 << P ## port ## number; }\
6    else {\
7      PORT ## port &= ~(1 << P ## port ## number); }
8
9  #define PIN_READ_DEF(var, port, number) \
10     DDR ## port &= ~(1 << DD ## port ## number); \
11     asm volatile("nop"); /*synchronisation*/\
12     var = PIN ## port & (1 << P ## port ## number);
13
14 #define WRITE_AVR_PINA0(output) PIN_WRITE_DEF(A, 0, output)
15 #define READ_AVR_PINA0(var) PIN_READ_DEF(var, A, 0)
```

Listing 6: GPIO Access Macros

corresponding IO port. In C applications, this is done at startup or when the application needs to change the pin mode.

The ConceptCar HAL is not able to determine the usage of variables before execution (topic for future work, see Section 8.3), both because the C synthesis provides no such information besides the calls to the read or write C macros, and because it is impossible to determine which macros are used in another part in the application without executing the corresponding code. Because the setting of the data direction of a pin requires only a single bit set or bit clear operation, it was decided to do this at every time the value of a pin is accessed from Quartz.

Macros were written for every pin available on the AT90CAN128, such that these are now all accessible to Quartz. Access to a pin is identical on all AVR micro controllers, hence the macros written for the AT90CAN128 can be used for any of these without further adaption. The basic implementation is shown in Listing 6, where only the last two lines are written for a specific pin and have to be reproduced.

5.6 Pulse-Width Modulation

Both the decoding of the steering and throttle remote control signal and the generation of signals for the steering servo motor and the motor controller requires timing that is beyond the resolution possible with the architecture of the synthesised C and HAL.

To alleviate this, PWM decoding and generation is done in C and interfaces allow Quartz modules to read and set pulse durations as well as starting pulses. The low-level implementation has been ported from the legacy C implementation, and, together with corresponding CAN message transmission and reception written in Quartz implements the same behaviour shown by the previous firmware.

PWM decoding is done by edge detection through external interrupts and usage of a dedicated timer for precise time measurements. Signal generation uses separate timers for the pulse lengths of steering and throttle.

5.7 Analogue to Digital Converter

Analogue to digital conversions are a good example for tasks being done in parallel to the execution of the Quartz macro steps.

The ADC built into the AT90CAN128 microcontroller is able to choose an input from 8 of the pins, a conversion on which is triggered through access by the HAL. After being started, these conversions take several cycles to complete, but are done entirely by the Analogue to Digital Converter (ADC) and require no further software interaction until the result is available. It is possible to have the ADC run in continuous mode, in which it will start a new conversion every time it is finished, and to trigger an interrupt when a result is available, but the macro step system employed by Quartz favours, as detailed in Section 4.2, the use of polling.

Most of the work is done in the `adc_step()` function, the `READ_ADC<n>` macros are only used to mark required channels and read finished values from the buffers written in `adc_step()`. Every macro step, the HAL checks if the ADC is running, and if it is not busy, copies the result from the hardware buffer into a local buffer that is annotated with the channel number. Afterwards, it chooses the channel for the next conversion and triggers a new conversion.

5.8 Quartz Integration Workflow

Most of the existing libraries for the ConceptCar could be integrated into the hardware abstraction layer by following the same few steps. Thus, the following are a guideline for the implementation of future drivers:

5.8.1 Dependencies and Conflict Avoidance

When working with libraries from different sources or revisions, it is important to make sure that they don't interfere in their usage of the existing hardware - which for AVR means accessing the same registers - or bits therein. One strategy to solve hardware conflicts is to move the required functionality into a new component that serves as an arbiter between the hardware and drivers. Two examples for this are the runtime library that serves as a means for timekeeping in all the ConceptCar drivers, and the SPI driver, which provides a foundation for the SD card interface, the accelerometer and the gyroscope drivers.

Another possibility, applicable to the timers of the AVR micro controllers, is to rewrite one of the conflicting drivers to use another hardware component. This is possible because the AT90CAN128 and other micro controllers have multiple timers that differ only in details like bit width of the counter or number of compare registers.

5.8.2 Initial Setup

Most AVR libraries contain a method used for initialisation, locating these and calling them from the main `avr.c` file is the first step integrating the library. If the added driver is optional, the initialisation and per-macro-step methods should be declared as weak symbols like this to allow link-time configuration:

```
1 extern void spi_init(void) __attribute__ ((weak));
```

The actual call to the initialisation function should then first check if the symbol is defined and call the method if it is like this:

```
1 if(spi_init)
2   spi_init();
```

Obviously, dependencies between drivers must be reflected in the order in which they are initialised.

5.8.3 Interface Variables

The interface variables of the main module are the only way to transfer data between a Quartz description and the HAL libraries.

For clarity's sake, the macros to access these variables are defined in a separate C header file which is stored in `avr/interface/` and included by `avr/interface/qrz.h`, which makes it accessible from every Quartz module.

Depending on the complexity and implementation of the library, `READ_<var>()` or `WRITE_<var>()` can contain a call to a library method, access to a global variable, or even the whole functionality to access hardware. One example for the latter is the input and output library defined in `avr/interface/pins.h`, in which macros to input and output values for each pin is reduced to two bit operations each.

For more complex interfaces, variable synchronisation may be required, which is explained in the following section.

When dealing with event variables, it is important to note that they are declared and reset to their default value at the beginning of the code block for each macro step, which means that they are only valid for the duration of one macro step.

5.8.4 Per-Macro-Step Tasks

The architecture of most embedded systems contains tasks that are executed in regular intervals, which is usually done in either an infinite main loop, through timer interrupts or through tasks called by a scheduler.

In cases where data from and to the Quartz module has to be processed, there is the possibility of buffering them between the calls to `STEP_FINISHED()` and `READ_<var>()` or `WRITE_<var>()`, which allows those to be handled atomically and all at once. In trivial cases, it suffices to use the calls from `READ_<var>()` macros, but when there are dependencies between variables it is best to buffer and handle them all at once.

In those cases, integration into the HAL is done by adding a per-macro-step method to the `avr_step()` function defined in the `avr.c` using weak symbols like the initial setup methods described in Section 5.8.2.

5.9 Drivenby Test Case Transformation

I used Extensible Stylesheet Language Transformation (XSLT) [5] to merge drivenby test cases with the module itself.

XSLT is a turing-complete language that is intended and well-suited to transform from and to XML. Being a standard published by the w3c, it is available as a library for several languages, Saxon being a popular example, and stand-alone tools like `xsltproc`.

The transformation done in the stylesheet may be summarised as follows:

- All interface variables are converted to local variables.
- An new output interface variable `ASSERTION_ERROR` is introduced, the HAL is responsible for indicating this to the user.
- Assertions in the drivenby test case are transformed into assignments to the `ASSERTION_ERROR` interface variable. Each guarded action containing an assertion $(\gamma, assert(\sigma))$ of the drivenby test case is rewritten into $(\gamma \wedge \neg\sigma, ASSERTION\_ERROR = true)$.
- The control- and data-flow sections of the Quartz module and the drivenby block are merged.

The version of the Averest framework I am working with (2.2.4.5) contains two errors because of which it refuses to parse the valid XML produced by xsltproc. Both, not being able to handle the XML header and not expecting self-closing tags for certain elements, are worked around with string replacements.

The entire process of translating a Quartz description to AIF, transforming it with an XSLT stylesheet and the necessary intermediate string replacements are managed by a separate makefile, executed with:

```
1 ||        make testmodule.out.aifs TEST_NAME=testcase1
```

to create an AIFS file in which the module described in `testmodule.qrz` is merged with the drivenby test case named ”testcase1”.

The complete XSLT source code is listed in Appendix A and can be run using all standard-conform XSLT processors.

6 Usage Examples & Testing

This section gives a report on the testing done to assure correct operation of individual components as well as their interaction, giving example code used for testing individual drivers and which show usage of APIs described in previous sections. Also, a verification done through model-checking is described.

6.1 Component Testing

For testing and demonstration, there are various example Quartz descriptions that interface with individual components of the ConceptCar HAL. A selection of these is presented in Section 6.2.

Having implemented the description of the SensorBoardInertial first, the available LCD provided a first, and reliable, means of testing the correct implementation of components as well as various aspects of the synthesis.

After the LCD, the CAN driver was a further valuable means of debugging. Because it was not possible to get the CAN library of the legacy firmware implementation to work, the C++ FreeRTOS Code was ported to the ConceptCar HAL. Once the driver was in a working state, the SensorBoardInertial

could be used to receive sensor readings and state information from connected ECUs, which allowed the drivers and Quartz code of those to be tested without additional hardware.

While proper reading of the switch input state and the subsequent CAN broadcast of the ActorBoard could be confirmed using the InertialBoard LCD, the generated PWM signal was checked by connecting the steering servo to a power supply and running an example program that sweeps the front wheels between the two extreme positions.

Using the `WAIT_MS` module, LEDs that should be flashing in regular intervals were found to ignore the pauses and flash at much higher rates. The occurrence of this could be tracked to submodule calls at the beginning of a loop body, in which case the execution of the first macro step of that module accesses variables of its instance in the previous loop iteration. This is a bug in the Averest framework and can be worked around (see Section 3.1.8).

6.2 Example Source Code

The first, Listing 7, is used to test the correctness and performance of the ADC driver and also provides an example of the diagnostic LCD interface. Both the value measured by the ADC and the number of macro steps needed for conversion are written to the LCD driver, which takes care of handling the hardware switches for output selection.

```
1  module adc(
2     [2] nat{LCD_FORMAT_MAX} !LCD_FORMATS,
3     [2] int{LCD_VALUE_MAX} LCD_VALUES,
4     int{ADV_MAX} ?ADC3
5  ) {
6     int{exp(2,16)} cycles;
7     LCD_FORMATS = [4,  // Voltage
8                    2]; // Steps
9     loop {
10       await(ADC3 != -1);
11       LCD_VALUES[0] = ADC3;
12    } || loop {
13       if(ADC3 == -1) {
14          // Count cycles between measurements
15          next(cycles) = cycles + 1;
16       } else {
17          LCD_VALUES[1] = cycles;
18          next(cycles) = 1;
19       }
20       pause;
21    }
22 }
```

Listing 7: Test Program for ADC

The EmergencyBoard is not connected to the CAN bus and requires other means for testing. Listing 8 shows the SHR77 remote control interface and use of the runtime library as well as a submodule for millisecond delays.

```
1  macro LED_RED = AVR_PINC3;
2  module shr77 (
3     // Bitvector containing button presses
4     event bv{8} ?SHR77_BUTTONS,
5     // Timing input, required for LED flashing
6     nat{TIME_MAX} ?RUNTIME_MILLISECONDS,
7                   ?RUNTIME_SECONDS,
8     // Red LED
9     bool AVR_PINC3,
10 ) {
11    nat{exp(2,8)} flashes;
12    LED_RED = false;
13    loop {
14       // Await button press
15       await(SHR77_BUTTONS != 0x00);
16       // Flash unary encoded button value on LED
17       flashes = bv2nat(SHR77_BUTTONS);
18       while(flashes > 0) {
19          next(flashes) = flashes - 1;
20          pause;
21          LED_RED = true;
22          WAIT_MS(500, RUNTIME_MILLISECONDS, RUNTIME_SECONDS);
23          LED_RED = false;
24          WAIT_MS(500, RUNTIME_MILLISECONDS, RUNTIME_SECONDS);
25       }
26       // Wait for two seconds before accepting
27       // the next button press
28       WAIT_MS(2000, RUNTIME_MILLISECONDS, RUNTIME_SECONDS);
29    }
30 }
```

Listing 8: SHR77 Usage Example

Driving the steering servo and the motor controller is done by issuing a pulse of specific length every 20ms. The test program for the drivers of the ActorBoard is shown in Listing 9 and performs a continuous sweep between the two extreme positions of the front wheels.

```
1  module steering(
2      // PWM Output
3      event (bool * int{ACTOR_MAX}) STEERING_PULSE,
4                                    THROTTLE_PULSE,
5      // Runtime
6      nat{TIME_MAX} ?RUNTIME_MILLISECONDS,
7                    ?RUNTIME_SECONDS
8  ) {
9      int{ACTOR_MAX} steering_test;
10     int{64} steering_add;
11     // Position of steering servo(between -255 and 255)
12     steering_test = -255;
13     // and change per 20ms
14     steering_add = 5;
15     loop {
16        // Wait 20ms before pulse
17        WAIT_MS(20, RUNTIME_MILLISECONDS, RUNTIME_SECONDS);
18        // Emit one PWM Pulse for steering servo
19        STEERING_PULSE = (true, steering_test);
20        // Sweep
21        if((steering_test >= 255 and steering_add > 0) or
22           (steering_test <= -255 and steering_add < 0)) {
23           next(steering_add) = -steering_add;
24        }
25        next(steering_test) = steering_test + steering_add;
26     }
27 }
```

Listing 9: Interface for Steering Servo

The CAN message reception was tested using the SensorBoardInertial using the example source code shown in Listing10, which allowed to display received sensor values and thus confirm correct implementation.

```
1  module can_test (       [6] nat{LCD_FORMAT_MAX} !LCD_FORMATS,
2                          [6] int{LCD_VALUE_MAX}   LCD_VALUES,
3                  event       bool                !CAN_RESET,
4                          [5] nat{NUM_CANID_MAX} !CAN_RCV_IDS,
5                          [5] nat{NUM_CAN_MAX}   ?CAN_RCV_DATA,
6                          [5] bool                ?CAN_RCV_RDY,
7                          [3] nat{exp(2, 11)}     !CAN_SEND_IDS,
8                          [3] nat{NUM_CANID_MAX} !CAN_SEND_DATA,
9                  event   [3] bool                →
                      !CAN_SEND_REQ) {
10
11    int{exp(2,16)} counter;
12    // In the first macro step
13    // the receiving and sending CAN IDs are defined.
14    CAN_RCV_IDS = [CANID_ERROR,
15                   CANID_WHEELSPEED_FL, CANID_WHEELSPEED_FR,
16                   CANID_WHEELSPEED_RL, CANID_WHEELSPEED_RR];
17    emit(CAN_RESET);
18    LCD_FORMATS = [0,  // Hallo Quartz
19                   14, // WHEEL_FL
20                   15, // WHEEL_FR
21                   16, // WHEEL_RL
22                   17]; // WHEEL_RR
23    pause;
24    loop {
25       if(CAN_RCV_RDY[1]) {
26          LCD_VALUES[1] = CAN_RCV_DATA[1];
27       }
28       if(CAN_RCV_RDY[2]) {
29          LCD_VALUES[2] = CAN_RCV_DATA[2];
30       }
31       if(CAN_RCV_RDY[3]) {
32          LCD_VALUES[3] = CAN_RCV_DATA[3];
33       }
34       if(CAN_RCV_RDY[4]) {
35          LCD_VALUES[4] = CAN_RCV_DATA[4];
36       }
37       pause;
38    }
39 }
```

Listing 10: CAN Message Reception Test

A partial implementation for the steering signal reception and CAN broadcast of the SensorBoard-Steering is shown in Listing 11. Sending messages through CAN is analogous to the reception shown in Listing 10

```
1  module SensorBoardSteering(
2     // CAN Interface
3     event         bool                  !CAN_RESET,
4           [1]     nat{NUM_CANID_BITS}   !CAN_SEND_IDS,
5           [1]     nat{NUM_CAN_BITS}     !CAN_SEND_DATA,
6     event [1]     bool                  !CAN_SEND_REQ,
7    // Calculated Steering value from the hardware timer
8     nat{NUM_STEERING_MAX} ?PWMDECODE_PULSE,
9  ){
10    // Configure CAN bus
11    CAN_SEND_IDS = [CANID_STEERING];
12    emit(CAN_RESET);
13    loop { // Broadcast each decoded PWM pulse on CAN
14       await(PWMDECODE_PULSE > 0);
15       CAN_SEND_DATA[0] = PWMDECODE_PULSE;
16       emit(CAN_SEND_REQ[0]);
17    }
18 }
```

Listing 11: Example for CAN Sending and Steering Input

6.2.1 EmergencyBoard

Not being connected to the CAN bus and having another micro controller than the other Boards, testing the EmergencyBoard was done by using the status indicator LEDs. The SHR-77 remote control messages provided by the HAL were encoded unary and output through an LED as human-readable flash sequences through a Quartz module.

The EmergencyBoard also requires switching between the PWM output generated by the ActorBoard for steering and motor control and an emergency PWM signal providing safe values generated on-board. This was tested successfully after the implementation of the ActorBoard.

Having tested the HAL interface for the EmergencyBoard, there are two proof goals shown in Listing 12 that were specified for the Quartz model and verified using SRI's Symbolic Analysis Laboratory (SAL), the procedure for which is documented in [8].

As an example for a complete ECU description, the source code of the EmergencyBoard is given in Appendix B.

6.3 System Testing

The ConceptCar as a whole could not be tested because of hardware failure. Because all ECUs display the same behaviour for the Quartz and legacy firmware, it can be assumed that the problems are not related to the software implementation and will function once the hardware problems are fixed.

```
satisfies {
  // If anything but all three buttons is pressed
  // Hold emergency mode until all three buttons are pressed
  s1: assert G (((SHR77_BUTTON0 or SHR77_BUTTON1 or SHR77_BUTTON2)
                and !(SHR77_BUTTON0 and
                      SHR77_BUTTON1 and
                      SHR77_BUTTON2))
                -> X [(!MUX_SELECT and LED_EMERGENCY)
                      WU (SHR77_BUTTON0 and
                          SHR77_BUTTON1 and
                          SHR77_BUTTON2)]);
  // Inverse case: When all three buttons are pressed,
  // enter the working state until the emergency button press happens.
  // Proof goal only valid after the first macro step because
  // of initial setting
  s2: assert G X((SHR77_BUTTON0 and SHR77_BUTTON1 and SHR77_BUTTON2)
                 -> X [(MUX_SELECT and !LED_EMERGENCY)
                       WU (!(SHR77_BUTTON0 and
                             SHR77_BUTTON1 and
                             SHR77_BUTTON2)
                           and (SHR77_BUTTON0 or
                                SHR77_BUTTON1 or
                                SHR77_BUTTON2)
                           )
                       ]);
}
```

Listing 12: EmergencyBoard Proof Goals

6.4 Test Case Transformation

Checking the correctness of the test case transformation is possible through both the simulation of the resulting AIFS descriptions and by manual inspection of these using the aif2txt tool that gives a human-readable list of guarded actions.

There are test cases included in the ConceptCar code repository that test for known problems with either the C synthesis or the Quartz compiler.

7 Summary & Conclusion

The HAL defined in this thesis allows models written in the Quartz language to interface the hardware available on the ConceptCar. It provides drivers for specific hardware as well as the AVR platform in general, and thus can be used for other embedded systems featuring that micro controller platform.

Documentation is provided that gives an overview of the Averest framework, the ConceptCar and the AVR micro controllers involved, as well as the requirements necessary for developing firmware able to control the ECUs. The basic architecture and details of the implementation have been described, and thus this work provides a point of reference for future development on the ConceptCar and C synthesis.

To support fixing the problems discovered in the Averest framework, these have been described in detail.

The test case transformation developed is, because working on the intermediate format, applicable to all synthesis targets and not limited to the generation of C code.

In conclusion, the C synthesis - in its current state - should not be used to develop applications that rely on the strict typing provided by the Quartz language and the Averest framework or require interface variables larger than the platform-dependent `int` and `unsigned int` types. While the usage of AVR pin input and output, the analogue-to-digital converter and other interfaces whose data values fit into the 16 bit integer types provided by synthesis is safe as long as those bounds are kept in mind, interfaces and computations with larger data values should be considered unsafe or just broken.

This, however, is the only problem with the C synthesis I could find. Apart from the typing issue, the generated code appears to be correct and - while probably not optimal - works perfectly in every example I built. The use of macros to read and write Quartz module interface variables from C works quite well and allows the library to be optimised more thoroughly than simple function calls would allow, as shown by the array access implementation in Section 5.4.

While the C code and parts of the build process are highly platform-dependent, the HAL built in this thesis can be used as a reference if not even basis for future embedded platforms on which Quartz applications are run.

8 Future Work

During the development of the HAL, the C synthesis was found to produce code that is considered incorrect, which is explained in Section 8.1.

Apart from this, an optimisation useful for saving energy on embedded systems (Section 8.5) and the work required for future Quartz versions (Section 8.6), the future work includes ideas to make the development of applications and lower-level drivers more comfortable.

8.1 C Synthesis

The main problem for the proper application of the Quartz to C synthesis is the lack of support for different data types. With the version of the Averest framework I use, the only two data types supported are **`int`** and **`unsigned int`**. Because Quartz relies on strict data types for both simulation/testing and verification, having all interface and local variables be implemented as **`int`**, **`unsigned int`** or structs and arrays of those changes the semantics of synthesised Quartz modules. Obviously, these changes in semantics have to be regarded as a fault in the synthesis process.

To fix this for natural and integer variables, a first approach would be the use of types defined in the `stdint.h` header given in the C standard library. The C99 [9] standard which provides numeric types of at least 64 bits. To avoid producing semantically incorrect code, I recommend aborting the synthesis when these are not sufficient.

Because the HAL accesses data of the generated C code by pointer, changing the types produced by the C synthesis requires at least those to be adjusted.

There are two changes to the way the synthesis works that I imagine being interesting and useful: Making the generated C code reentrant would be accomplished by moving all variables, i. e., the state of the module, which is declared as local in the module functions now, into a struct that is passed to a function which executes a single macro step. A small usage example is shown in Listing 13. This allows multiple modules to be executed in a pseudo-parallel fashion with interleaved macro steps and might make parts of the state struct a viable interface to other components. The latter would require the definition of a common layout for the data structures. Reentrant modules might be a requirement to answer a question proposed by my supervisor: Can Quartz modules be compiled to objects files, which are connected to each other at link-time?

8.2 Quartz Interfaces

A problem with the Quartz descriptions I wrote for this thesis was the amount of redundant code required because all macros and interface variables for a library interface have to be included in every Quartz module accessing it. This makes writing and maintaining modules more time-consuming and error-prone than it would be with, for example, Java-like interface declarations.

While the file includes possible in future versions will alleviate this problem, I can imagine a possible implementation of interfaces to look like the one sketched in Listings 14 and 15. At compile-time, those two definitions could, through regular text replacement, be merged into a module with the usual syntax as shown in Listing 16.

Generic interfaces would allow implementations of components in larger applications to be swapped without having to copy or move the interface declaration or specification.

```
#include "ReModule.h"
#include "avr/avr.h"

int main(void) {
  ReModule_state state;
  ReModule_init(&state);

  avr_init(&state);


  while(!state->terminated) {
    ReModule_step(&state);
    avr_step(&state);
  }
}
```

Listing 13: Reentrant Synthesised Module

```
interface CAN_TRANSCEIVER<N,M> (
   event     bool                !CAN_RESET,
         [N] nat{NUM_CANID_BITS} !CAN_RCV_IDS,
         [N] nat{NUM_CAN_BITS}   ?CAN_RCV_DATA,
         [N] bool                ?CAN_RCV_RDY,
         [M] nat{NUM_CANID_BITS} !CAN_SEND_IDS,
         [M] nat{NUM_CAN_BITS}   !CAN_SEND_DATA,
   event [M] bool                !CAN_SEND_REQ)

satisfies {
   // Make sure the Quartz module never changes sending and
   // receiving IDs without issuing a reset signal.
   // Used to keep the hardware abstraction layer simple.
   s1: assert G ((CAN_RCV_IDS != X(CAN_RCV_IDS)) -> →
      X(CAN_RESET))
   s1: assert G ((CAN_SEND_IDS != X(CAN_SEND_IDS)) -> →
      X(CAN_RESET))
}
```

Listing 14: Quartz Interface Definition

```
1  module TestBoard implements CAN_TRANSCEIVER<2,2> {
2     // Configure CAN message IDs.
3     CAN_RCV_IDS = [1, 2];
4     CAN_SEND_IDS = [3, 4];
5     emit(CAN_RESET);
6
7     loop {
8        // Forward message with ID 1 as ID 3 with the same data
9        await(CAN_RCV_RDY[0]);
10       CAN_SEND_DATA[0] = CAN_RCV_DATA[0];
11       emit(CAN_SEND_REQ[0])
12    } || loop {
13       // Second slot, forward message with ID 2...
14       await(CAN_RCV_RDY[1]);
15       CAN_SEND_DATA[1] = CAN_RCV_DATA[1];
16       emit(CAN_SEND_REQ[1])
17    }
18 }
```

Listing 15: Quartz Interface Usage

8.3 Automatic Library Configuration

A feature that would make developing modules for the ConceptCar more comfortable is automatic configuration of the required libraries. The current implementation requires modules with dependencies other than the default libraries, for example the CAN or the LCD library, to have those manually written into the makefile. Detecting dependencies is a trivial task, because each interface variable used can be associated with the library it is provided by with a simple look-up table.

8.4 Portability

Large parts of the C code developed for the HAL are portable to other AVR micro controllers with little or no modification. Also, the Quartz interface is developed to be as versatile as possible while keeping the complexity of the C implementation within limits.

By changing the build process to output target-specific code, object files and executables to corresponding build directories and some small modifications of the C code involved, the HAL could produce firmware for a wide range of AVR micro controllers.

While the C code is not portable to other micro controllers like the ARM7TDMI employed on the ControlBoard, the Quartz interface itself is, and can be used to build platform-independent model descriptions once drivers for the ARM architecture are written.

```
1  macro N = 2;
2  macro M = 2;
3
4  module TestBoard{
5     event     bool                 !CAN_RESET,
6           [N] nat{NUM_CANID_BITS} !CAN_RCV_IDS,
7           [N] nat{NUM_CAN_BITS}   ?CAN_RCV_DATA,
8           [N] bool                ?CAN_RCV_RDY,
9           [M] nat{NUM_CANID_BITS} !CAN_SEND_IDS,
10          [M] nat{NUM_CAN_BITS}   !CAN_SEND_DATA,
11    event [M] bool                !CAN_SEND_REQ)
12 {
13    // Configure CAN message IDs.
14    CAN_RCV_IDS = [1, 2];
15    CAN_SEND_IDS = [3, 4];
16    emit(CAN_RESET);
17
18    loop {
19       // Forward message with ID 1 as ID 3 with the same data
20       await(CAN_RCV_RDY[0]);
21       CAN_SEND_DATA[0] = CAN_RCV_DATA[0];
22       emit(CAN_SEND_REQ[0])
23    } || loop {
24       // Second slot, forward message with ID 2...
25       await(CAN_RCV_RDY[1]);
26       CAN_SEND_DATA[1] = CAN_RCV_DATA[1];
27       emit(CAN_SEND_REQ[1])
28    }
29 }
30 satisfies {
31    // Make sure the Quartz module never changes sending and
32    // receiving IDs without issuing a reset signal.
33    // Used to keep the hardware abstraction layer simple.
34    s1: assert G ((CAN_RCV_IDS != X(CAN_RCV_IDS)) -> →
         X(CAN_RESET))
35    s1: assert G ((CAN_SEND_IDS != X(CAN_SEND_IDS)) -> →
         X(CAN_RESET))
36 }
```

Listing 16: Quartz Interface Merged

8.5 Sleep

For many embedded systems, energy efficiency is an important consideration, for both hardware and software design. Most micro controllers used in embedded applications allow the firmware to switch the processor and other components to sleep modes in which its execution is stopped and energy consumption is usually reduced to a fraction of what it takes under load. Depending on the application, Quartz modules might be made to execute their macro steps in larger intervals between which the processor is halted, or run at a lower clock speed, which is another way to conserve energy when applicable.

It might be possible for a Quartz transformation and libraries to be combined in a way that halts the processor whenever a Quartz module ”waits” for external events without changing the internal state.

8.6 Operating System Abstraction Layer

With the ongoing research on desynchronisation of synchronous programs, the Averest framework will, in coming versions, support the generation of asynchronous code.

To retain platform-independence, the C synthesis will require a portable API that has access to basic multithreading primitives, such as the creation of threads, semaphores, mutexes and queues.

Being a common problem of multithreaded applications that have to be supported for heterogenous platforms, there have been previous publications on the topic of Operating System Abstraction Layers (OSALs), [14], for example, describing the implementation for wireless sensor networks.

[11], as an example for platform-dependent code generation, shows the problem of interfacing the FreeRTOS API for multithreaded applications and appears to be worth a closer examination for the usability as an interface for Averest-synthesised C code.

9 Bibliography

References

[1] *The Averest System*. Available at `http://www.averest.org/`.

[2] *Concept Car*. Available at `http://es.cs.uni-kl.de/research/applications/concept-car/`.

[3] *FreeRTOS - Market leading RTOS*. Available at `http://www.freertos.org/`.

[4] *Simulink: Simulation and Model-Based Design*. Available at `http://www.mathworks.de/products/simulink/index.html`.

[5] *XSL Transformation (XSLT) Version 1.0*. Available at `http://www.w3.org/TR/xslt`.

[6] (2001): *IEEE Standard Test Access Port and Boundary Scan Architecture*. *IEEE Std 1149.1-2001* , pp. 1–212doi:10.1109/IEEESTD.2001.92950.

[7] Y. Bai, J. Brandt & K. Schneider (2011): *Data-Flow Analysis of Extended Finite State Machines*. In B. Caillaud, J. Carmona & K. Hiraishi, editors: *Application of Concurrency to System Design (ACSD)*, IEEE Computer Society, Newcastle Upon Tyne, England, UK, pp. 163–172.

[8] M. Gesell, F. Bichued & K. Schneider (2014): *Using Different Representations of Synchronous Systems in SAL*. In J. Ruf & D. Allmendinger, editors: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, BÃ¶blingen, Germany.

[9] ISO (1999): *ISO C Standard 1999*. Technical Report. Available at `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf`. ISO/IEC 9899:1999 draft.

[10] P. Iyenghar, E. Pulvermueller, C. Westerkamp & J. Wuebbelmann (2011): *Integrated model-based approach and test framework for embedded systems*. In: *Forum on Specification and Design Languages (FDL)*, IEEE Computer Society, Oldenburg, Germany, pp. 1–8.

[11] B. Kim, L.T.X. Phan, O. Sokolsky & I. Lee (2013): *Platform-dependent code generation for embedded real-time software*. In: *Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, IEEE Computer Society, Montreal, Québec, Canada, pp. 1–10.

[12] G. Logothetis (2004): *Specification, Modelling, Verification and Runtime Analysis of Real Time Systems*. *DISKI (Dissertationen zur Künstlichen Intelligenz)* 280, IOS Press. ISBN 1-58603-413-8, ISBN 3-89838-280-X.

[13] Jonas Mitschang (2009): *Evaluation of a Model-Based Development Process for Automotive Embedded Systems - Model-Based Development of an Adaptive Cruise Control System*. Master's thesis.

[14] Ramon Serna Oliver, Ivan Shcherbakov & Gerhard Fohler (2010): *An Operating System Abstraction Layer for Portable Applications in Wireless Sensor Networks*. In: *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, ACM, New York, NY, USA, pp. 742–748, doi:10.1145/1774088.1774243. Available at `http://doi.acm.org/10.1145/1774088.1774243`.

[15] O. Rafique (2013): *Design, Development, and Integration of a Wireless Communication Unit in ConceptCar*. Master's thesis, Department of Computer Science, University of Kaiserslautern, Germany.

[16] O. Rafique, M. Gesell & K. Schneider (2013): *Targeting Different Abstraction Layers by Model-Based Design Methods for Embedded Systems: A Case Study*. In: *Real-Time Computing Systems and Applications (RTCSA)*, IEEE Computer Society, Taipei, Taiwan.

[17] K. Schneider (2009): *The Synchronous Programming Language Quartz*. Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany.

[18] ISO Standard (1993): *ISO 11898, 1993. Road vehicles–interchange of digital information–Controller Area Network (CAN) for high-speed communication* .

[19] J. Tsay, C. Hylands & E. Lee (2000): *A code generation framework for Java component-based designs*. In: *Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, ACM, San Jose, California, USA, pp. 18–25.

[20] Marcel Zimmer (2009): *Prototypische Implementierung und Evaluation von Sicherheitsmustern in eingebetteten Systemen*. Master's thesis.

Appendices

A Test Case Transformation

This is the XSLT code used for the drivenby test case transformation. It can be run by standard-conform XSLT processors which are available for various languages and exist as independent applications. For development, and in the makefile, the xsltproc command-line tool was used.

Listing 17: XSLT Source Code for Transformation

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <!-- AIF parsers choke on xml header -->
  <xsl:output method="xml" omit-xml-declaration="yes"/>
  <xsl:template match="/system">
    <xsl:variable name="testcase" select="drivers/systempart[@name=$TEST_NAME]" />
    <system name="{@name}">
      <interface>
        <output storage="memorized">
          <basic name="ASSERTION_ERROR" /><qBool />
        </output>
      </interface>
      <locals>
        <!-- Transform interface into local variables -->
        <xsl:for-each select="interface/*">
          <locvar storage="memorized">
            <xsl:copy-of select="./*" />
          </locvar>
        </xsl:for-each>
        <xsl:copy-of select="locals/*" />
        <xsl:copy-of select="$testcase/locals/*" />
      </locals>
      <abbreviations>
        <xsl:copy-of select="abbreviations/*" />
        <xsl:copy-of select="$testcase/abbreviations/*" />
      </abbreviations>
      <init>
        <ctrlFlow>
          <xsl:copy-of select="init/ctrlFlow/*" />
          <xsl:copy-of select="$testcase/init/ctrlFlow/*" />
        </ctrlFlow>
        <dataFlow>
          <xsl:copy-of select="init/dataFlow/*" />
          <xsl:copy-of select="$testcase/init/dataFlow/*" />
          <!-- Transform all assertions to dataflow assignments -->
          <xsl:for-each select="init/asserts/if">
            <xsl:call-template name="if-transform" />
          </xsl:for-each>
          <xsl:for-each select="$testcase/init/asserts/if">
            <xsl:call-template name="if-transform" />
          </xsl:for-each>
        </dataFlow>
        <asserts>
          <xsl:comment> Asserts are integrated into the data flow </xsl:comment>
        </asserts>
      </init>
      <main>
        <ctrlFlow>
```

```
49          <xsl:copy-of select="main/ctrlFlow/*" />
50          <xsl:copy-of select="$testcase/main/ctrlFlow/*" />
51        </ctrlFlow>
52        <dataFlow>
53          <xsl:copy-of select="main/dataFlow/*" />
54          <xsl:copy-of select="$testcase/main/dataFlow/*" />
55          <!-- Transform all assertions to dataflow assignments -->
56          <xsl:for-each select="main/asserts/if">
57            <xsl:call-template name="if-transform" />
58          </xsl:for-each>
59          <xsl:for-each select="$testcase/main/asserts/if">
60            <xsl:call-template name="if-transform" />
61          </xsl:for-each>
62        </dataFlow>
63        <asserts>
64          <!--<xsl:comment> Asserts are integrated into the data flow </xsl:comment>-->
65        </asserts>
66      </main>
67      <absReacts>
68        <xsl:copy-of select="absReacts/*" />
69        <xsl:copy-of select="$testcase/absReacts/*" />
70      </absReacts>
71      <invar>
72        <conj>
73          <xsl:copy-of select="invar/*" />
74          <xsl:copy-of select="$testcase/invar/*" />
75        </conj>
76      </invar>
77      <xsl:copy-of select="specifications" />
78      <drivers>
79        <xsl:copy-of select="drivers/systempart[@name!=$TEST_NAME]" />
80      </drivers>
81    </system>
82  </xsl:template>
83  <!-- Translate assertions to guarded assignments on error variable -->
84  <xsl:template name="if-transform">
85    <xsl:variable name="assertion" select="following-sibling::*[1]" />
86    <!-- If the guard and condition of the assertion are true... -->
87    <if>
88      <conj>
89        <xsl:copy-of select="*" />
90        <neg>
91          <xsl:copy-of select="$assertion/*[1]/following-sibling::*" />
92        </neg>
93      </conj>
94    </if>
95    <!-- ...set the ASSERTION_ERROR to true -->
96    <assign type="now">
97      <qBool />
98      <var>
99        <basic name="ASSERTION_ERROR" />
100       <qBool />
101     </var>
102     <boolExpr><boolConst val="true" /></boolExpr>
103   </assign>
104 </xsl:template>
```

```
105 ||   <!-- identity template -->
106 ||   <xsl:template match="@*|node()">
107 ||     <xsl:copy>
108 ||       <xsl:apply-templates select="@*|node()"/>
109 ||     </xsl:copy>
110 ||   </xsl:template>
111 || </xsl:stylesheet>
```

Listing 17: XSLT Source Code for Transformation

B EmergencyBoard

The entire Quartz description for the EmergencyBoard as it is employed now:

```
1  package conceptcar.Boards;
2  import conceptcar.Boards.core.*;
3
4  module EmergencyBoard (bool ?SHR77_BUTTON0,
5                         bool ?SHR77_BUTTON1,
6                         bool ?SHR77_BUTTON2,
7                         bool LED_EMERGENCY,
8                         bool MUX_SELECT){
9     //***keep track of Emergency signal from the SHR-7 transmitter***
10    loop
11    {
12       // Initial state: STOP/BRAKE
13       ENABLE_LED(LED_EMERGENCY);
14       DISABLE_MUX(MUX_SELECT);
15
16       // Wait for three buttons, then start
17       await(SHR77_BUTTON0 and SHR77_BUTTON1 and SHR77_BUTTON2);
18
19       DISABLE_LED(LED_EMERGENCY);
20       ENABLE_MUX(MUX_SELECT);
21
22       // If anything but all three buttons is pressed on the remote:
23       // Back to initial stop state.
24       await((SHR77_BUTTON0 or SHR77_BUTTON1 or SHR77_BUTTON2)
25             and !(SHR77_BUTTON0 and SHR77_BUTTON1 and →
                    SHR77_BUTTON2));
26    }
27 }
28 satisfies {
29    // If anything but all three buttons is pressed
30    // Hold emergency mode until all three buttons are pressed
31    s1: assert G (((SHR77_BUTTON0 or SHR77_BUTTON1 or SHR77_BUTTON2)
32                  and !(SHR77_BUTTON0 and SHR77_BUTTON1 and →
                        SHR77_BUTTON2))
33                  -> X [(!MUX_SELECT and LED_EMERGENCY)
34                        WU (SHR77_BUTTON0 and SHR77_BUTTON1 and →
                              SHR77_BUTTON2)]);
35
36    // Inverse case: When all three buttons are pressed,
37    // enter the working state until the emergency button press →
         happens.
38    // Proof goal only valid after the first macro step because of →
         initial setting
39    s2: assert G X((SHR77_BUTTON0 and SHR77_BUTTON1 and →
         SHR77_BUTTON2)
40                   -> X [(MUX_SELECT and !LED_EMERGENCY)
41                         WU (!(SHR77_BUTTON0 and SHR77_BUTTON1 and →
                               SHR77_BUTTON2)
```

```
42                          and (SHR77_BUTTON0 or SHR77_BUTTON1 or →
                                SHR77_BUTTON2)
43                          )
44                      ]);
45  }
```

Listing 18: EmergencyBoard

C ABRO

An example for the C code synthesis, Listing 20 is generated from Listing 19 using the Averest framework.

```
module ABRO(event ?a,?b,?r,!o) {
    loop
    abort {
    await(a); || await(b);
    emit(o);
    await(r);
    } when(r);
}
```

Listing 19: ABRO Example Quartz Source

Listing 20: ABRO Example C Source

```
#include "ABRO.h"
#include "ABRO_io.h"
void ABRO(int __steps)
{
    /* declare variables */
    int a = 0;
    int b = 0;
    int r = 0;
    unsigned int __state000 = 0;
    /* initial state */
    {
        /* declare local event variables */
        int o = 0;
        int __lvar008 = 0;
        int __lvar021 = 0;
        /* read inputs */
        READ_a(a);
        READ_b(b);
        READ_r(r);
        /* evaluate guarded actions */
        __lvar008 = (((__state000 == 1)
                    || (__state000 == 4))
                   || ((__state000 == 1)
                       || (__state000 == 3)));
        __lvar021 = (__lvar008 && (__state000 == 2));
        __state000 = 1;
        /* write outputs */
        WRITE_o(o);
        STEP_FINISHED();
    }
    /* main loop */
    while(1)
    {
        switch(__state000)
        {
            case 1:
            {
```

```
38                  /* declare local event variables */
39                  int o = 0;
40                  int __lvar008 = 0;
41                  int __lvar021 = 0;
42                  /* read inputs */
43                  READ_a(a);
44                  READ_b(b);
45                  READ_r(r);
46                  /* evaluate guarded actions */
47                  if(((! r) && (a && b))) {
48                      o = 1;
49                  }
50                  __lvar008 = (((__state000 == 1)
51                                || (__state000 == 4))
52                               || ((__state000 == 1)
53                                   || (__state000 == 3)));
54                  __lvar021 = (__lvar008 && (__state000 == 2));
55                  if(r) {
56                      __state000 = 1;
57                  }
58                  if(((! r) && (a && b))) {
59                      __state000 = 2;
60                  }
61                  if(((! r) && (a && (! b)))) {
62                      __state000 = 3;
63                  }
64                  if(((! r) && (b && (! a)))) {
65                      __state000 = 4;
66                  }
67                  if(((! r) && ((! a) && (! b)))) {
68                      __state000 = 1;
69                  }
70                  /* write outputs */
71                  WRITE_o(o);
72
73                  STEP_FINISHED();
74              }
75              break;
76
77              case 2:
78              {
79                  /* declare local event variables */
80                  int o = 0;
81                  int __lvar008 = 0;
82                  int __lvar021 = 0;
83                  /* read inputs */
84                  READ_a(a);
85                  READ_b(b);
86                  READ_r(r);
87                  /* evaluate guarded actions */
88                  __lvar008 = (((__state000 == 1)
89                                || (__state000 == 4))
90                               || ((__state000 == 1)
91                                   || (__state000 == 3)));
92                  __lvar021 = (__lvar008 && (__state000 == 2));
93                  if(r) {
```

```
94                      __state000 = 1;
95                  }
96                  if((! r)) {
97                      __state000 = 2;
98                  }
99                  /* write outputs */
100                 WRITE_o(o);
101                 STEP_FINISHED();
102             }
103             break;
104             case 3:
105             {
106                 /* declare local event variables */
107                 int o = 0;
108                 int __lvar008 = 0;
109                 int __lvar021 = 0;
110                 /* read inputs */
111                 READ_a(a);
112                 READ_b(b);
113                 READ_r(r);
114                 /* evaluate guarded actions */
115                 if((b && (! r))) {
116                     o = 1;
117                 }
118                 __lvar008 = (((__state000 == 1)
119                               || (__state000 == 4))
120                              || ((__state000 == 1)
121                                  || (__state000 == 3)));
122                 __lvar021 = (__lvar008 && (__state000 == 2));
123                 if(r) {
124                     __state000 = 1;
125                 }
126                 if((b && (! r))) {
127                     __state000 = 2;
128                 }
129                 if(((! b) && (! r))) {
130                     __state000 = 3;
131                 }
132                 /* write outputs */
133                 WRITE_o(o);
134                 STEP_FINISHED();
135             }
136             break;
137
138             case 4:
139             {
140                 /* declare local event variables */
141                 int o = 0;
142                 int __lvar008 = 0;
143                 int __lvar021 = 0;
144                 /* read inputs */
145                 READ_a(a);
146                 READ_b(b);
147                 READ_r(r);
148                 /* evaluate guarded actions */
149                 if((a && (! r))) {
```

```
150 ||                      o = 1;
151 ||                  }
152 ||                  __lvar008 = (((__state000 == 1)
153 ||                                || (__state000 == 4))
154 ||                               || ((__state000 == 1)
155 ||                                   || (__state000 == 3)));
156 ||                  __lvar021 = (__lvar008 && (__state000 == 2));
157 ||                  if(r) {
158 ||                      __state000 = 1;
159 ||                  }
160 ||                  if((a && (! r))) {
161 ||                      __state000 = 2;
162 ||                  }
163 ||                  if(((! a) && (! r))) {
164 ||                      __state000 = 4;
165 ||                  }
166 ||                  /* write outputs */
167 ||                  WRITE_o(o);
168 ||                  STEP_FINISHED();
169 ||              }
170 ||              break;
171 ||          }
172 ||          /* check for stopping */
173 ||          if(__steps > 0) __steps --;
174 ||          if(__steps == 0) return;
175 ||      }
176 || }
```

Listing 20: ABRO Example C Source