Fraunhofer Institut Experimentelles Software Engineering

Developing µController-Systems with UML
A MARMOT Case Study

Author:
Christian Bunse

IESE-Report No. 111.06/E
Version 1.0
September 18, 2006

A publication by Fraunhofer IESE

Abstract

According to its proponents, model-based and component-oriented software development has the capacity to compete successfully, and perhaps in many cases displace, traditional commercial development methods even for embedded system development. In order to investigate these claims, the development of a small embedded system (i.e., control of an exterior mirror) using the MARMOT development approach is presented. To evaluate the promised ease of reuse by following a component-oriented approach the components of the mirror system are used in the context of different projects. Thereby, several aspects of reuse, application size, adaptation, and development effort are quantified. This analysis reveals that model-based and component-oriented development performs well for small embedded systems. This allows the conclusion that applying model-based development for small embedded systems lead to adaptable systems and a higher-than-normal reuse rate.

Keywords: Embedded Systems, UML. Case Study, MARMOT

Table of Contents

1 Introduction

One of the most important motivations for the application of object technology and subsequently component-based software engineering techniques in practice is that new applications can be created with significantly less effort than in traditional approaches, simply by assembling the appropriate prefabricated parts. However, contemporary object and component technologies are still some way from realizing the vision of rapid application assembly, in particular when embedded system development is considered. Even today, a high number of embedded system development projects are targeted at small systems with 8/16 bit processors and limited memory resources. However, the complexity of these systems is continuously increasing. Thus, engineers used to applying informal but structured development techniques are now resorting to investigate the use of object-oriented development methods, component technology, and, as a unifying link, the application of UML [25] as the emerging standard notation in software development. The underlying hypothesis is that the use of such techniques will help to control the complexity of embedded systems more easily, to improve maintainability, adaptability and portability, as well as time-to-market [11].

Component-based development and reuse are as attractive in the embedded system domain as they are in other software domains, and they may be considered the single most important basis technologies to appease the ever increasing demand in new and more complex systems. Component-based development methods, technologies, and tools have come along way in the past years to meet the increasing demand of most con-temporary information systems. However, in the embedded domain, the impact that component technology, in particular, and software engineering methods, in general, could have, is not readily exploited for an apparent reason: The disciplines that are dealing with embedded system development, mechanical-, electronic-, and software engineering, are not in sync. Moreover, most embedded systems are implemented in ways leading to the conclusion that the last ten years of advances in the field of software engineering were entirely meaningless for the embedded world. This situation cannot really be attributed to one of these fields alone. As a matter of fact, engineers are struggling hard to master the pitfalls of modern, complex embedded systems, but they only approach the problems from their individual perspectives. What is really lacking in embedded system development is a vehicle to transport the recent advances in software engineering and component technologies into the embedded world in a way that engineers of the three disciplines can actually communicate and understand each other.

This report introduces and describes a new system development method, known as MARMOT, intended to provide all the ingredients to master the multi-disciplinary effort of developing component-based embedded systems. A development method provides templates, models and guidelines for the artifacts describing a (software) system, the product model, and how these how these artifacts are related throughout the development life-cycle, the process model. Furthermore, the paper presents a case-study on applying MARMOT to the development, adaptation and reuse of components in the context of small embedded systems. In specific, a control system for an exterior car mirror is developed, ported to other target platforms, adapted concerning changed functionality, and reused within a larger project. To validate expected benefits concerning reuse, time-to-market, adaptability, etc. several aspects are quantified and analyzed such as number and size of models, amount of reuse, defect numbers, etc.

The remainder of the report is structured as follows: In Section 2, a overview on related work is presented, while Section 3 describes the research methodology as well as the relevant research questions, followed by a description of the MARMOT development process in Section 4. Section 5 presents the case study in more detail including example UML models, and Section 6 presents the evaluation of the case study and describes its quantitative results. Finally, Section 7 presents a brief summary, conclusions drawn, and the hypotheses for future research.

2 Related Work

The growing complexity and the short release cycles of embedded systems stimulated the transfer of model-driven development techniques to the domain of embedded systems. Research in this area focuses primarily on two general directions: Modeling Languages for embedded system design, and approaches using standard notations such as UML.

As a first step, formal languages such as Z [16], functional decomposition [23], or state-based notations [10] were used. However, these approaches are lacking proper tool support, and do not facilitate reuse on higher level of abstractions than the implementation level. Newer developments such as MATLAB [21] or MODELICA [8] provide sufficient tool and (additional) methodological support. However, they lack effective means for reuse and adaptation compared to those provided by component-oriented approaches. Recently the Unified Modeling Language (UML) [25] was adapted for modeling embedded and real-time systems. However, UML still lacks precise semantics unlike more formal modeling notations such as SDL, and being a mere language, it requires a systematic method which defines what should be modeled, when and how it should be modeled.

Approaches such as OMEGA [12], HIDOORS [29], or FLEXICON [19], or the work presented in [6], [7], [17], [20], [27] define development methods for real-time and embedded systems using the UML. Although, a step in the right direction, they often do not use the enhanced features of UML 2.0 concerning embedded systems, do not establish systematic means for handling complexity, nor do they address reuse according to the component paradigm. In addition, they often neglect the specific requirements of small embedded systems that run on a microcontroller and that have scarce resources.

Another problem, as stated by Khan et al [13], is that the support for mapping UML (2.0) models to code is still inadequate. Typically, embedded system developers are used to developing systems according to the procedural paradigm. In general, a complete transition to object- or component-orientation is impossible due to the required time and space efficiency of the product, or due to standards to be followed (e.g., DO-178B in the civil aviation domain). However, embedded system development would benefit from the advantages of MDD [13] if the advocated technologies can be integrated into the existing development processes (i.e., keep C as target language). But, most approaches and tools either map models to languages such as Java (i.e., resulting in run-time performance, memory, or timing problems [13]), or use straightforward mapping strategies (UML to C) that neglect concepts such as inheritance or dynamic binding.

3 Research Approach

By applying MDD and CBSE, engineers expect an increase of model or component reuse, and, thus, shorter time-to-market, improved adaptability, and higher quality. However, introducing MDD and CBSE principles in an organization is generally a slow and incremental procedure [18]. Typically a company will build some reusable components in the beginning, and, in case of successful reuse, more and more code will be encapsulated into reusable components. The motivation for performing the case-study presented in this paper is to investigate the relationship between initial component developments and later reuse for a real system, as well as the impact of reuse technologies on quality and time-to-market.

3.1 Research Questions

Several factors concerning the development process and its resulting product are recorded throughout the case study in order to gain knowledge about using MDD and CBSE for the development of small embedded systems. The research questions of this case-study focus on two key sets of properties of MDD in the context of component-oriented development. The first set of questions (Q1-Q4) will lead to an understanding of basic and/or general properties of a methodological approach to embedded system development:

- *Q1*: Which process was used to develop the system?
  Answering this question will give a brief qualitative description of the method used for developing the initial 'Exterior Mirror System'.
- Q2: Which types of diagrams have been used?
  The UML standard, in its current form, offers 13 different diagram types for modeling software system properties. Are all diagram types needed within system development or is there a specific subset sufficient for a domain.
- Q3: How were models transferred to source-code?
  Embedded system developers typically work in a procedural language environment (i.e., C). They often have difficulties to start MDD using UML since the transformation of UML concepts into C language concepts is tricky [13].
- Q4: How was reuse applied and organized?
  Reuse is a central element of MDD and CBSE and it is generally recognized as the most relevant factor concerning quality, time-to-market, and effort. How-ever, reuse does not simply happen, rather it must be systematically built-in and supported (i.e., components have to be developed for reuse).

The second set of questions (Q5-Q9) is concerned with the resulting product of the MDD/CBSE approach. The developed systems are examined from a customer's point of view, with respect to code-size, defect density of the released code, and time-to-market.

- *Q5: What is the model-size of the systems?*
  MDD often has the smell of creating a large overhead of models even for tiny projects, assuming more complexity and problems in understanding and communication. Model-Size is calculated by using standard metrics as defined in [15].
- *Q6: What is the defect density of the code?*
  Defect density is computed per one hundred lines of code.
- *Q7: How long did it take to develop the systems and how is this effort distributed over the requirements, design, implementation, and test phases?*
  Savings in effort are one major promises of MDD and CBSE [28], though it is expected that these do not occur immediately (i.e., in the first project), but in follow-up projects (re-) using pre-defined components. Effort is measured for all development phases to identify where savings are realized.
- *Q8: What is the size of the resulting systems?*
  In embedded-system development, memory space is a sparse resource (e.g., often below 10Kbyte) and program size of uttermost interest. MDD for embedded systems will only be successful if the resulting code size, obtained from the models, is small.
- *Q9: How much reuse did take place?*
  Reuse is one of the central success factors of MDD and CBSE. However, reuse has to be viewed as an upfront investment which pays-off in follow-up projects. Thus, reuse must be examined between projects and not within a project.

3.2 Research Procedure

The focus of this paper is on methodological support for the component-based and model-driven development of embedded systems with UML. The assumption thereby is that by using a systematic method, such as MARMOT, efficient reuse, including other benefits such as shorter time-to-market, can be obtained. However, such claims need to be evaluated (e.g., in form of a case study). Since it is expected that the benefits of MDD and CBSE are first realized in follow-up projects, an initial mirror control system was developed and documented, to be used as basis for further application engineering.

For the case study, students of the Department of Computer Science at the Technical University of Kaiserslautern used the initial system documentation and MARMOT as method in the context of different embedded systems development projects. The students were taught basic software engineering principles, object-oriented development techniques, and UML. All students already had a good, industrial-level, knowledge of developing micro-controller based applications due to part-time employment at local companies. Students knew that data would be collected and that an analysis would be performed on the data. However, they were unaware of the concrete nature of questions/hypotheses being tested.

The student projects were organized according to typical reuse-situations in component-based development. In the context of these projects, a number of measurements to answer the research questions of section 3.1 were performed. In detail, the data comprised the following measures:

- *Model-Size* measured using the absolute and relative size measures proposed in [15]. In addition, figures on the number of classes in a model (NCM), number of components in a model (NCOM), number of diagrams (ND), etc. were used. In an embedded project NCOM describes the number of hardware and software components within the system. Since CBSE views every entity with a significant functionality as a component, NCM is used to denote the number of software components. In general, these metrics are comparable to the traditional LOC or McCabe's cyclomatic complexity (MVG) metric for estimating the size and nesting of a system's program code [14]. They can be used to compare sizes of implementations.
- *Code-Size* measured in normalized LOC (i.e., without comment and blank lines).
- The *amount of reused elements* within a system is described as the proportion of the system which can be reused without any changes or with small adaptations. Measures are taken at the model and the code level and are normalized using the system size (model, LOC).
- *Defect density* is measured in defects per 100 LOC.
- *Development effort* and its distribution over development phases are measured as development time. Since all projects are quite small, development hours are used as the unit for measurement. Effort data was gathered by filing effort sheets every day.

4 Overview of MARMOT

Reuse is a key success factor in industry today, and it can be seen as a major driving force in hardware and software development. Reuse is pushed forward mainly by the growing complexity of systems. This section introduces a methodology for the component-based development of embedded systems, referred to as MARMOT that is specifically geared toward facilitating reuse in embedded systems development. MARMOT is an extension to the KobrA method [2], a component-based development framework for information systems, and it adds concepts addressing the specific requirements of developing embedded systems.

4.1 Principles of MARMOT

Composition is a key activity in component-based development. MARMOT recognizes this fact in that it advocates composition as the single most important engineering activity. A system can be viewed as a tree-shaped hierarchy of components, in which the parent/child relationship represents composition, i.e., a super-ordinate component is composed out of its contained sub-ordinate components.

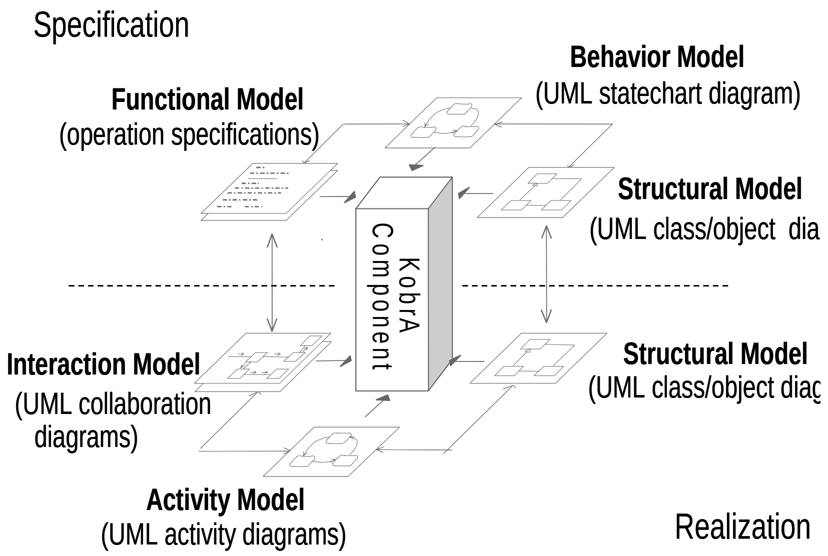


Figure 1 MARMOT Component Model

A long established principle of software engineering is the separation of the description of what a software unit does from the description of how it does it.

This facilitates a "divide-and-conquer" approach to modeling in which a component can be developed independently. It also allows new versions of a component to be interchanged with old versions provided that they do the same thing and abide by the same interface. Following this principle, each component within a system can be described through a suite of models, for example UML diagrams or other textual documents, as if it was an independent system in its own right (see Figure 1).

4.2 MARMOT Process Model

The core principle of MARMOT is separation of concerns, so it associates its main development effort with two basic dimensions that map to four basic activities [2]. These are depicted in Figure 2:

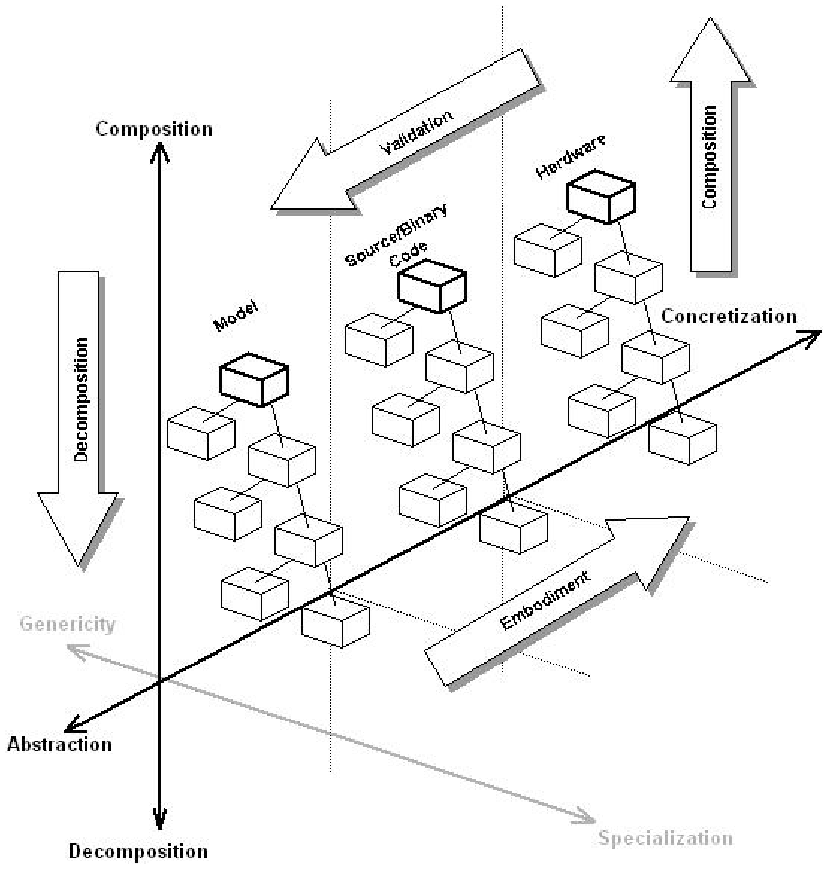


Figure 2 Development dimensions of MARMOT

- **Composition/Decomposition dimension**.
  Decomposition follows the "divide-and-conquer" paradigm, and it is performed to subdivide the entire embedded system into smaller parts that are easier to understand and control. Composition represents the opposite activity, which is performed when the individual components have been implemented, or some others reused, and the system is put together.
- **Abstraction/Concretization dimension**.
  This is concerned with the implementation of a system and a move toward more and more executable representations. The activity is called *embodiment*, and it turns the abstract system represented by models into more concrete representations that can be executed by a computer. The move back is called *validation*. This activity checks whether the concrete representations are in line with the abstract ones.

4.2.1 Decomposition

An embedded system development project always starts above the top left-hand side box in Figure 2. The box represents the entire system to be built. Before the specification of the box, the concepts of the domain or the physical world in which the system is supposed to operate have to be determined. This comprises descriptions of all entities relevant in the domain including standard hardware components that will eventually appear on the right-hand side towards concretization. In embedded systems, these implementation-specific entities often determine the way in which a system is divided into smaller parts [9]. During decomposition, newly identified logical parts of the system are mapped to existing components. Whether these are hard- or software does not play a role at this early phase because of the way all components are treated in terms of collections of descriptive artifacts, that is, models.

4.2.2 Embodiment

During decomposition, the shapes of each identified individual component are defined in an abstract and logical way. The system, or its parts thereof, can then be moved towards more concrete representations. This means they become platform specific.

4.2.3 Composition

After having implemented some of the boxes and having some others reused, the system can be assembled according to the abstract model. Therefore, the subordinate with their respective super-ordinate boxes have to be coordinated in a way that exactly follows the component standard previously described.

4.2.4 Validation

A final activity, validation, is carried out in order to check whether the concrete composition of the embedded system corresponds to its abstract description.

4.3 The Basic MARMOT Product Model

With MARMOT, components are built on the same fundamental principles that are coming from object technology. Therefore components follow the principles of encapsulation, modularity and unique identity that most component definitions put forward [28], and these lead to a number of obligatory properties:

- Composability is the primary property of a MARMOT component, and it can be applied recursively: components make up components, which make up components, and so on.
- Reusability is the second key property that can be separated into development for reuse, which deals with how components have to be specified and treated, so that they can be reused, and development with reuse, dealing with the integration and adaptation of existing components in a new application.
- Having unique identities requires that a component may be uniquely identifiable within its development environment as well as within its runtime environment. MARMOT provides the principles for that.
- Modularity/encapsulation refer to a component's scoping property as an assembly of services, which is also true for a hardware component, and as an assembly of common data, which is true for the hardware and the software parts of an embedded component. Here, the software only represents an abstraction of the hardware that essentially provides the memory for the data.
- An additional important property is communication through interface contracts which becomes feasible in the hardware or embedded world through typical software abstractions. Here, the additional hardware wrapper of MARMOT realizes that the typical hardware communication protocol is translated into a typical component communication contract.

Composition along the Composition/Decomposition dimension turns a MARMOT project into a tree-shaped structure with consecutively nested abstract component representations. Such a tree is called containment tree. Every box in the tree, each representing a component or a system in its own right, is made up of a component specification and a component realization. The specification is a suite of descriptive artifacts that collectively define everything externally knowable about a component. These descriptions fully specify a component in a way that it can be assembled in a system and used by the system. The realization is a suite of descriptive artifacts that collectively define how a component is internally realized. According to the composition principles, components can be made up of other components. Any component in a MARMOT containment tree can therefore be a containment tree in its own right, and, as a consequence, another MARMOT project.

5 CASE-Study

5.1 Mirror-Control System

The Mirror-Control system is an embedded system composed of electrical and mechanical components and is used to control the movement of an exterior mirror of a car (see Figure 3). The system allows the mirror to be moved horizontally and vertically to a convenient position for a driver. Cars supporting different driver profiles can store the mirror position and recall as soon as the profile is activated. Within this paper a simplified version is used in order to illustrate the approach without going into the very detail.

Figure 3 Exterior Mirror

5.2 System Description

The Mirror-Control system was realized in a simplified version using a microcontroller, a button, and two servos (aka the Servo-Control System). The microcontroller (i.e., an ATMEL^{TM} Mega8) has limited performance (i.e., 1-8 Mhz) and I/O capabilities. In detail, the system controls two servo-drives[1] via potentiometers, and indicates their movement (-45 — +45 degrees) on a small LCD panel.

In detail, this system requires the microcontroller to read in values from the potentiometers (requires an analog-digital conversion), converts them to the turn-

[1] A servo drive receives a command signal from a control system, amplifies the signal, and transmits electric current to a servo motor in order to produce motion proportional to the command signal. Typically the command signal represents a desired velocity. A velocity sensor attached to the servo motor transmits the actual motor velocity to the servo drive. The servo drive continually compares the actual motor velocity with the commanded motor velocity to generate an output to the motor that will tend to correct any error in the velocity. [wikipedia]

ing degree, and generates the needed servo control signals (i.e., this requires PWM signal generation using timers and interrupts), while at the same time indicating movement and degree on the LCD display. In addition, the system can store a position which can be recalled by simply pressing the button. Positions are stored by pressing the button for more than five seconds. Storing and recalling is also visualized on the LCD display. Figure 4 shows the system based on the MyAVR-Board.

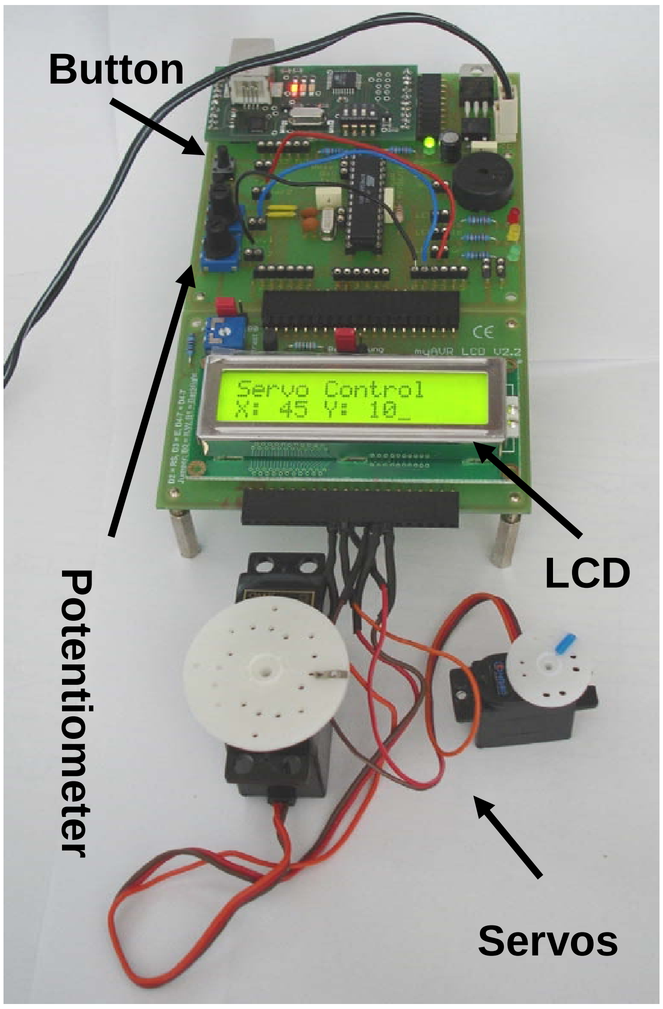


Figure 4 Example System

5.3 Hardware

Figure 5 shows a simplified circuit diagram of the board used for the Servo-Control system. However, the diagram, it is not complete. In detail, external components such as the servos or the LCD display are missing. In addition, unused components, which are part of the commercial (e.g., three LEDs and a beeper) are still contained. The missing components are meant to be attached to the female connectors. Connections between the processor, the button, and the potentiometers are simply realized using patch cables.

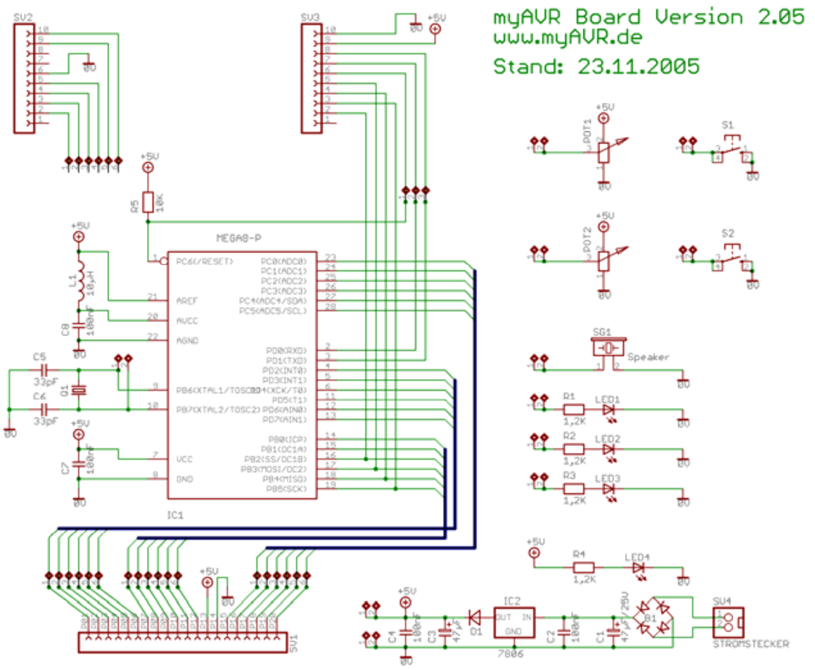


Figure 5 Electronic Circuit – Servo Control [myAVR06]

The control circuitry inside the servo must receive a stream of pulses whose widths may vary between about 1 ms and 2 ms. These pulses must occur at intervals of about 10 to 20 ms. A potentiometer coupled to the rotation of the output shaft produces a voltage corresponding to the angle of the shaft. The control circuitry compares the “average” (i.e., low pass filtered) voltage of the control signal with the voltage from the potentiometer, and the shaft rotates until the two voltages are the same.

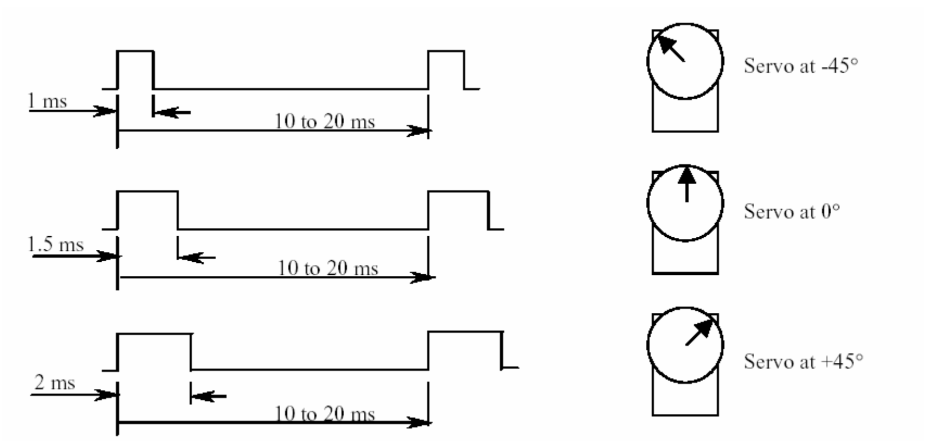


Figure 6 Timing and Servo Rotation

As shown in Figure 6 the servo needs a pulse-width modulated (PWM) control signal in order to position the output shaft. Pulse-widths vary between approximately 1ms - 2 ms, and have a period of 10 ms - 20 ms. The Atmega8 can produce these signals directly, however, the timing of these signals must be very precise in order to keep the servos from jittering. This is not a problem when all the microcontroller is doing is running servos (as in this system). However, for more complex systems a multiplexer has to be used.

5.4 Embedded Software

5.4.1 Requirements Modeling – Context Realization

The requirements of the servo control system are described by use case diagrams consisting out of a textual and a graphical representation and an activity diagram representing the general flow of control. Figure 7 presents the use case diagram for the servo control system. The actor 'User' initiates the task of controlling the servo rotation, represented by the 'Control Rotation' use case. This use case requires …

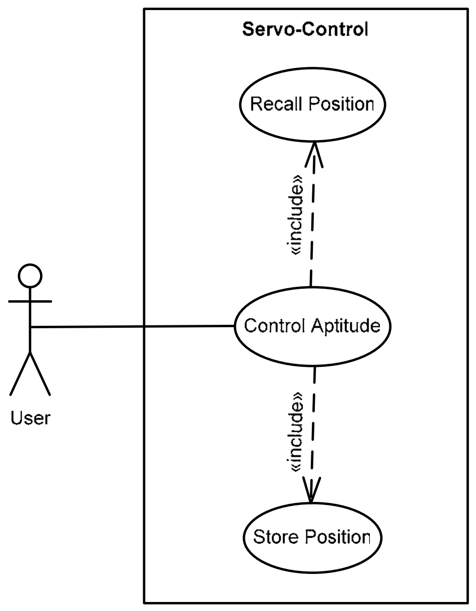


Figure 7 Use Case Diagram – Servo Control System

In addition to the graphical depiction every use case is also textually specified in order to capture necessary details, not contained in the UML diagram. Table 1, Table 2, and Table 3 presented these textual representations using a tabular notation. In general, the system has to provide three different functionalities: (1) Control the horizontal and vertical aptitude (Table 1), (2) Persistently store the current mirror position (Table 2), and (3) Move the mirror to a previously stored position (Table 3).

Name	Control Aptitude
Actor	User
Goal	To control the vertical and horizontal aptitude of an exterior mirror.
Description	The user controls the rotation of two servos (i.e., horizontal and vertical aptitude), and can store or recall a specific position by pressing a button. All activities are additionally visualized via a connected LCD display.
Exceptions	• The aptitude is limited to a range in degrees of -45 to +45 in every direction. • Direct servo control (using the potentiometer) is disabled while the button is pressed
Rules	NA
Quality Requirements	Timing issues are highly important since controlling servos requires the generation of PWM signals.

I/O	Input • Potentiometer (2x) Output • LCD Display
Pre-Conditions	System is powered and initialized
Post-Conditions	The vertical and horizontal aptitudes are controlled.

Table 1 Use Case Description – Control Aptitude

Name	Store Position
Actor	User
Goal	To persistently store the vertical and horizontal aptitude of an exterior mirror.
Description	By pressing the button for more than 5 sec the current rotation degree for both servos (i.e., aptitude) is persistently stored. Storing is indicated on the LCD display.
Exceptions	NA
Rules	NA
Quality Requirements	NA
I/O	Input • Button Output • LCD Display
Pre-Conditions	System is powered and initialized
Post-Conditions	The current mirror position has been persistently stored.

Table 2 Use Case Description – Store Position

Name	Recall Position
Actor	User
Goal	To recall a stored mirror position (vertical and horizontal aptitude).
Description	By pressing the button not longer than 5 sec the rotation degree of both servos is brought to the stored, persistent, position. Recalling is indicated on the LCD display.
Exceptions	Assumes that there is a position stored. If not (i.e., at first start-up) a default position has to be used and stored.
Rules	NA
Quality Requirements	NA
I/O	Input • Button

	Output • LCD Display
Pre-Conditions	System is powered and initialized
Post-Conditions	The mirror was moved to the persistently stored position.

Table 3 Use Case Description – Recall Position

Figure 8 visualizes the interaction diagram related to the context realization of the Servo-Control system. It provides an alternative view of the way in which user tasks are performed and shows the typical sequence of operations concerning the overall system. In addition, the figure shows the signals that, created by user events, are send from the micro-controller to the software system. To keep things consistent user and hardware events are named equally.

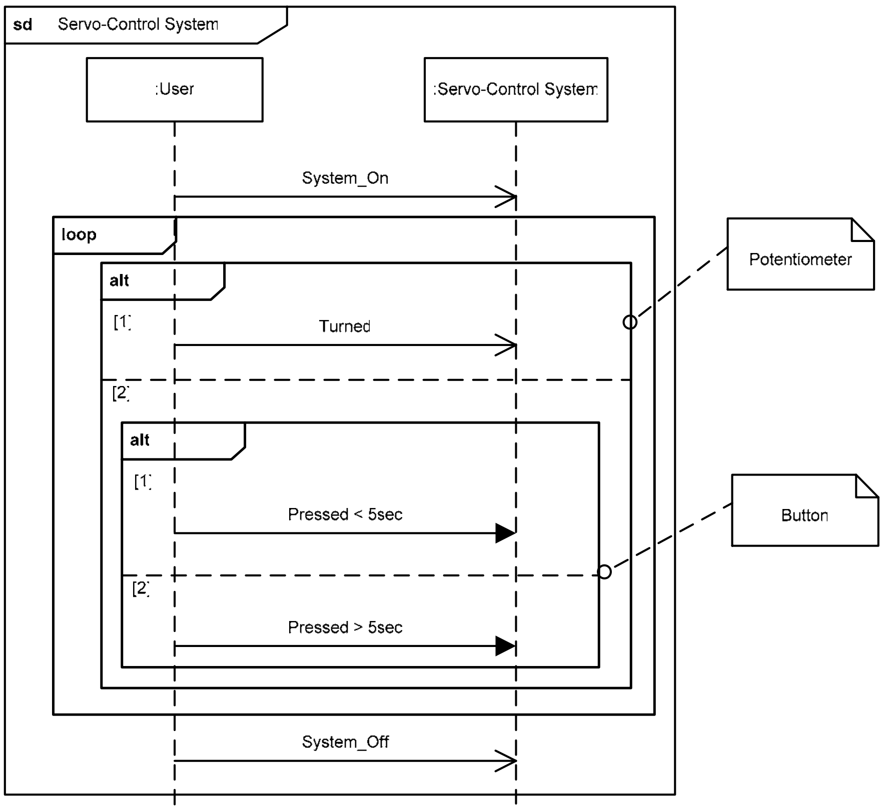


Figure 8 Interaction Model – Servo-Control System

Figure 9 shows two UML representations of the electronic circuit shown in Figure 5. These diagrams represent the structural model of the context realization. Electronic components are mapped to UML classes marked by the stereotypes “Component”, to indicate their nature, and “Hardware” to distinguish them from later driver classes using the same name. This fact is also depicted in

Figure 10 by distinguishing hardware and software components. In order to ensure consistency software components addressing hardware functions (i.e., driver components) are named according to the controlled hardware component.

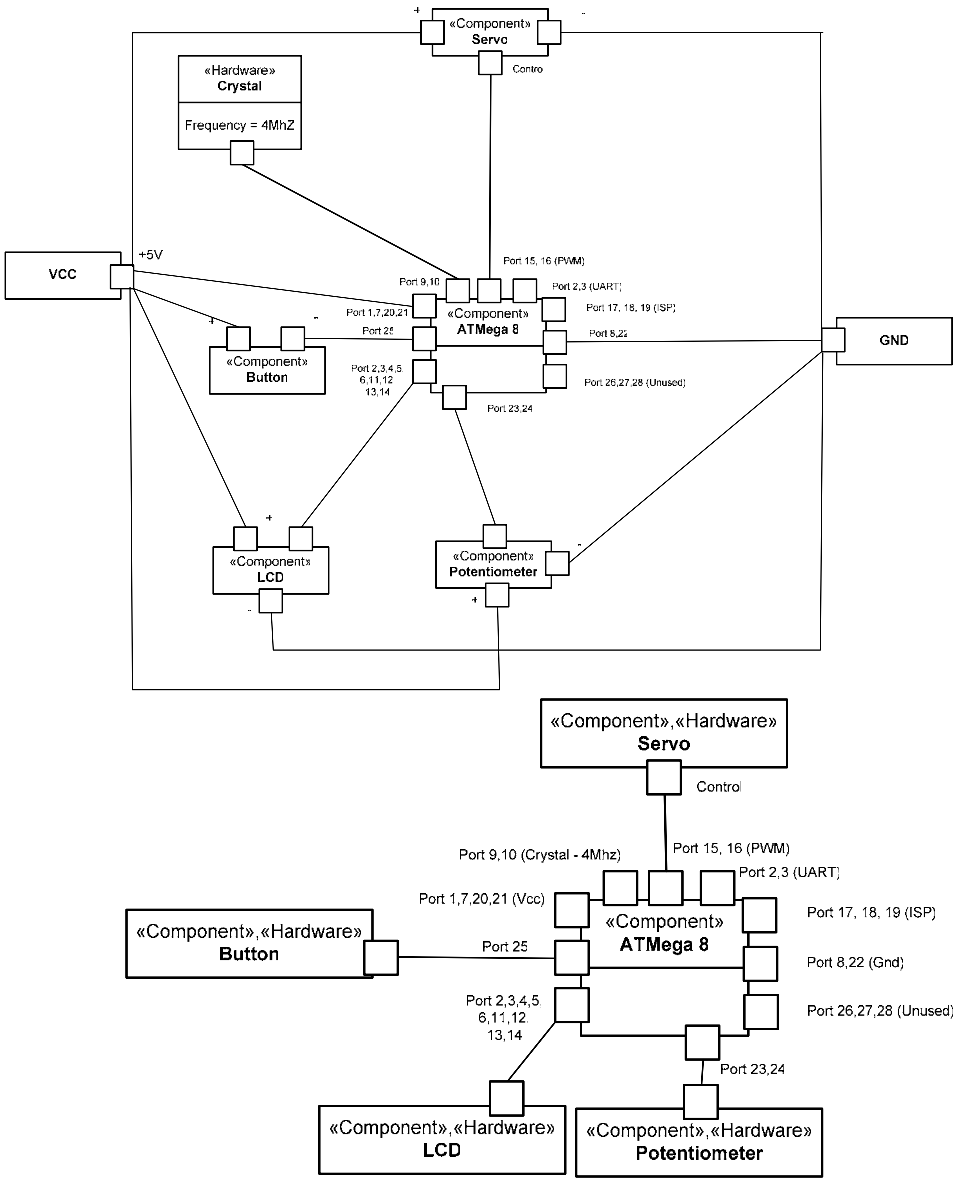


Figure 9 UML Representation – Hardware

The diagrams of Figure 9 were manually translated from the circuit diagram (partly shown in Figure 5) using specific mapping rules and guidelines. In future, mappings (EAGLE format circuit diagrams to XMI based UML diagrams) will be performed automatically, using the *HERMES* toolkit currently developed at Fraunhofer IESE.

HERMES based mappings are performed in three semi-automatic steps. The first step results in a one-to-one mapping of the circuit to a UML diagram. In the second step, relevant components and their ports are identified, whereby all other elements (e.g., resistors, capacitators, etc.) are removed This step then results in a diagram as that shown on the top of Figure 9. In a final step, component ports are minimized, e.g., by removing ground (GND) ports. This then results in a diagram as that shown on the bottom of Figure 9.

Please note that the class 'Crystal' represents a quartz oscillator which determines the processor clock speed[2], and which is therefore needed for timing and signaling (e.g., PWM signals) issues. However, the 'Crystal' class has been removed from the final diagram since it is sufficient to determine the port and speed at the processor side.

The previously defined diagrams and textual specifications form the 'context realization' of the servo control system. In a sense the "context" can be viewed as a pseudo component at the root of the development tree. The system is then treated as a regular component, since the context provides the encapsulating realization against which it can be specified. Using the preliminary information together with the planned system architecture (see Figure 10), allows the definition of a preliminary containment hierarchy of the Servo-Control system (see Figure 10). This hierarchy specifies how course-grained components are "made up of" finer-grained components, in a recursive manner, down to the level of small, primitive components.

[2] The processor used for the Servo Control System is a ATMEL Mega8L designed for low-power and clock speeds up to 8Mhz. Concerning the Servo Control system a 4Mhz oscillator is used.

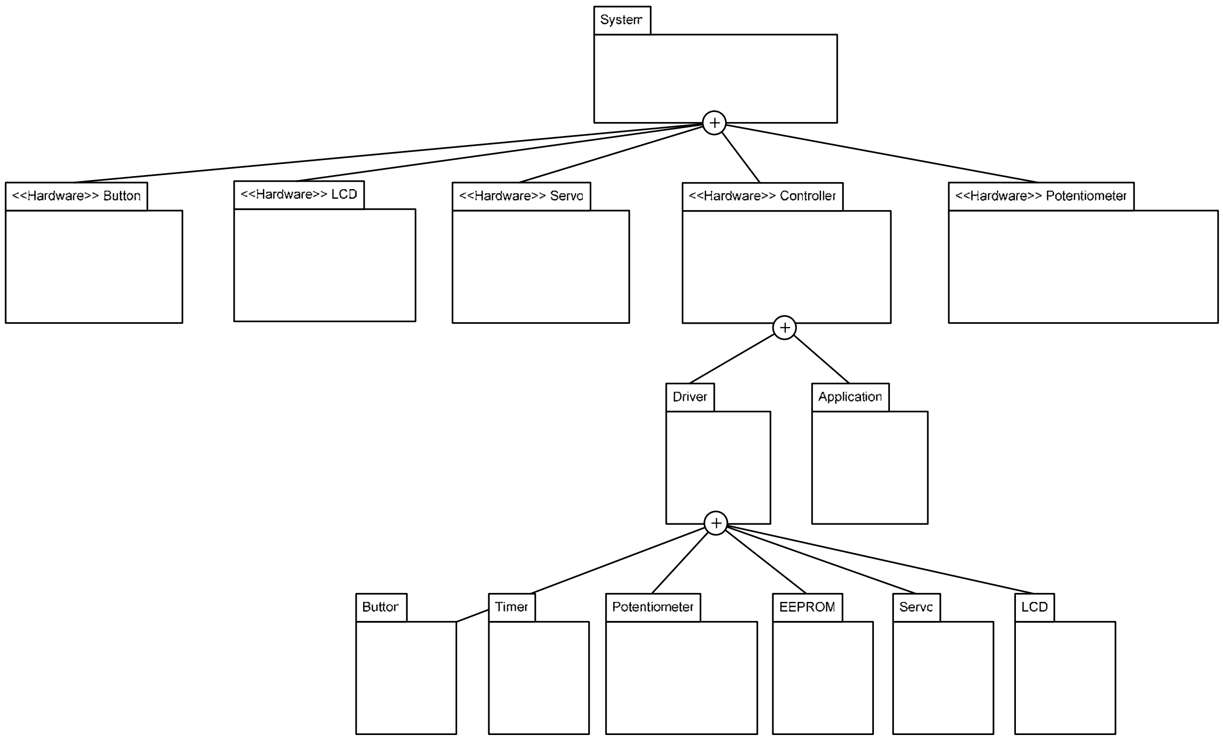


Figure 10 Containment Hierarchy

5.4.2 Application Engineering

The purpose of the application engineering activity is to generate a specific set of modeling artifacts that meet the needs of a specific system. Since the hardware environment is pre-defined, development focuses on the software part, namely the *Driver* and *Application*, components. The *Controller* component is a container without any software functionality.

5.4.2.1 Application

Figure 11 shows the specification level class-diagram of the *Application* component and denotes its context. Since this is an embedded system, the component does not offer any operations to the outer world, but reacts to signals. This is denoted by the UML 2.0 stereotype <<signal>>.

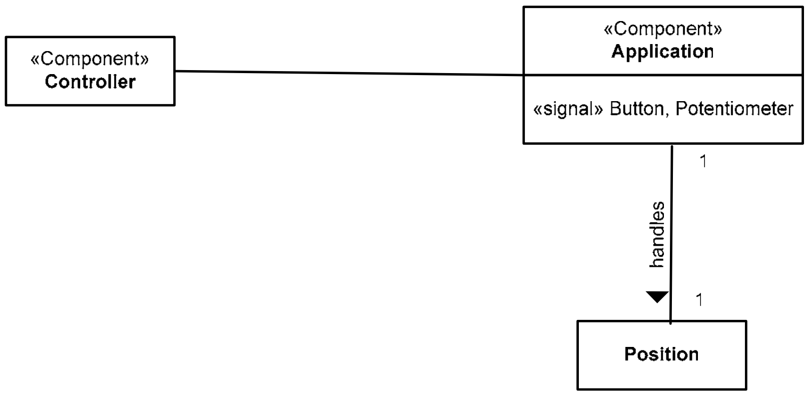


Figure 11 Class Diagram - Application (Specification)

Table 4 specifies the functionality of the *Application* component by means of an operation specification.

Name	Main
Description	The turning degree of two servos (i.e., for controlling the vertical and horizontal aptitude of an exterior mirror) is controlled by means of two potentiometers. In addition by the press of a button the current position (turning degrees) can be persistently stored, if the button is pressed for more than 5. If the button is pressed shorter than 5 sec the turning degrees of both servos is set to the stored position.
Constraints	--
Receives	• Signals are received from the Button and the Potentiometer. • In addition, the System_On and System_Off signals are received from the microcontroller.
Returns	Controls the Servo Position and prints messages on a display (see Use Cases).
Sends	Operations calls to the *Driver* component concerning • LCD (init, write) • Servo (init, set) • Potentiometer (init, send) • Button (init) • EEPROM (store, retrieve) • Timer (start, stop)
Reads	• The ADC value of the Potentiometer (in *Driver*). • Stored data within the EEPROM (in *Driver*).
Changes	• The turning degree of both servos (in *Driver*). • Stored position data in the EEPROM (in *Driver*)
Rules	• Pressing the Button for less than 5 sec will move the servos to the stored position

	• Pressing the Button for more than 5 sec will store the actual position as the default position
Assumes	--
Result	The vertical and horizontal aptitudes are controlled, whereby positions are stored and retrieved.

Table 4 Operation Specification – Application

The state diagram of the *Application* component (see Figure 12) shows that the system is typically in the Set state where the position of the servos can be controlled. By pressing the button the system will go into the Recall or Store state in which manual servo control is disabled.

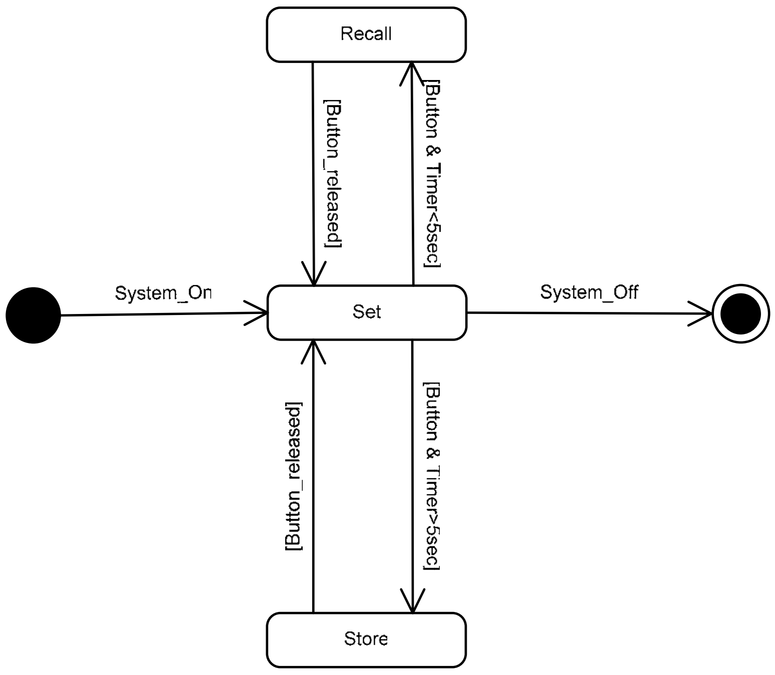


Figure 12 State Diagram – Application

Figure 13 shows the realization level class-diagram of the *Application* component. The diagram defines that the application component needs services of the 'Driver' component to fulfill its tasks. However, there is no direct link between both components at this level since they appear at the same level of the component hierarchy. However, the MARMOT visibility rules define that they can see each other, via their super component, so that there is a dependency between these two.

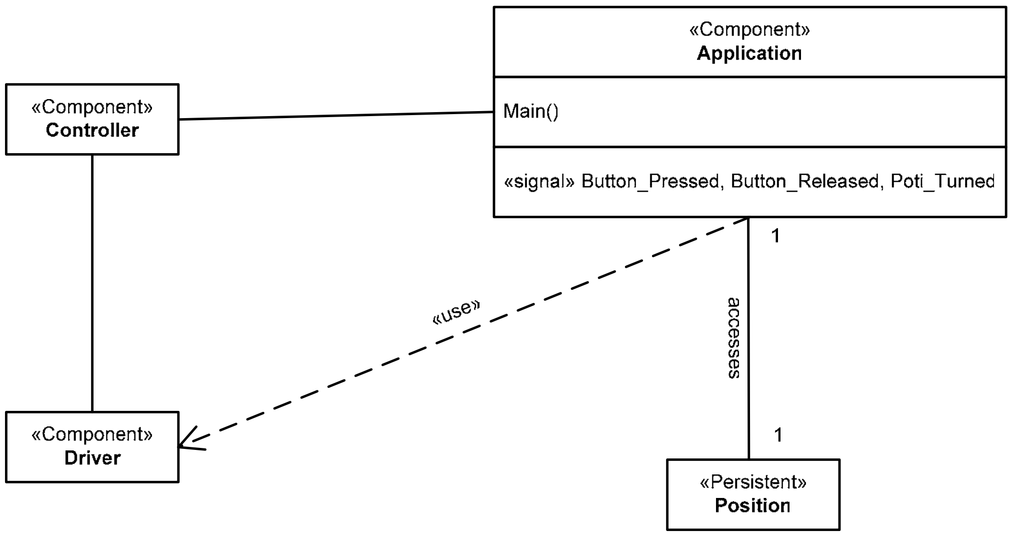


Figure 13 Class Diagram – Application (Realization)

The class *Position*, which is a data-entity handled by the *Application* component, has been refined using the <<Persistent>> stereotype. In detail, this means that stored position data should survive system shutdowns. In turn, this means that the position data are stored in, and retrieved from the EEPROM of the controller (i.e., 1KByte concerning the ATMega8), which results in additional operations concerning the *Driver* component.

Another refinement within the realization of the *Application* component is the addition of the Main() operation which encapsulates the system control algorithm. This is described in more detail within the component's activity diagram (Figure 15). Furthermore the general signals mentioned in the specification are refined to depict the nature of the concrete signals onto which the component has to react.

Based on the realization class diagram the communication of the *Application* component with other components has to be specified by means of interaction diagrams.

Figure 14 displays the relevant sequence diagrams for the *Application* component and defines the flow of communication between the *Application* and the *Driver* component.

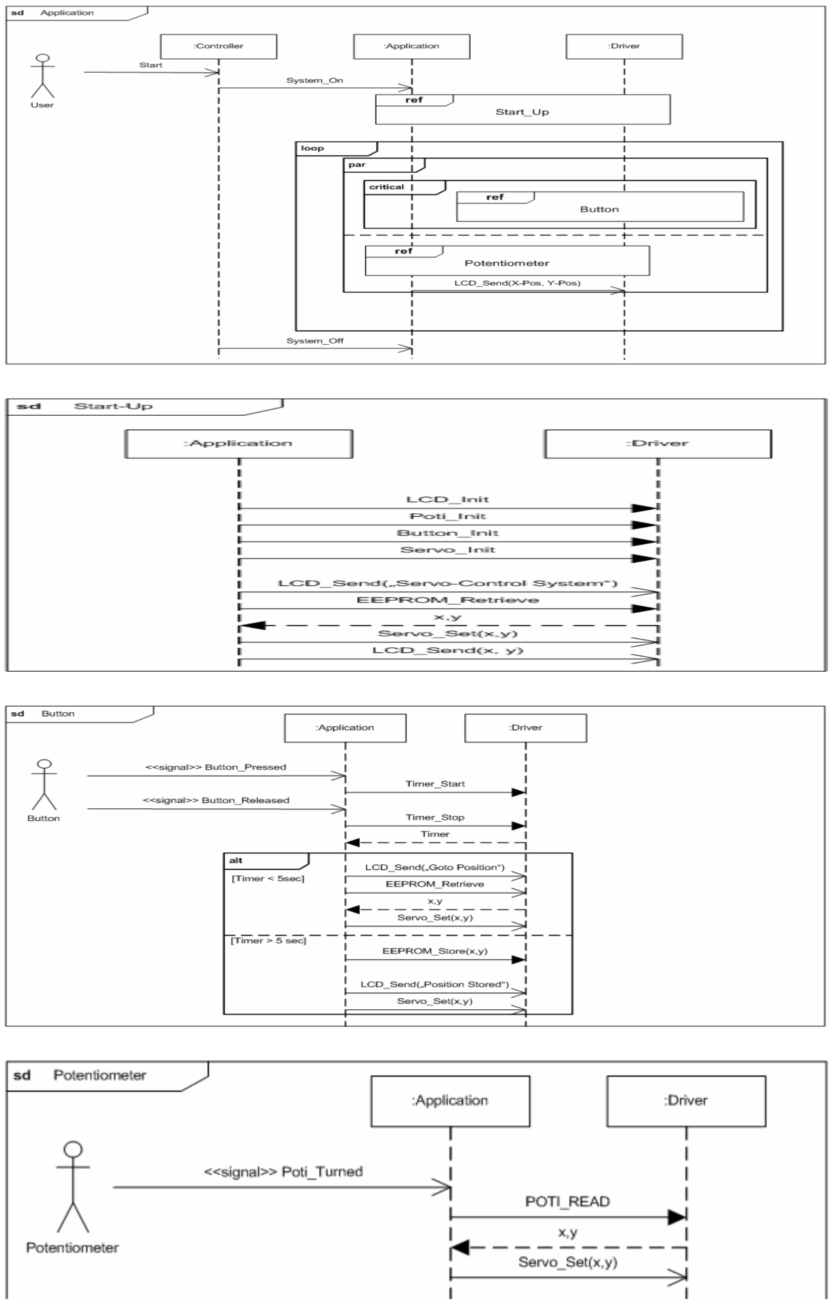


Figure 14 Sequence Diagrams – Application

The algorithmic behavior of the *Application* component is specified by means of an activity diagram (see Figure 15). Basically after an initialization an endless-loop is started in which the necessary control activities are performed. Using an endless loop is typical for micro-controller systems in order to prevent the system from resetting or getting in an undefined state. Within the loop the component reacts to various signals it receives from the *Driver* component. In general, the flow of control shown in Figure 15 becomes the Main() routine of the latter implementation.

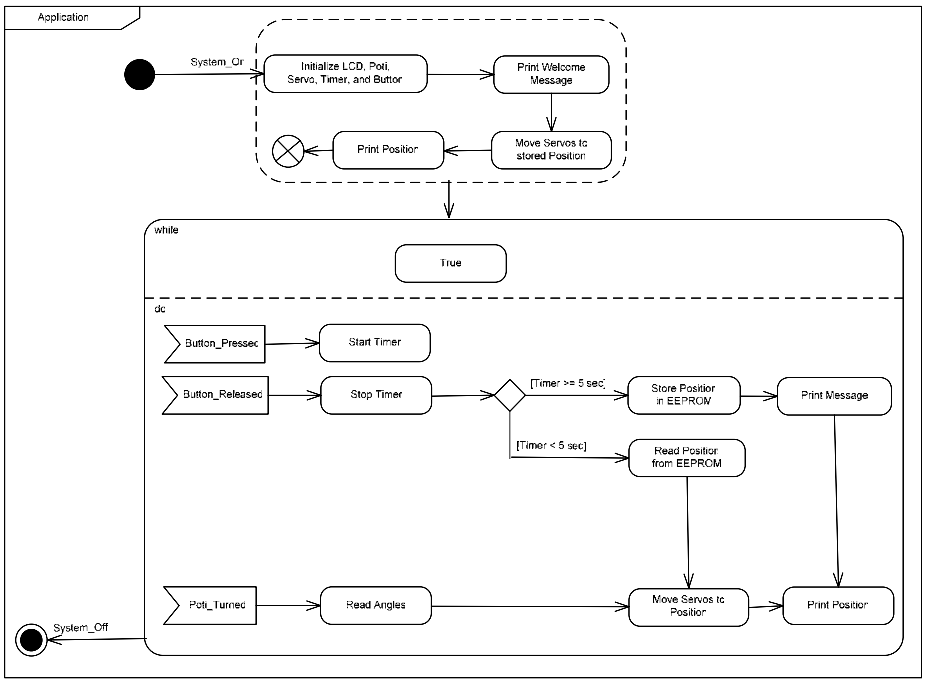


Figure 15 Activity Diagram – Application

5.4.2.2 Driver

The *Driver* component encapsulates the Service Access Points (SAPs) of the system from the application. Thus, it separates application and hardware, and thus, allows exchanging hardware components more easily. Therefore, the application can only access hardware elements via the *Driver* component.

Figure 16 shows the specification class diagram of the *Driver* component and specifies the operations[3] available via its interface. Furthermore, the diagram depicts that the driver component is able to send signals. These are then rerouted via the controller (i.e., a container) to the *Application* component. Thus, there is no need for a state diagram since *Driver* has only one state.

[3] Please note, that operation names are already indicating the later decomposition of *Driver* into subcomponents. Naming is a result of later consistency checks, aimed at making decomposition clearly visible.

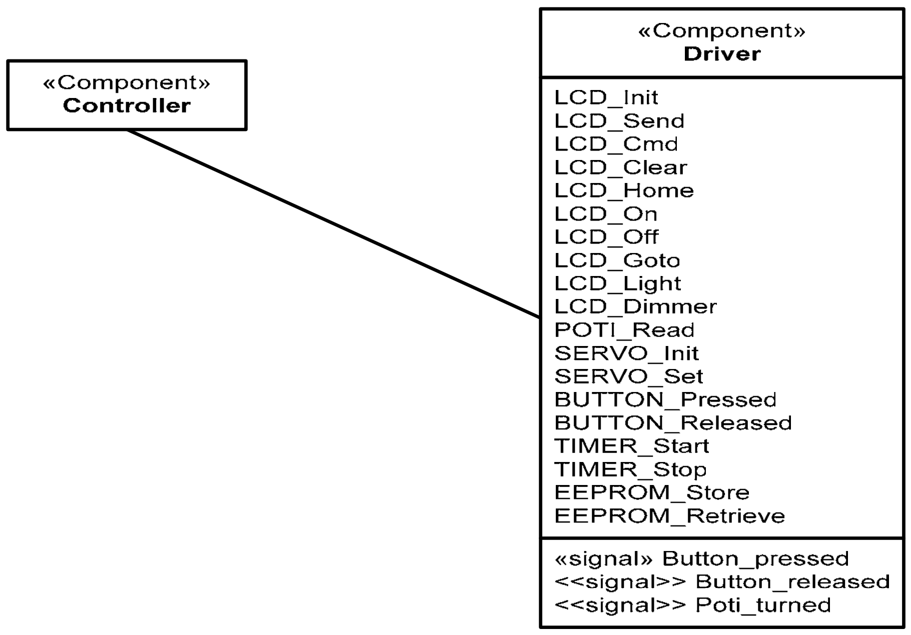


Figure 16 Class-Diagram – Driver (Specification)

Name	Name of the operation.
Description	Identification of the purpose of the operation, followed by an informal description of the normal and exceptional effects.
Constraints	Properties that constrain the realization and implementation of the komponent.
Receives	Information input to the operation by the invoker.
Returns	Information returned to the invoker by the operation.
Sends	Signals which the operation sends to imported komponents. These can be events or operation invocations.
Reads	Externally visible information accessed by the operation.
Changes	Externally visible information changed by the operation.
Rules	Rules governing the computation of the result.
Assumes	Precondition on the externally visible state of the komponent and on the inputs (in receives clause) that must be true for the komponent to guarantee the post condition (result clause).
Result	Strongest post condition on the externally visible properties of the komponent and the returned entities (returns clause) that become true after execution of the operation with true assumes clause.

Table 5 Operation Specification – Driver´

Figure 17 shows the realization class diagram of the *Driver* component. The diagram specifies that *Driver* uses several classes to fulfill its services. Basically, every hardware component, connected to the controller (see Figure 9), is assigned to a class of the *Driver* component. This is also reflected by the naming of classes. Therefore, *Driver* is declared to be abstract which means that it will only route requests to specific "subclasses". Therefore, the *Timer* class was added since, which specifies the use of in-built (hardware) timers.

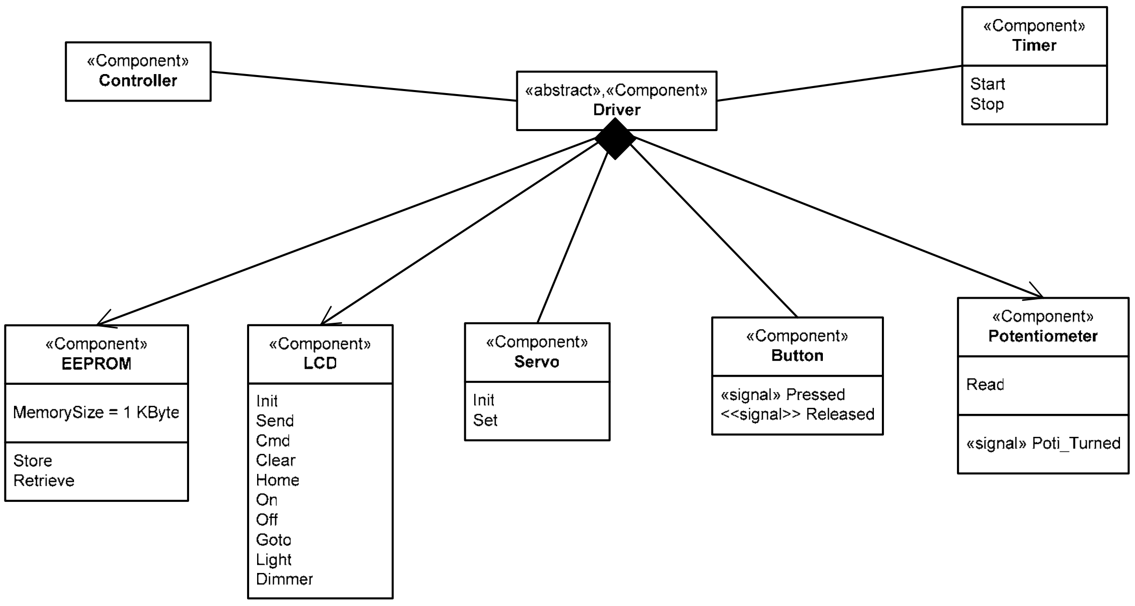


Figure 17 Class-Diagram – Driver (Realization)

Interaction Diagrams for the *Driver* component are simple. Since Driver is an abstract component it sends received command directly to its subcomponents. This is depicted by Figure 18[4]. The assignment of operations to subclasses is simple, since the specification class diagram uses the class-name as a prefix of the operation name. The only exception is the *Button* component. *Button* does not provide any externally visible operations but sends signals. Thus, there are no messages send to the *Button* component.

[4] For reasons of simplicity a wildcard notation has been used to indicated that all operations of a specific group are meant (e.g., LCD_* stands for all LCD display related operations of the *Driver* component).

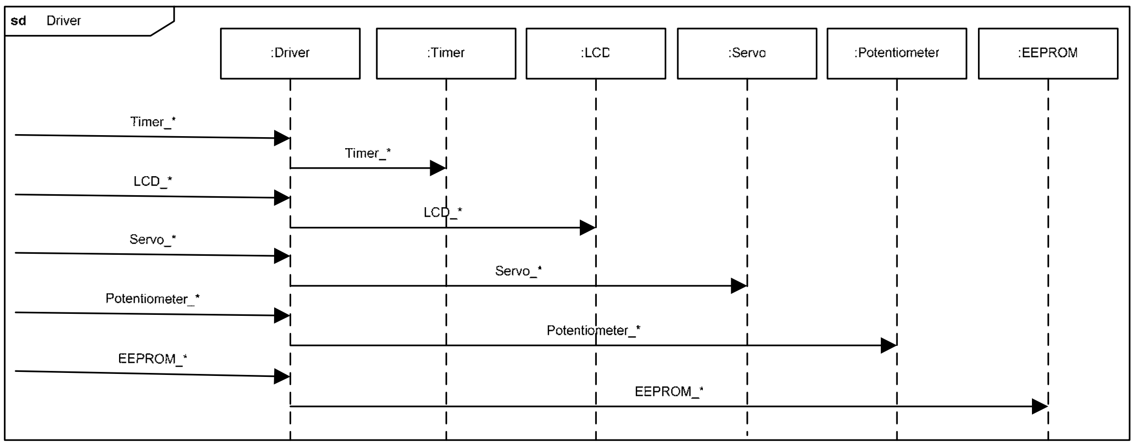


Figure 18 Sequence Diagram – Driver

Typically the realization of a component MARMOT contains activity diagrams, describing the algorithms and flow of control concerning specific operations. Since *Driver* is a container that propagates messages, activity diagrams would be quite simple, and are therefore intentionally left out.

5.4.2.3 Button

The *Button* component is responsible for reacting on physical button presses/releases in the context of the Servo-Control system. The specification class level diagram (see Figure 19) specifies that the *Button* component receives all its direct input via the *Driver* component. Communication relationships with components such as Controller are implicitly given since *Controller* is a super-component of *Button*.

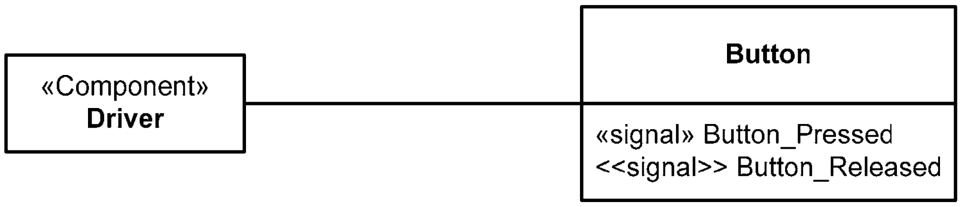
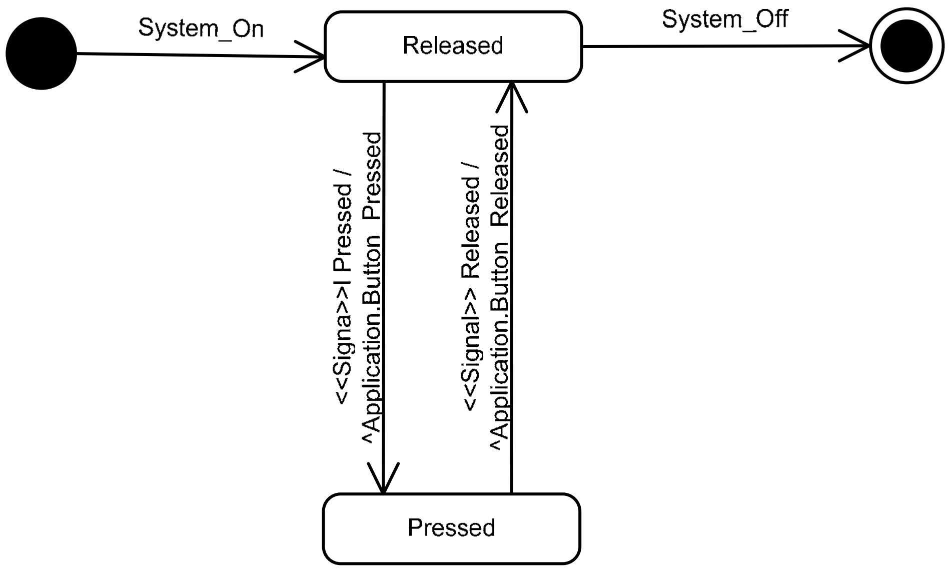


Figure 19 Class Diagram – Button (Specification)

In Figure 20 the state diagram concerning the *Button* component is shown. Basically the component has two different states since a button can solely be pressed or released. If one of these events occurs, the *Button* component will notify the *Application* (via Driver) by sending signals.



System_On
Released
System_Off
<<Signa>>l Pressed / ^Application.Button_Pressed
<<Signal>> Released / ^Application.Button_Released
Pressed

Figure 20 State Diagram – Button

Table 6 specifies Button’s behavior concerning received and send signals by means of an operation specification.

Name	<<Signal>> Button_Pressed / <<Signal>> Button_Released
Description	• <<Signal>> Button_Pressed: • <<Signal>> Button_Released:
Constraints	The system is powered and the hardware button is connected to port 25 of the microcontroller (see Figure 9).
Receives	The component receives signals (i.e., *Pressed* and *Released*) from the controller as soon as the hardware button has been pressed or released.
Returns	NA
Sends	As soon as the hardware button is pressed the component sends the signal *Button_Pressed,* or the signal *Button_Released* when the button is released to the *Application* component (via *Driver*).
Reads	NA
Changes	NA
Rules	NA
Assumes	NA
Result	Signals were sent to the Application component upon the receipt of the corresponding hardware events or signals.

Table 6 Operation Specification – Button

The realization class diagram for *Button* is equal to the specification class diagram (Figure 19) since no additional structural details have been identified. Therefore, it has been left out.

Figure 21 shows the sequence diagram concerning the *Button* component. In principle it sends signals to the *Application* component (via *Driver*) upon the receipt of signals from the microcontroller.

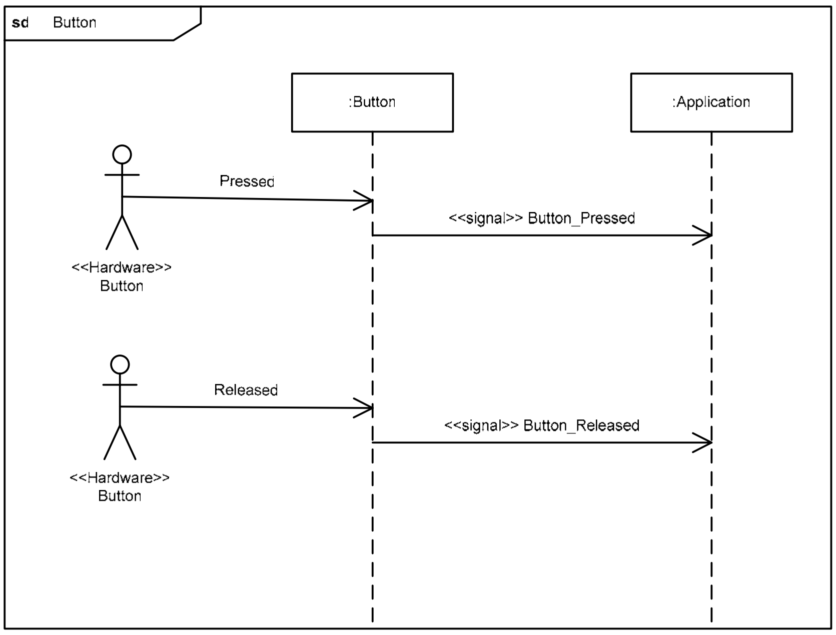


Figure 21 Sequence Diagram – Button

The simple behavior of the *Button* component is also reflected in the component's activity diagram (see Figure 22). In specific, once the component has initialized the button (i.e., define ports, interrupts, etc.), it reacts on a signal from the controller, occurring at physically pressing the button, and sends this signal on to the *Application* component.

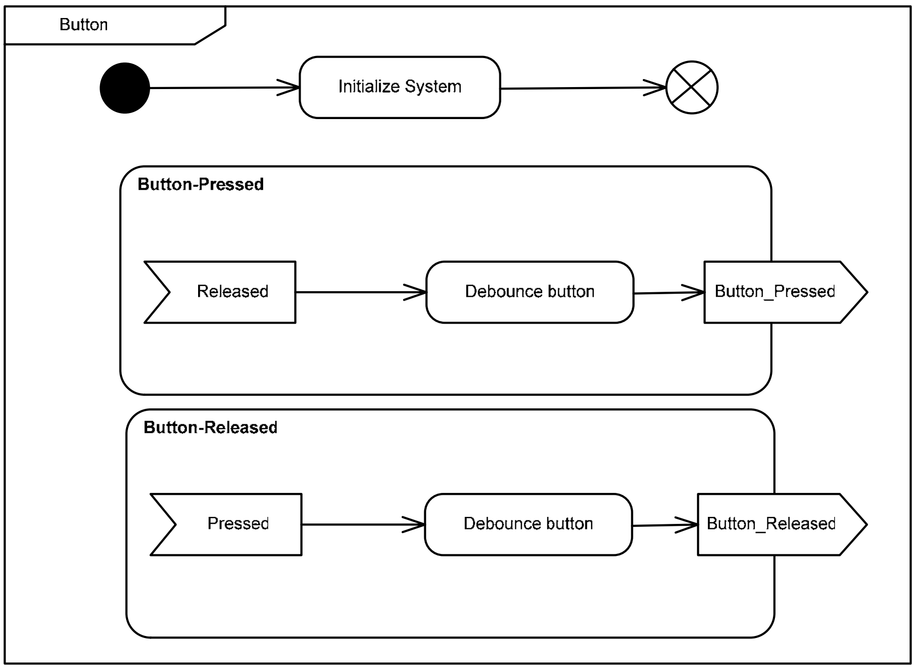


Figure 22 Activity Diagram – Button

The reason for having a driver component for the button is not only the encapsulation of hardware- from the application. Another reason is that buttons often bounce (see Figure 23). By physically pressing/releasing a button it might send a series of ON/OFF signals before it settles in a specific state. This is caused by the mechanical properties of a button and cannot be avoided easily. Thus, the hardware signal has to be preprocessed in order to get a clear signaling behavior.

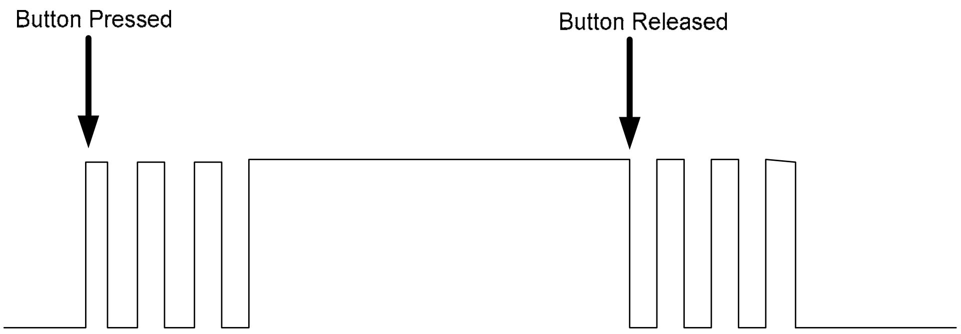


Figure 23 Button – Bounce

Timing diagrams provide a notation to specify system behavior with time. Figure 24 presents the timing diagram concerning storing/recalling positions. In general, the diagram describes system behavior with time in the context of messages/events passed among the different objects/lifelines. The 'Application' lifeline represents the 'Application' object which takes the main control decisions and which has three different states.

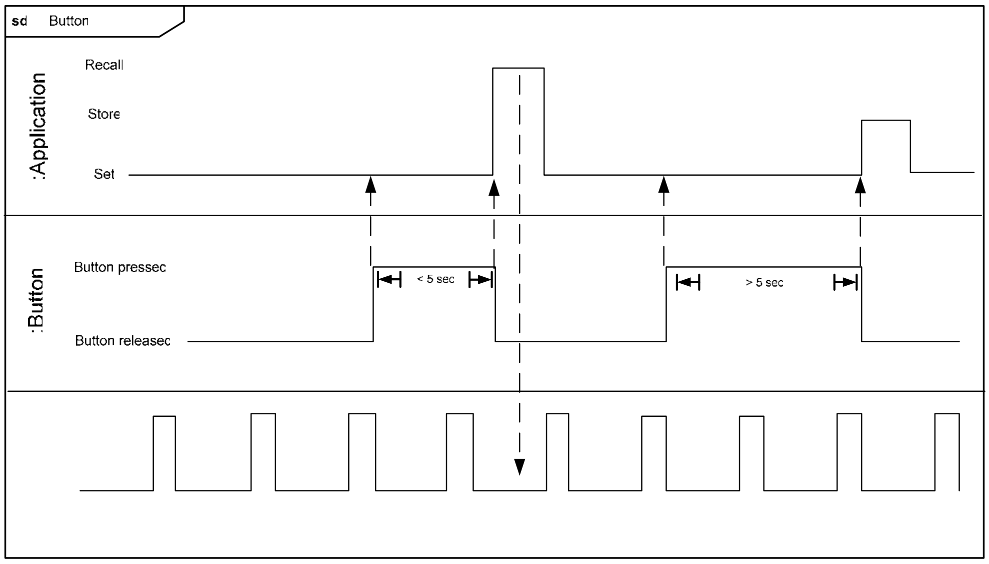


Figure 24 Timing Diagram – Button

5.4.2.4 Potentiometer

The *Potentiometer* component is responsible for reading in the actual position of the hardware potentiometer and providing it as a digital value on request. The specification class level diagram (see Figure 25) specifies that the *Potentiometer* component receives all its direct input via the *Driver* component. Communication relationships with components such as Controller are implicitly given since *Controller* is a super-component of *Potentiometer*.

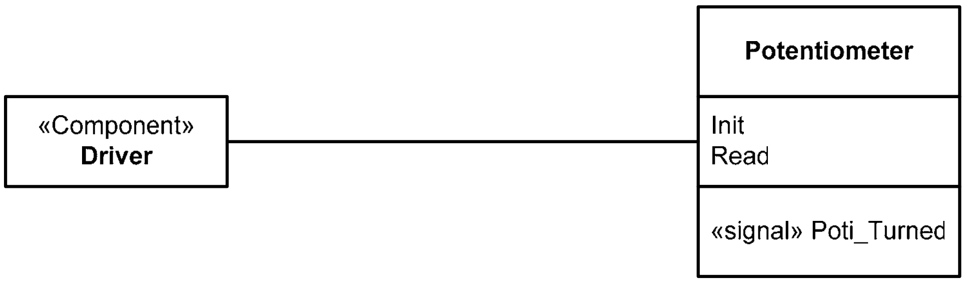


Figure 25 Class Diagram – Potentiometer (Specification)

In Figure 26 the state diagram concerning the *Potentiometer* component is presented. Basically the component has two different states an idle or standby state it is initially in, and a changed state which is entered as soon as the hardware potentiometer is moved (via an event from the controller). If this event occurs, the *Potentiometer* component will notify the *Application* (via Driver) by sending a signal. Any other events then will bring the component back to the *Standby* state. This is denoted by using the 'all' specifier.

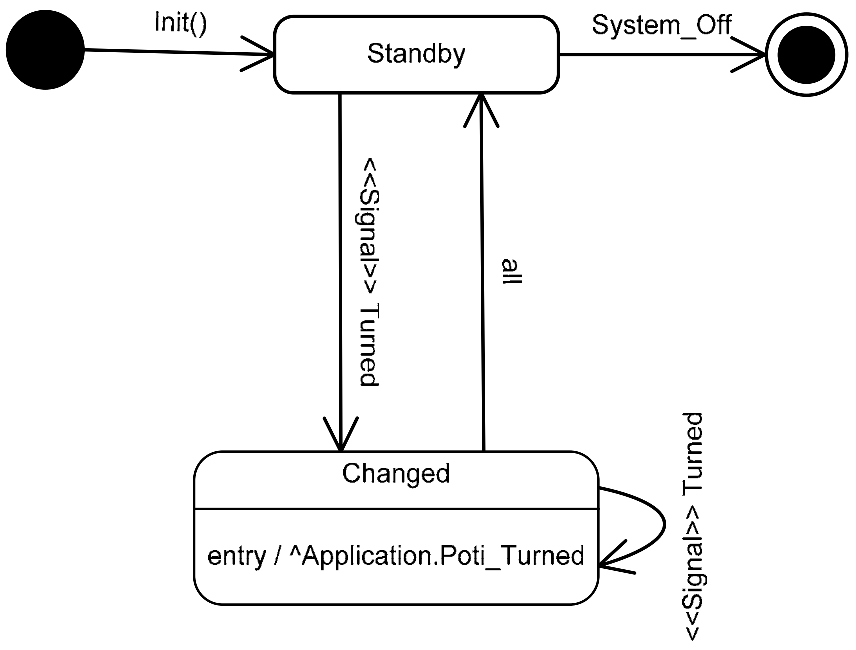


Figure 26 State Diagram – Potentiometer

Table 7 specifies Potentiometer's operations, as well as its received and send signals, by means of an operation specification.

Name	Init / Read / <<Signal>> Poti_Turned
Description	• Init: Initialize the system concerning the needs of the *Potentiometer* component. • Read: Return the actual value (turning degree) of each potentiometer to the caller by performing an ADC conversion. • <<Signal>> Poti_Turned: Notifies the application that the hardware potentiometer has been changed.
Constraints	The system is powered and the hardware poti's are connected to ports 23&24 of the microcontroller (see Figure 9).
Receives	The component receives a signal (i.e., *Turned*) from the controller as soon as the hardware potentiometer has been changed.
Returns	The Read operation returns two 10-Bit integer values (i.e., 0 -1024) representing the actual turning degree for the servo of the X- and Y-Axis.
Sends	As soon as the hardware potentiometer is moved the component sends the signal *Poti_Turned* to the *Application* component.

Reads	The Read() operation reads the turning degree of the hardware potentiometer by means of an ADC conversion.
Changes	NA
Rules	NA
Assumes	NA
Result	The hardware potentiometer was initialized and, upon request, the actual turning degree for both servos was returned to the caller. In addition the Application component was notified of any changes of the hardware potentiometer.

Table 7 Operation Specification – Potentiometer

The realization class diagram for *Potentiometer* (Figure 27) is similar to the corresponding specification class diagram (Figure 25), but has been refined concerning parameters and their type.

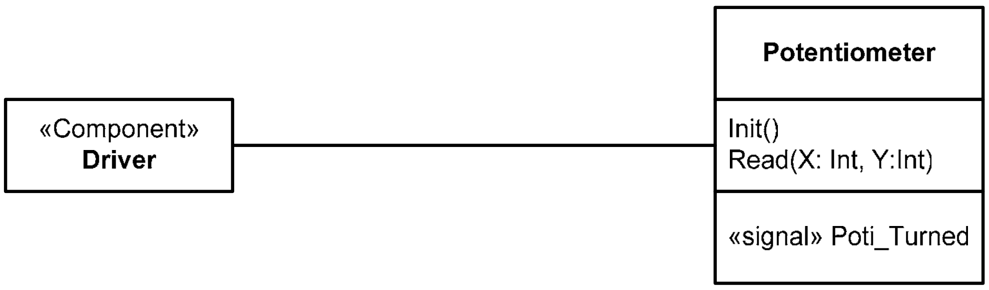


Figure 27 Class Diagram – Potentiometer (Realization)

The interactions of the Potentiometer component are specified using the sequence diagram shown in Figure 28. To ease understanding operation calls to *Potentiometer* are made directly via the *Application* component, neglecting that such requests are routed via the *Driver* component, although this is not entirely correct. Signals received from hardware components are directly modeled since the *Controller* is a super-component of *Potentiometer*.

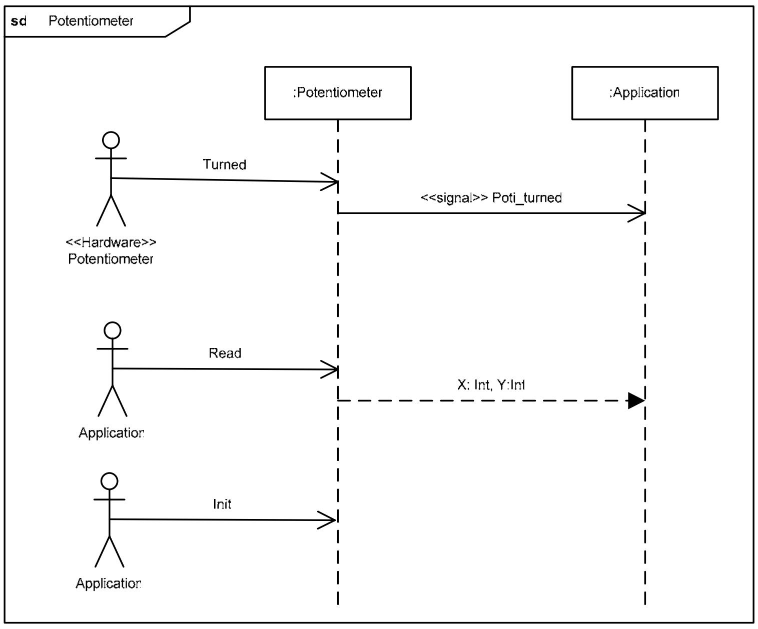


Figure 28 Sequence Diagram – Potentiometer

The activity diagram concerning the *Potentiometer* component is shown in Figure 29. After component initialization (e.g., initializing ports, etc.) the component waits for the *Turned* signal of the microcontroller, validates it (i.e., are the signals plausible and correct), and sends the *Poti_Turned* signal to the *Application* component. In addition, the component offers the *Read()* operation which return the actual turning degrees of the hardware potentiometers.

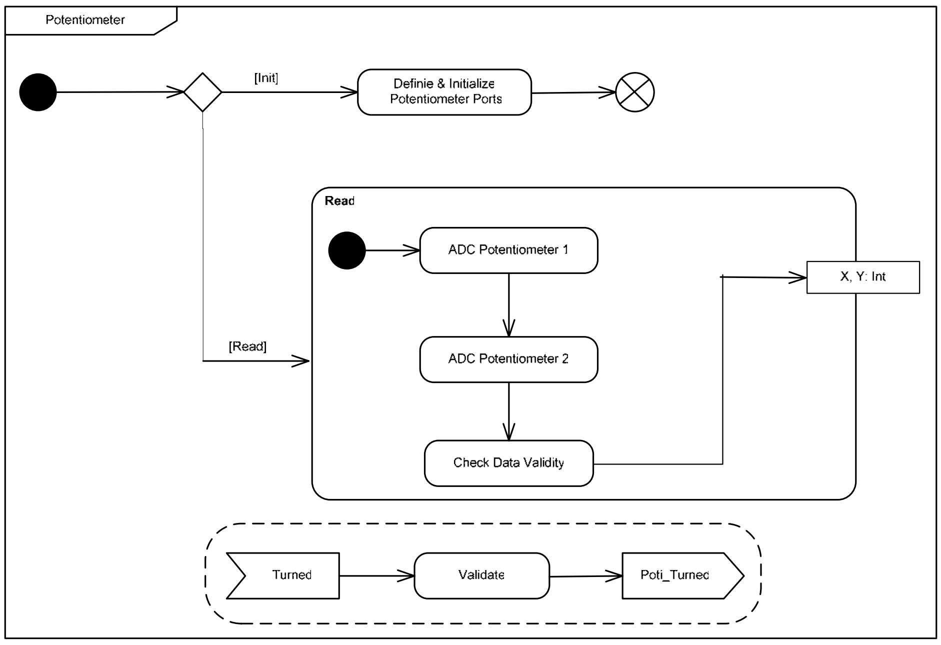


Figure 29 Activity Diagram – Potentiometer

5.4.2.5 Servo

The *servo* component is responsible for controlling the servos (i.e., concerning both mirror aptitudes) in the context of the Servo-Control system and therefore offers two operations to the outer world. The specification class level diagram (see Figure 30) specifies that the *Servo* component receives all its direct input via the *Driver* component, although in reality these may come from the *Application* component.

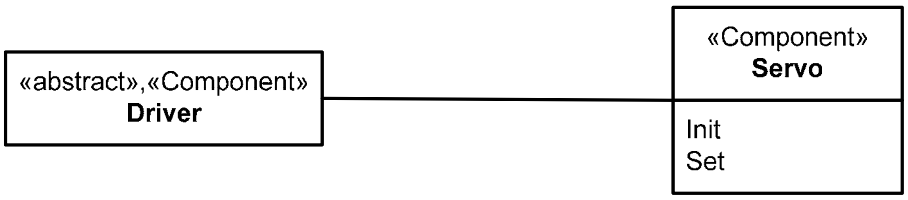


Figure 30 Class Diagram – Servo (Specification)

Name	Init / Set
Description	• <u>Init</u>: Initialize the system concerning the needs of the *Servo* component. • <u>Set</u>: Move the connected servos to a specific position specified by a 10-Bit integer value.
Constraints	The system is powered and the hardware servos are connected to ports 15 and 16 of the microcontroller (see Figure 9).
Receives	• <u>Init</u>: NA • <u>Set</u>: The set operation receives two 10-Bit integer values (i.e., 0-1024) representing the desired turning degree for the servo of the X- and Y-Axis.
Returns	NA
Sends	• Timer: Timer 1 and 2 of the *Timer* component are started and stopped.
Reads	Externally visible information accessed by the operation.
Changes	Externally visible information changed by the operation.
Rules	Rules governing the computation of the result.
Assumes	Precondition on the externally visible state of the komponent and on the inputs (in receives clause) that must be true for the komponent to guarantee the post condition (result clause).
Result	Strongest post condition on the externally visible properties of the komponent and the returned entities (returns clause) that become true after execution of the operation with true assumes clause.

Table 8 Operation Specification – Servo

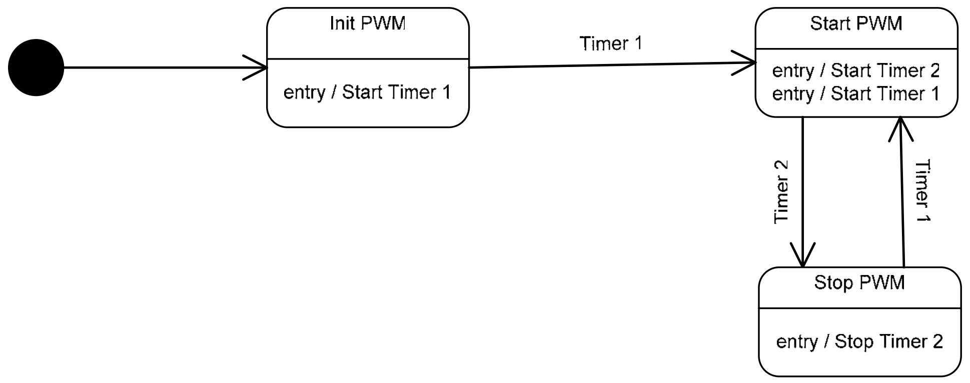


Figure 31 State Diagram concerning the PWM Signal generation

The realization class diagram for *Servo* (see Figure 32) is quite similar to the specification class diagram (Figure 30). One difference being, that the operation of the *Servo* component have been refined by adding parameters and data-types. In addition a use relationship to the *Timer* component has been added which is implicitly realized via both components super-component *Driver*.

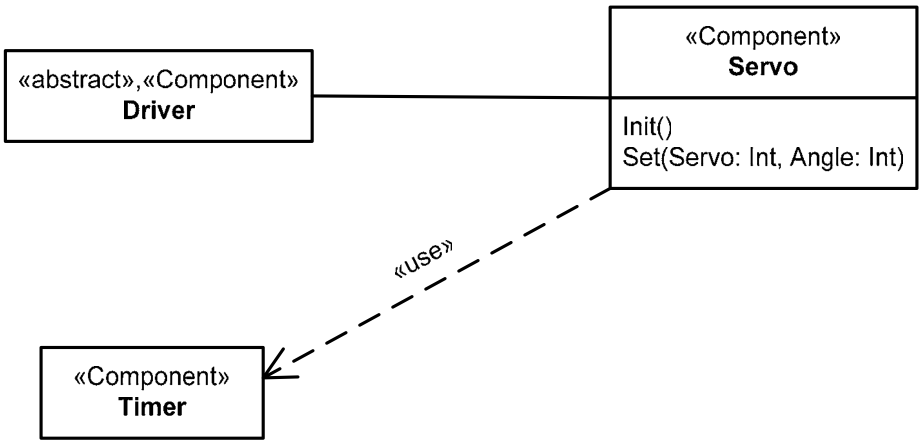


Figure 32 Class Diagram – Servo (Realization)

Figure 33 presents the flow of control within the servo component in form of an activity diagram. In addition, Figure 34 presents timing information of the servo.

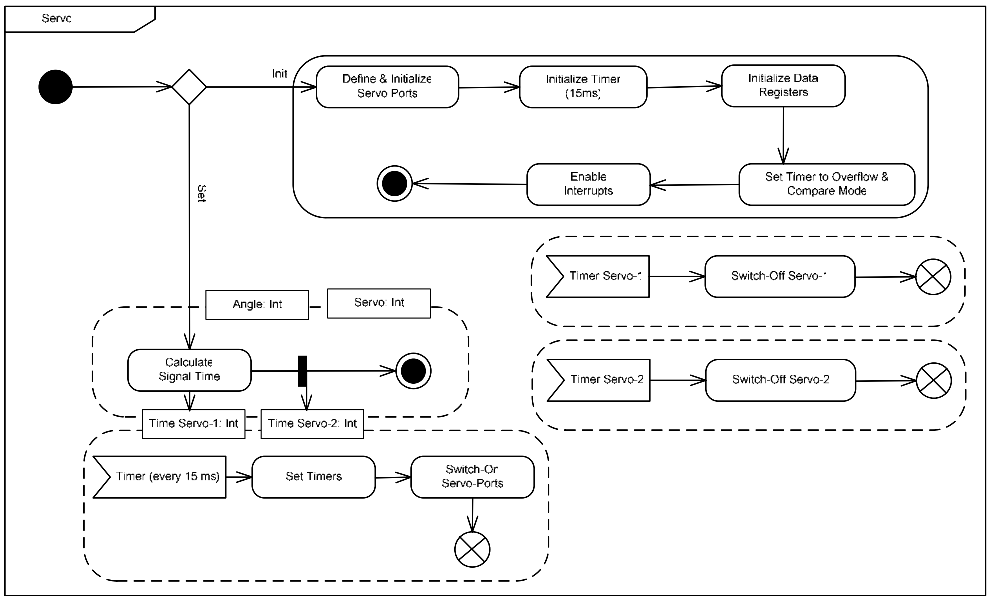


Figure 33 Activity Diagram – Servo

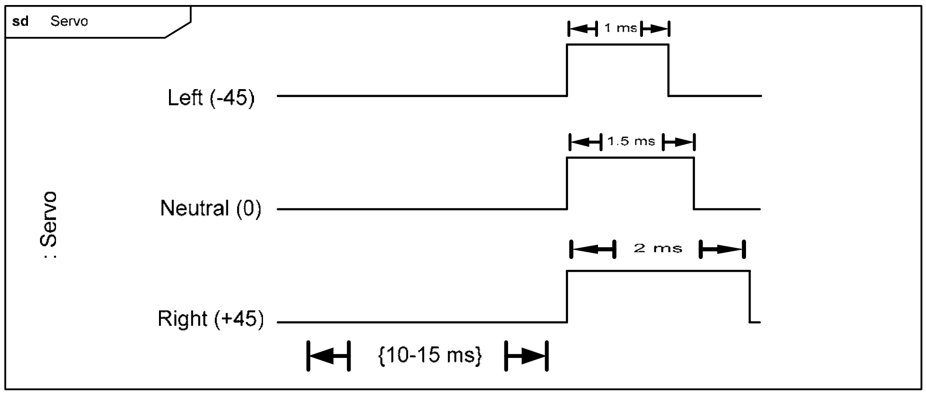


Figure 34 Timing Diagram – Servo

5.4.2.6 LCD

The LCD component (hardware) represents a standard module using a 8/4 Bit parallel interface[5], whereby the example system uses the 4-Bit variant in order to minimize the number of needed pins.

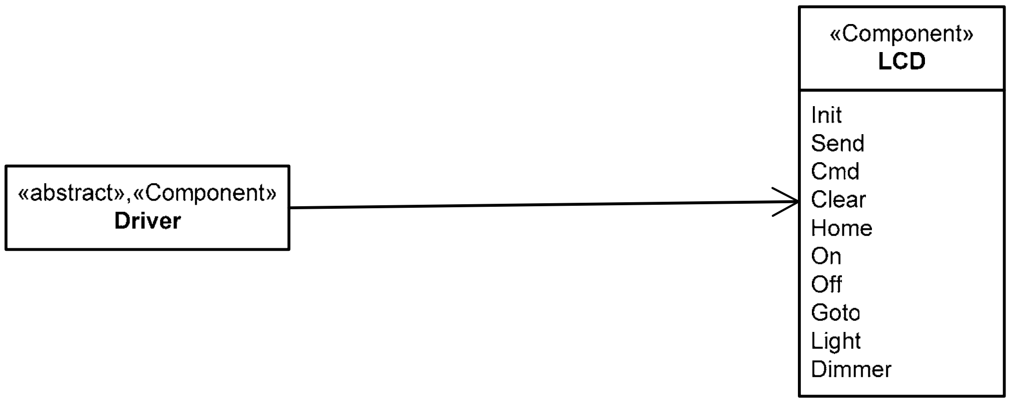


Figure 35 Class Diagram – LCD (Specification)

[5] In detail the module has 16 Pins: 8 data, 3 control, 3 connectors (VCC, Ground and contrast), 2 Backlight (anode, cathode).

Name	Name of the operation.
Description	Identification of the purpose of the operation, followed by an informal description of the normal and exceptional effects.
Constraints	The system is powered and the hardware LCD are connected to the specified ports of the microcontroller.
Receives	Information input to the operation by the invoker.
Returns	Information returned to the invoker by the operation.
Sends	Signals which the operation sends to imported komponents. These can be events or operation invocations.
Reads	Externally visible information accessed by the operation.
Changes	Externally visible information changed by the operation.
Rules	Rules governing the computation of the result.
Assumes	Precondition on the externally visible state of the komponent and on the inputs (in receives clause) that must be true for the komponent to guarantee the post condition (result clause).
Result	Strongest post condition on the externally visible properties of the komponent and the returned entities (returns clause) that become true after execution of the operation with true assumes clause.

Table 9 Operation Specification – LCD

The LCD hardware component uses a Hitachi HD44780 controller for controlling the display. The controller has its own set of command which are called by sending 8-Bit ASCII codes to the controller. This controller is slightly slower (2 MhZ) than the controller of the Servo-Control system. Therefore, the software driver of the Servo-Control system concerning the LCD component has to guarantee some timing rules. Basically after initialization the system has to wait 30ms before the first commands are accepted and executed. Furthermore, after sending a command to the LCD a 5ms pause is required (i.e., for executing the command) before the next command can be send. This is depicted by the state diagram (see Figure 36).

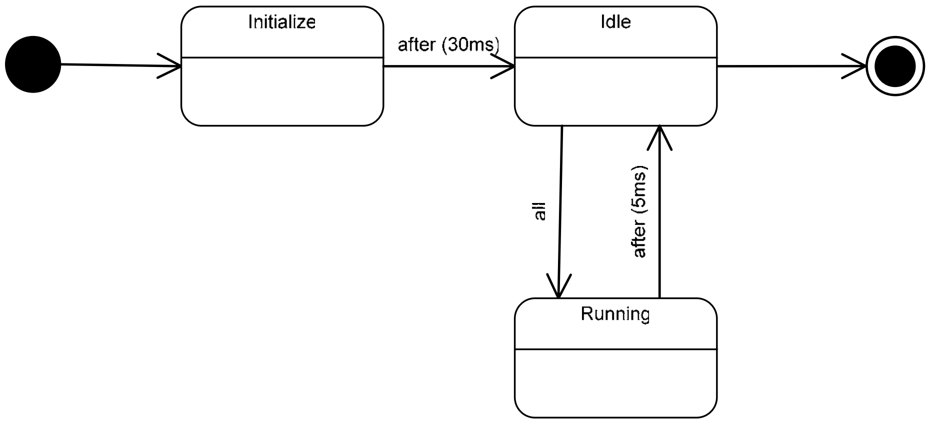


Figure 36 State Diagram – LCD

The realization class diagram for *LCD* is equal to the specification class diagram (Figure 35) since no additional structural details have been identified.

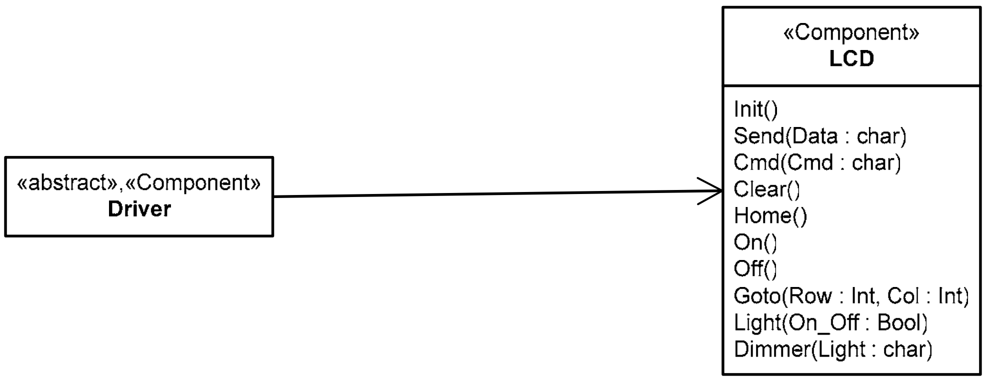


Figure 37 Class Diagram – LCD (Realization)

5.4.2.7 EEPROM

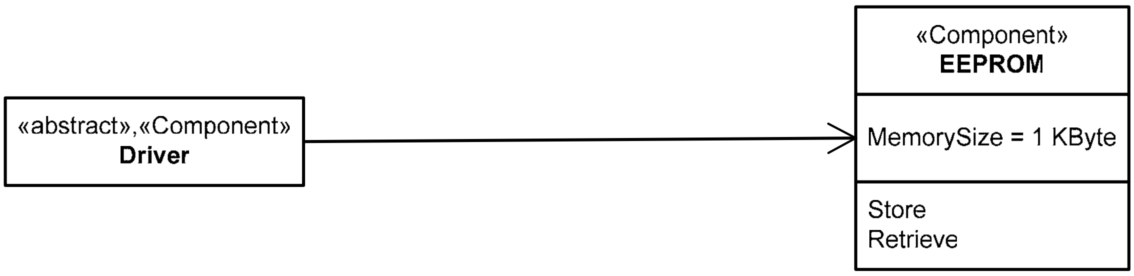


Figure 38 Class Diagram – EEPROM (Specification)

The state diagram for the *EEPROM* component is quite simple (see Figure 39). Basically the component is typically in the idle state and goes into the Read or Write state by calling the Store() or Retrieve() operation. Thus, the *EEPROM* component is only able to store or retrieve data sequentially and not in parallel.

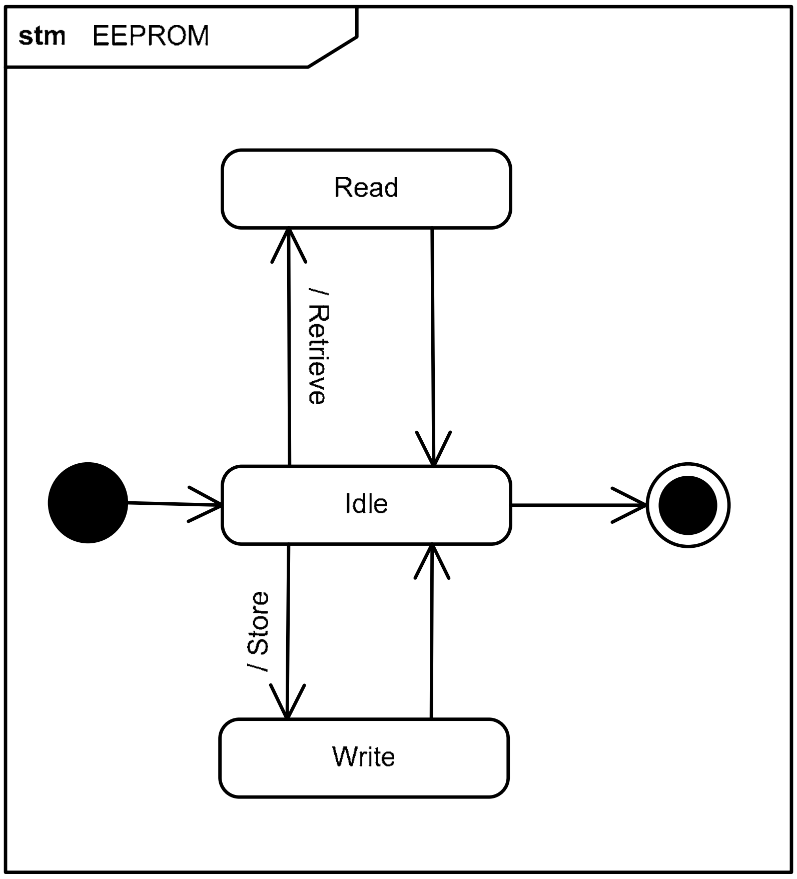


Figure 39 State Diagram – EEPROM

The operations of the *EEPROM* component are specified by the operation schemata of Table 10. It is important to note that there is a physical life-time limitation for EEPROM operations and that the operation assumes that there are stored values, which in turn requires a fail-safe handling by guaranteeing that always meaningful data is stored. Thus, within the realization this issue has to be addressed.

Name	Store / Retrieve
Description	• The Store operation supports the persistent storage of two integer values within the EEPROM of the microcontroller. • The Retrieve operation allows reading two stored integer values from the EEPROM of the microcontroller.
Constraints	The EEPROM allows approx. 100.000 storages before it is physically damaged.
Receives	• Store: X,Y : Integer • Retrieve: --
Returns	• Store: -- • Retrieve: X, Y: Integer
Sends	--
Reads	The Retrieve operations reads two integer-values from the EEPROM.
Changes	The Store operation changes the two stored integer values within the EEPROM to the new values it has received .
Rules	--
Assumes	Prior to the first invocation of Retrieve default values were stored.
Result	• Store: Two integer values have been persistently stored. • Retrieve: Two integer values have been read from the EEPROM and returned.

Table 10 Operation Specification – EEPROM

The realization class diagram for *EEPROM* (Figure 40) is quite similar to the specification class diagram (Figure 38). The only difference that the operation signatures are more precise due to parameter and type definitions.

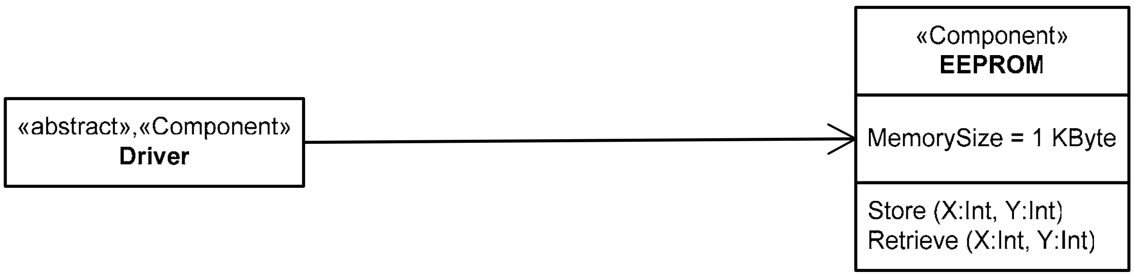


Figure 40 Class Diagram – EEPROM (Realization)

The *EEPROM* component does only communicate with its caller, and does not need any additional external communication relationships. Therefore, the components' sequence diagram is simple (see Figure 41).

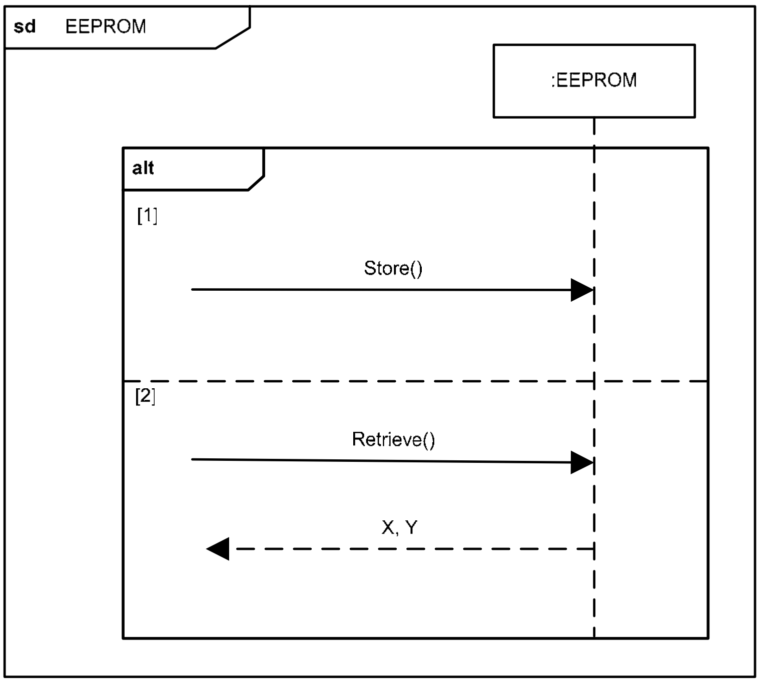


Figure 41 Sequence Diagram – EEPROM

The activity diagram for the *EEPROM* component (Figure 42) describes the flow of control/algorithm of its operations. Within the Retrieve operation the constraint defined in the operation specification (Table 10), concerning the initialization of values, has been realized.

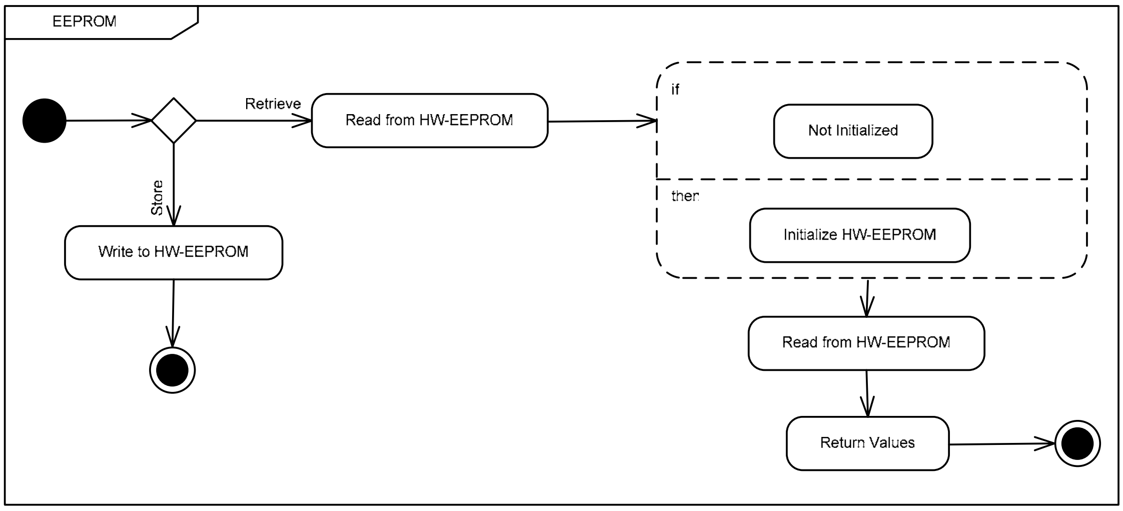


Figure 42 Activity Diagram – EEPROM

5.4.2.8 Timer

The *Timer* component is used for measuring time between different events. Since it must be generally applicable timers are started and stopped via operation calls. Figure 43 shows the specification level class diagram of the *Timer* which receives its operation calls via the *Driver* component. Of course *Driver* is a container component without specific operations which reroutes calls from the *Application* component directly to the timer.

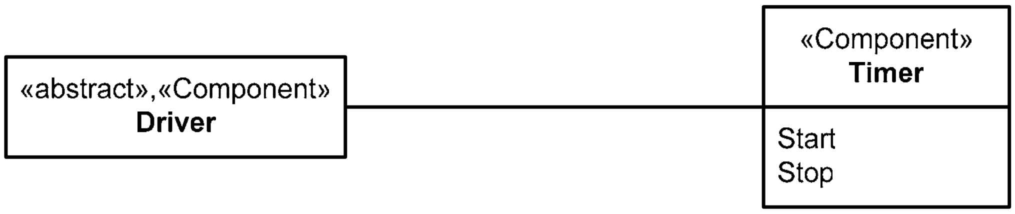


Figure 43 Class Diagram – Timer (Specification)

The state diagram of Figure 44 describes the externally visible states of *Timer* and the transitions between them. Basically the *Timer* is in the *Idle* state at start-up and moves to the *Running* state as soon as the *Start()* operation is called, and stays there until the *Stop()* operation is called. By entering the *Idle* state the counter is set back to zero in order to be prepared for the next run. It is important to note that the counter cannot be started again if running and that overflows are handled as self-transitions. The latter is needed in order to count the number of overflows.

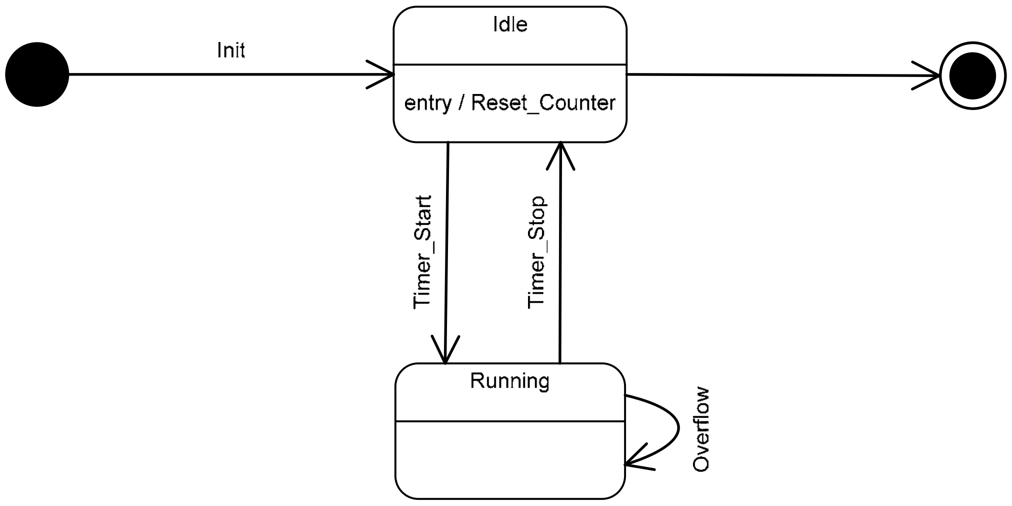


Figure 44 State Diagram – Timer

Table 11 specifies the operation schemata for the *Timer* operations. In addition to a textual description it provides formulas and technical background information according the use of timers. At this time an attribute of the *Controller* component that is visible to all sub-components becomes important because clock speed has a major impact on time measurement.

Name	Start / Stop
Description	This operation starts or stops the timers of the microcontroller. When a timer is stopped a time value is returned, describing the runtime period of the timer. In addition, the timer registers of the microcontroller are reset.
Constraints	Given that we are interested in timing periods between 1 - 5s, and that the controller is running on 4Mhz using an 8Bit counter, there are 15 timer overflows every millisecond. Thus, a pre-scaler (e.g. 1024) should be used. This results in an overflow rate of 65.6 ms.
Receives	• Every operation receives a unique identifier which of the four different timers is addressed. • The *Start()* operation receives a *Timer_Value* which specifies the value at which the timer sends a signal.
Returns	The stop operation returns an integer value describing the runtime of the timer in milliseconds.
Sends	Signals concerning the reach of the predefined threshold value (i.e., *Timer_Value*) are send to the *Servo* component.
Reads	--
Changes	--
Rules	• Time is calculated by using an controller-internal 8BIT timer by counting overflows and timer values. In detail the formula can be defined as: $Duration = \left(\sum Overflows\right) * 65 + Timer\_Value$ • Multiplying the sum of overflows with 65 adds a little impreciseness, however this is tolerable since using floats increase memory consumption. • The *Timer_Value* for the timer used by the *Application* component is preset to 6 secs, since *Application* is interested in time ranges of 0-5.1 secs.
Assumes	The timer will not run for more than 65000 ms (65sec). Otherwise the data type of the return value has to be changed.
Result	The time between the start and stop event has been measured and returned to the caller. Furthermore timing signals needed by the *Servo* component were sent.

Table 11 Operation Specification – Timer

The realization class diagram for *Timer* (see Figure 45) is quite similar to its specification class diagram (see Figure 43). The only difference being that the

Stop operation returns an integer value depicting how long the timer had run. Furthermore, two new operations (identified within the state diagram) were added that allows the timer to be initialized and reset. The realizations of the *Application* and *Servo* component require that the *Timer* component should handle at least four different timers[6]. Thus, the *Start, Stop* and *Reset* operation allow selecting a specific timer. This in turn, has to be reflected in the implementation of the *Application* and the *Servo* component (→ embodiment).

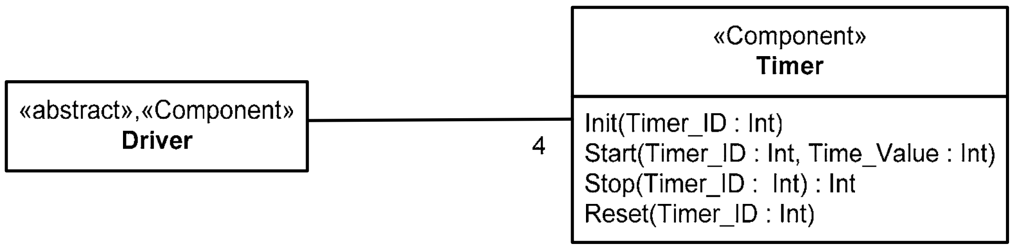


Figure 45 Class Diagram – Timer (Realization)

The realization class diagram of Timer does not show any newly identified subcomponents or classes (i.e., *Time* is a leaf component). This is also supported by the sequence diagram (see Figure 46) which clearly indicates that *Timer* does not need to cooperate with other components to fulfill its task. In general, the sequence diagram, except concerning the Stop() operation, is therefore superfluous and can be neglected.

[6] This is also denoted by setting the multiplicity of *Timer* to four.

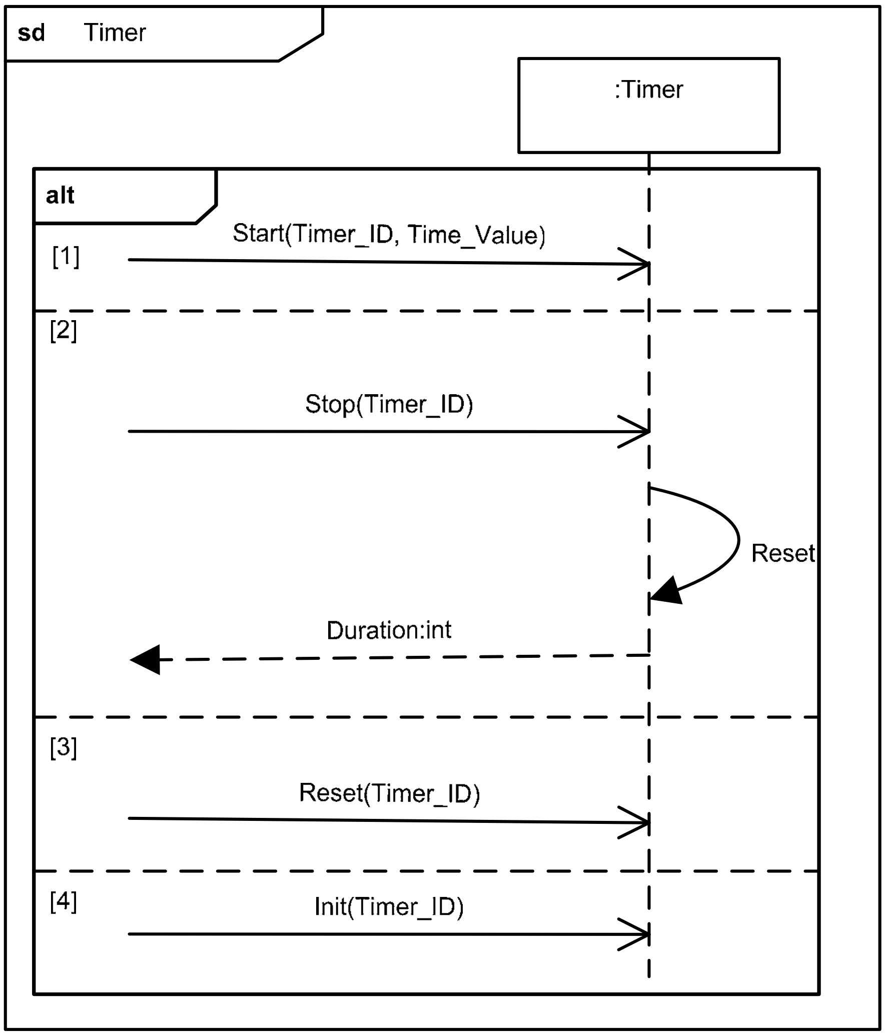


Figure 46 Sequence Diagram – Timer

The algorithmic behavior of the *Timer* component is specified by means of two activity diagrams (see Figure 47 and Figure 48). In this context, it was easy to see that *Reset()* is an internal operation which is only used in the context of the *Stop()* operation. Therefore, *Reset()* does not need its own activity diagram.

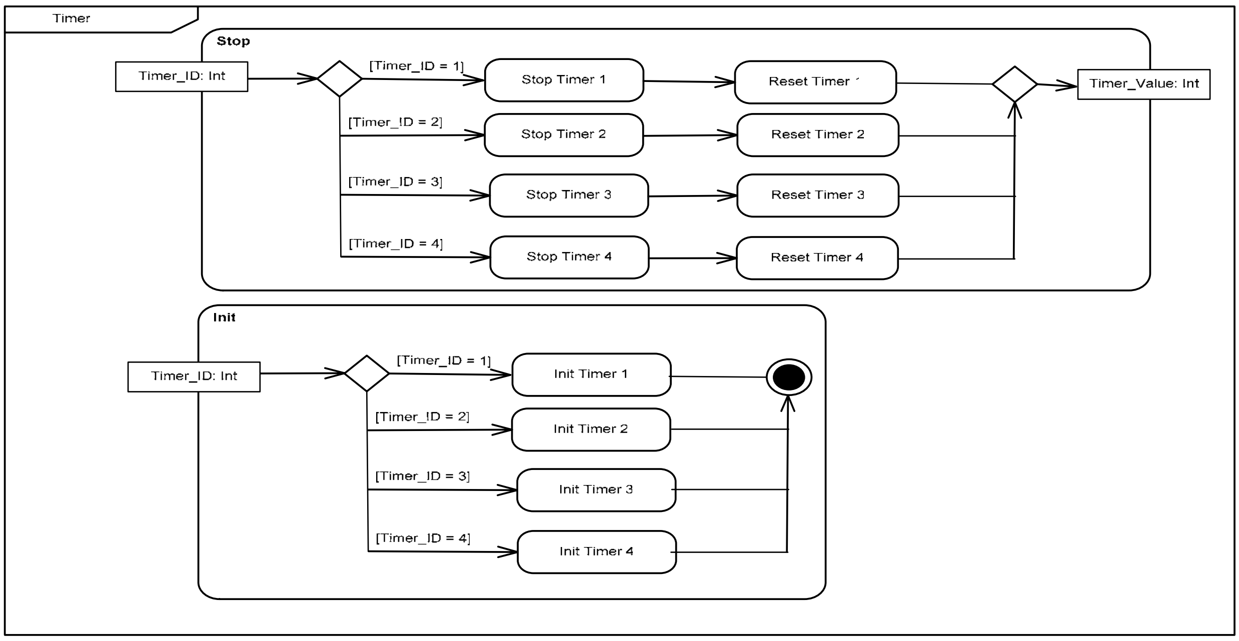


Figure 47 Activity Diagram – Timer

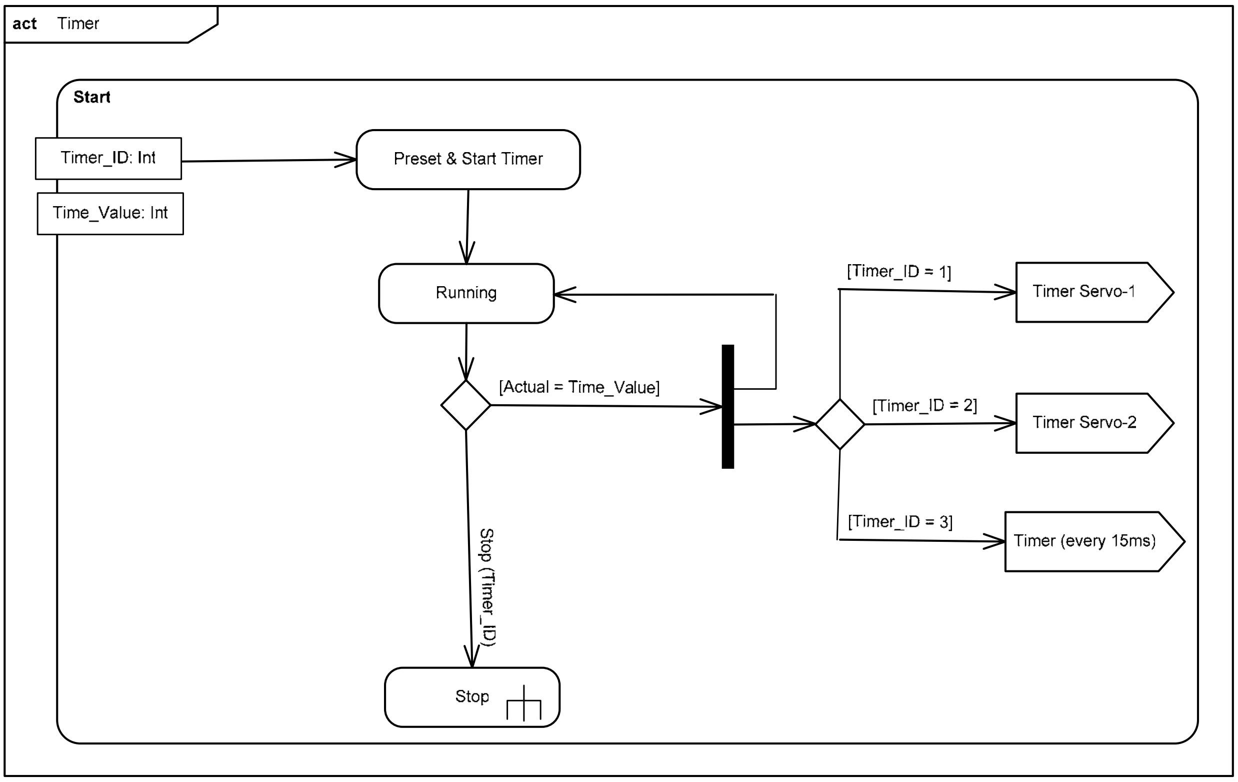


Figure 48 Activity Diagram – Timer

5.4.3 Embodiment & Model Transformation to C

The general goal of the MDA approach to software development is to reduce time-to-market by automating the mapping of models to code. Therefore, the models and artifacts, generated by the component engineering activities, describe the properties of components at a level akin to analysis and design. Before any running software can be deployed, these must be transformed into semantically equivalent executable forms. In practice this involves a manual step, in which the artifacts are translated into an intermediate representation amenable to processing by automatic tools (e.g. source code), and an automated step in which these tool are applied to generate the truly executable (e.g. binary) forms. In common with prevailing terminology, this intermediate representation is termed an implementation, and the manual activity responsible for transforming a realization into an implementation is known as the implementation activity.

The only constraint on a component implementation is that it faithfully embodies the semantics of the component realization in a way that conforms to the non-functional requirements. Any implementation technology capable of satisfying this constraint can be used to implement components, including high-level programming languages (e.g. Java, C++, C, Ada, Python), physical component technologies (e.g. JavaBeans, EJBs, COM, CORBA), and databases. Because of the properties of components, physical component technologies are likely to provide the most flexible and natural implementation of KobrA components, followed by programs written in object-oriented programming languages.

Although a component's implementation can be thought of as being closely coupled to its realization, it is not regarded as being part of the component containment hierarchy. Component implementations exist in a separate dimension, orthogonal to the component containment hierarchy. This dimension is concerned with the abstraction level and representation form of components rather than their containment relationships. The orthogonality of these dimensions is important because it means that, in general, any process capable of producing a faithful implementation of a component can be used in conjunction with the KobrA component modeling activities. However, KobrA defines its own recommended implementation activity that fits in with the spirit of simplicity and systematic development that underpins KobrA. This is the topic of this chapter.

Within the context of the MARMOT method, the logical component hierarchy (see Figure 10) is first mapped to the resulting physical component hierarchy prior to its translation to source-code. This is the goal of the embodiment step. The resulting physical hierarchy, as shown in Figure 49, is significantly smaller than the logical one. All hardware related components have been removed since the MARMOT development focuses on the software part. Furthermore, the *Driver* component, which acted as a 'forwarder', does not provide own functionality and can therefore be left out. Finally the *Application* component has been moved up in the component hierarchy in order to allow direct accesses to the hardware-access components.

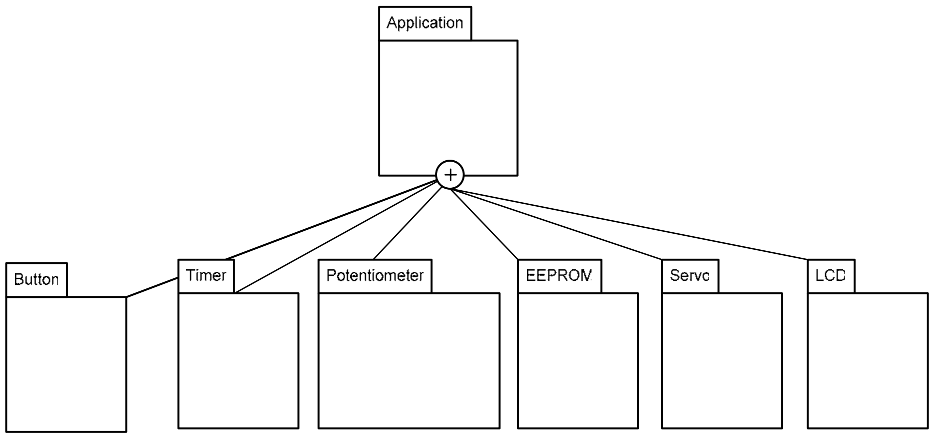


Figure 49 Containment Hierarchy – Physical

Concerning implementation, the physical containment hierarchy can nicely be mapped to source code structure. Figure 50 shows the hierarchy of C body and header files as well as their dependencies. The *Application* component, comprising the overall system functionality, contains the *main()* function. A deployment model is not needed since it is a one-node (processor) system

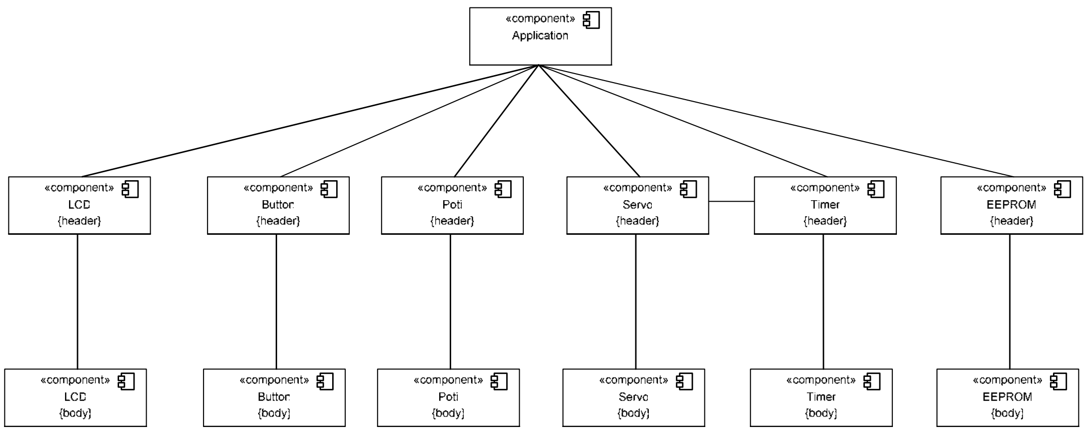

«component» Application
«component» LCD {header}
«component» Button {header}
«component» Poti {header}
«component» Servo {header}
«component» Timer {header}
«component» EEPROM {header}
«component» LCD {body}
«component» Button {body}
«component» Poti {body}
«component» Servo {body}
«component» Timer {body}
«component» EEPROM {body}

Figure 50 Source-Code Structure

The final translation step then results in the associated C source-code (e.g., see Figure 51 for an excerpt). The translation was done by applying the mapping-rules provided in [13] which were found to be more efficient than other tool-based generators (e.g., Ameos or Rose Real-Time).

```
#include "servo.h"

volatile int servo1, servo2;

SIGNAL(SIG_OUTPUT_COMPARE1A)
{
 //End Servo Pulse 1
 PORTB |= (0<<DDB1);
}

SIGNAL(SIG_OUTPUT_COMPARE1B)
{
 //End Servo Pulse 2
 PORTB &= ~(1<<DDB2);
}

SIGNAL(SIG_OVERFLOW1)
{
 // Trigger every 16ms
 int srv;

 PORTB |= (1 << DDB1);
 PORTB |= (1 << DDB2);

//Servo1
 srv = servo1 + 6000;
 OCR1AH = srv >> 8;
 OCR1AL = srv & 0xFF;

 //Servo 2
 srv = servo2+6000;
 OCR1BH = srv >> 8;
 OCR1BL = srv & 0xFF;

 // Reset Counter
 TCNT1H = 0x00;
 TCNT1L = 0x00;
}
```

```
void Init_Servos()
{
DDRB |= (1 << DDB1);
DDRB |= (1 << DDB2);
PORTB &= ~(0<<DDB1);
PORTB &= ~(1<<DDB2);
TCCR1A = 0x00;
TCCR1B |= (1<<CS10);
TCNT1H = 0x00;
TCNT1L = 0x00;
servo1 = servo2 = 0;
TIMSK |= ((1 << OCIE1A) | (1 <<
OCIE1B) | (1 << TOIE1));
TIMSK |= ((1 << OCIE1B) | (1 <<
TOIE1));
sei();
}

void Set_Servo(unsigned char
servo, unsigned int angle)
{
 int intermediate;

 intermediate = ((angle * 4) -
2000);
 if (intermediate < -2000)
   intermediate = -2000;
 if (intermediate > 2000)
   intermediate = 2000;
 if (servo == 1)
   servo1 = intermediate;
 else
   servo2 = intermediate;
}
```

Figure 51 Source Code – Servo Component

5.4.4 Testing and Debugging

As soon as executable code exists it can be tested to check if all requirements are fulfilled. In the context of micro-controller based systems testing can be done in different steps. In principle one can distinguish between:

- Debugging to correct errors and check the general behavior. This is simply done by compiling the system and using the inbuilt simulator of AVRStudio. Therefore, we do not have to transfer the binary to the flash memory of the controller, and are able to step throughout the code and to manually manipulate registers of the processor.

- In a second step, the binary is flashed on the micro-controller and the system is tested by using its user-interface (e.g., button, potentiometer, etc.) and checking the corresponding behavior and output on the LCD display.
- The previous steps are unsystematic in that the do not use defined test-cases or follow specific procedures. Therefore, the third testing step requires the definition of test cases (within the requirements and design phase) in order to obtain systematic and repeatable tests.

5.5 Follow-Up Projects

Since the effects of reuse can only be measured and analyzed in follow-up projects, a number of student development assignments, using the original mirror system as a basis, were defined and carried out. The goal of these assignments was to cover typical reuse situations such as those defined in [5]. In detail:

- The system was ported to different hardware platform whereby its functionality kept unchanged. First, the system was ported to a processor of the same family but with different characteristics (i.e., ATMega32, 32Kb Flash, 2Kb RAM, 1Kb EEPROM, 16MhZ). Second, the system was ported to a different processor family (i.e., PICF, 7Kb Flash, 192Byte RAM, 128Byte EEPROM, 20MHz).
  In practice, porting a component implementation to other micro-controller platforms can often be automated at the code-level using high-level programming languages and advanced compilers. However, porting is a common reuse scenario that, following the ideas of the Model-Driven Architecture (MDA) [22], should also be supported on the model level.
- The mirror-system was adapted according to changed system requirements. In a third project (aka Adapt-), the functionality of storing and recalling position data was removed and the system had to be adapted according to this situation. In a fourth project (aka Adapt+) the mirror-system was extended by a defreeze/defog functionality using a humidity sensor and a heater.
- The mirror systems (parts thereof) were reused in the context of a different system (aka Door project). In detail, a door-control system was developed that controls the window and mirror of a door, including “puddle lights”. This system was realized using three micro-controller boards (i.e., central control, window- and mirror control), and serial-line inter-board communication.

The projects were performed by students, applying MARMOT and (re)using the mirror system. As development tools the students used Rational Rose [26] or ARGO UML [1] for modeling and the AVRStudio/GCC environment [3] for programming and debugging. The mapping of models to code was performed manually without using automatic code generation facilities of software engineering tools.

The first impression of the participants was that reuse on both, the model and the implementation level, in the context of a MARMOT project is simple and straightforward. In addition, the participants believed that they completed their projects earlier than expected. However, these are the subjective statements of students that must be confirmed by objective measurement data.

6 Results

In the context of these small case studies, a number of measurements was performed in order to get a first impression if the maintainability, portability, and adaptability of software systems, developed with MARMOT, can be improved. Table 12 provides data concerning model and code size as well as data on quality- (e.g., number of defects), and process measures (e.g., effort). The metrics thereby follow the definitions of section 3.2.

At a first glance, the number of diagrams seems to be quite high for systems of that size. By modeling components at the specification and realization level, a component might be described by several diagrams including textual specifications. However, due to MARMOT's modeling principles, there is an overlap of diagrams between levels (e.g., structural diagrams), which reduces model complexity significantly.

It is interesting that porting the system to another hardware platform required only minimal changes to the models (e.g., UML hardware representation, ports, etc.). Thus, MARMOT supports the MDA idea [22] of platform independent modeling. Only in the embodiment step models become platform specific. The ease of porting a system to different platforms is also supported by the high amount of reuse with minimal changes, the low effort, and the low number of defects.

Concerning the adaptation of existing systems by adding or removing functionality, the data of Table 12 reveals that MARMOT provides sufficient support. First of all a large proportion of the systems could be reused from the original system. Second, in comparison to the initial development project (i.e., 'Original'), the effort for adaptation is low (26hrs vs. 3/10hrs). In addition, the quality of the system profits from the quality assurance activities carried out in the initial component development. Thus, the promises of component-oriented development concerning time-to-market and quality could be confirmed in this case-study.

Interesting to note is that the effort for the initial system corresponds to standardized effort distributions over development phases as used by common cost estimation methods, whereby the effort for the variants is significantly lower. In addition, this supports the assumption that component-oriented development has an effort-saving effect in subsequent projects.

In general, porting and adaptation of a component-based system takes place during developing system variants. Thus, the systems are highly similar, which, in turn, explains why reuse works that well. It would therefore be interesting to look at larger systems (of the same domain) that reuse the original system as a whole and/or some of its components. The 'Door' project is such a project. When looking at Table 12 it indicates that 60% of the overall system was re-used in form of the mirror system, which itself did not need major adaptations. The effort and defect density is higher than those of the mirror system variants due to the development of new additional components, major hardware extensions, and intensive quality-assurance. Thus, when directly compared to the initial effort and quality (i.e., the mirror system), a positive trend can be seen that supports the assumption that with MARMOT embedded systems can be quickly developed at a low cost but with high quality.

		Original	ATMega32	PICF	Adapt-	Adapt+	Door
LOC		310	310	320	280	350	490
Model Size (Abs.)	NCM	8	8	8	6	10	10
	NCOM	15	15	15	11	19	29
	ND	46	46	46	33	52	64
Model Size (Rel.)	$\frac{\mathit{NumberofStateCharts}}{\mathit{NumberofClasses}}$	1	1	1	1	0.8	1
	$\frac{\mathit{NumberofOperations}}{\mathit{NumberofClasses}}$	3.25	3.25	3.25	2.5	3	3.4
	$\frac{\mathit{NumberofAssociations}}{\mathit{NumberofClasses}}$	1.375	1.375	1.375	1.33	1.3	1.6
Reuse	Reuse Fraction(%)	0	100	97	100	89	60
	New (%)	100	0	3	0	11	40
	Unchanged (%)	0	95	86	75	90	95
	Changed (%)	0	5	14	5	10	5
	Removed (%)	0	0	0	20	0	40
Effort (h)	Global	26	6	10.5	3	10	24
	Hardware	10	2	4	0.5	2	8
	Requirements	1	0	0	0.5	1	2
	Design	9.5	0.5	1	0.5	5	6
	Implementation	3	1	3	0.5	2	4
	Test	2.5	2.5	2.5	1	2	4
Quality	Defect Density	9	0	2	0	3	4

Table 12 Case-Study Results

In summary the small case-studies presented in this paper show that the promises of component-oriented development concerning, reuse, effort, and quality can also be achieved in the context of embedded system development. However, similar to experiences with product-line engineering projects [5], component-based development projects require an upfront investment before they pay-off.

There are some threats to the validity of these ob-served results which may hinder their generalization. First, the people participating in this study were students, and they may not be representative for software professionals. However, the results may also be useful in an industrial context, since engineers in industry have often no more experience in model-based development with UML than students. After all, introducing such methodological support requires a steep training curve, even in professional organizations. Second, the use of volunteers may affect the validity of the study (i.e., selection bias). Individuals who volunteer for an activity, people who like something, are almost certainly different from those who do not volunteer. Volunteers are, by definition, motivated to participate and presumably expect to receive some benefit from the intervention. In companies it is often the case that employees have a negative attitude toward new technology simply because they are afraid that the new technology takes too much of their valuable time. These differences between study participants and people in real organizations limit the ability to generalize the results of this study beyond the research sample. Finally, the systems developed in the scope of this paper may not be representative in terms of their size and complexity. However, the results of this study can be used as a trend indicating possible benefits and therefore allows defining industrial-scale case-studies.

7 Summary and Conclusions

The growing interest in the Unified Modeling Language provides a unique opportunity to increase the amount of modeling work in software development, and to elevate quality standards. UML 2.0 promises new ways to apply object/component-oriented and model-based development techniques throughout embedded systems engineering. However, this chance will be lost, if developers are not given effective and practical means for handling the complexity of such systems, and guidelines for systematically applying them.

This paper has outlined the UML modeling practices, which are needed in order to fully leverage the component paradigm in the development of software for embedded systems. Following the principles of encapsulation and uniformity, and describing both levels with a standard set of models – it becomes feasible to model hardware and software components of an embedded system with UML. This facilitates also a "divide and conquer" approach to modeling, in which a system unit can be developed independently. It also allows new versions of a unit to be interchanged with old versions provided that they do the same thing.

To validate MARMOT, a series of case-studies was performed. Quantitative and qualitative results of these studies indicate that MARMOT supports systematic reuse and thereby reduces development effort, and improves the quality of a software system. However, these results are only a starting point for more elaborate validation and generalization of the results. Therefore, a controlled experiment with a larger system is currently planned in order to obtain more, and more objective data. The next step will then be the provision of a tool.

References

[1] ArgoUML Homepage: http://argouml.tigris.org/

[2] Atkinson, C., Bayer, J., Bunse, C., et al. Component-Based Product-Line Engineering with UML, Addison-Wesley, UK, 2001

[3] AVR Studio, Atmel Corp. http://www.atmel.com

[4] Bunse, C., Gross, H.G., Unifying Hardware and Software Components for Embedded System Development, In: Architecting Systems with Trustworthy Components, R. Reussner, J.A. Staffort, C.A. Szyperski (Eds), Lecture Notes in Computer Science, Vol. 3938, Springer, Heidelberg, 2006.

[5] Cohen, S.. Predicting when Product Line Investment Pays. Proc. of the Second International Workshop on Software Product Lines: Economics, Architectures, and Implications, pages 15–18, 2001.

[6] Crnkovic, I., Larsson, M. (Eds.), Building Reliable Component-Based Software Systems, Artech House, 2002

[7] Douglass, B.P., Real-Time Design Patterns. Addison-Wesley, 2003

[8] Fritzson, P., Principles of Object-Oriented Modeling and Simulation with Modelica2.1, Wiley, 2004

[9] Gross, H.G., Component-Based Software Testing with UML. Springer, Heidelberg, 2005.

[10] Harel, D., Lachover, H., Naamad, A. et al. Statemate: A working environment for the development of complex reactive systems. IEEE TSE, 16(4), April 1990.

[11] Heck, B., Wills, L., Vachtenavos, G., Software technology for Implementing Reusable, Distributed Control Systems, IEEE Control Systems magazine, February, 2003

[12] Hooman, J., Towards Formal Support for UML-based Development of Embedded Systems, Proc. of the 3d PROGRESS Workshop on Embedded Systems, Technology Foundation STW, 2002

[13] Khan, M.U., Geihs, K., Gutbrodt et al Model-Driven Development of Real-Time Systems with UML 2.0 and C, 3rd International Workshop on Model-

based Methodologies for Pervasive and Embedded Software at the 13th IEEE Int. Conf. on Engineering, 2006

[14] Kim, H., Boldyreff, C., Developing software metrics applicable to UML models. Proc. of the 6th ECOOP Workshop on Quantitative Approaches in Object-oriented engineering, Malaga, Spain, June 2002.

[15] Lange, C.F.J., Model Size Matters, Workshop on Model Size Metrics 2006 (co-located with the ACM/IEEE MoDELS/UML Conference); October, 2006.

[16] Lano, K. Formal Object-Oriented Development. Springer, 1995.

[17] Lavagno, L., Martin, G., Selic, B. (Eds.) UML for Real Design of Embedded Real-Time Systems, Kluwer Academic Publishers, 2003

[18] Li,J., Conradi, R., Mohagheghi, P., et al., A Study of Developer Attitude to Component Reuse in Three IT Companies, 5th Int. Conference Product Focused Software Process Improvement, PROFES 2004, Japan, 2004

[19] Marcos, M., Estévez, E., Gangoiti, U., et al., UML Modeling of Industrial Distr. Control Systems, Proc. of the 6th Portuguese Conf. on Automatic Control, Portugal, 2004

[20] Marwedel, P. Embedded System Design (Updated Version), Springer, 2006

[21] The MathWorks, Inc., Simulink Reference, 2005. http://www.mathworks.com

[22] Miller, J., Mukerji, J.: MDA Guide 1.0, omg/03-05-01, 2003 (http://www.omg.org/)

[23] Mills, H.D., Basili, V.R., Gannon, J.D. et al. Principles of Computer Programming: A Mathematical Approach. Allyn and Bacon Inc., 1987.

[24] Mockus, A., Fielding, R.T., Herbsleb, J., A Case Study of Open Source Software Development: The Apache Server, Proc. of the 22nd International Conference on Software Engineering, Limerick Ireland., 2000

[25] Object Management Group, Inc. UML 2.0 Superstructure Specification, OMG document formal/05-07-04, 2005, http://www.omg.org/cgi-bin/doc?formal/05-07-04.

[26] Rational Rose, http://www.rational.com

[27] Selic, B., Gullekson, G., Ward, P. T., Real-Time Object-Oriented Modeling, John Wiley & Sons, 1994

[28] Szyperski, J., Component Software. Beyond Object-Oriented Programming, Addison-Wesley, 2002

[29] Ventura, J., Siebert, F., Walter et al., HIDOORS – A High Integrity Distributed Deterministic Java Environment, Proc. of the 7th Int. Workshop on Object-Oriented Real-Time Dependable Systems, USA, 2002

Document Information

Title:	Developing µController-Systems with UML: A MARMOT Case Study
Date:	September 18, 2006
Report:	IESE-111.06/E
Status:	Final
Distribution:	Public