A Model-Based Development Process for Embedded System

Maritta Heisel[*] and Denis Hatebur[†]

Abstract: We present a development process for embedded systems which emerged from industrial practice. This process covers hardware and software components for systems engineering, but the main focus is on embedded software components and the modeling of problems, specifications, tests and architectures. Each step of the process has validation conditions associated with it that help to detect errors as early as possible.

1 Introduction

According to Broy and Pree [BP03], about 98% of the CPUs produced worldwide are used in embedded systems. Embedded systems can be found in almost every area of daily life. Moreover, they are often safety- or security-critical. Because embedded systems are usually produced in large numbers, incorrectly functioning systems might cause large damages. Hence, it is crucial to develop embedded systems in such a way that the probability of errors is minimized.

In this paper, we present a development process for embedded systems. That process was developed over time and gradually improved in an industrial context. It is based on development processes used for developing security-critical systems according to the Common Criteria [CC99] and the procedure required for developing safety-critical systems according IEC 61508 [Int98]. The process emerged from projects dealing for example with smartcard operating systems and applets for smartcards in the area of security-critical systems and motor control and automatic doors in the area of safety-critical systems.

The process consists of a sequence of steps to be performed. In each step, a natural-language description or a model (mostly expressed using UML2.0, [OMG03]) is developed. In addition, each step has some *validation conditions* associated with it that help to detect errors as early as possible in the process.

The development process as it is presented in this paper was successfully applied in the development of a protocol converter that connects a proprietary RS-485-based bus system with a CAN-bus system. For this system, there are hard real-time requirements, and the controller has limited memory and performance.

In Section 2, we explain the development process in some detail. Then, we discuss possibilities for tool support in Section 3. The paper closes with a discussion of the development process (Section 4).

2 Agenda for model-based development

We now present our model-based development process for embedded systems. As a means of presentation, we use the *agenda* concept [Hei98]. An agenda is a list of steps or phases to be performed when carrying out some task in the context of systems and software engineering. The result of the task will be a document expressed in some language. Agendas contain informal descriptions of the steps, which may depend on each other. Agendas are not only a means to guide systems and software development activities. They also support quality assurance, because the steps may have validation conditions associated with them. These

[*]Universität Duisburg-Essen, Fachbereich Ingenieurwissenschaften, Institut für Medientechnik und Software-Engineering, Germany, email: maritta.heisel@uni-duisburg-essen.de

[†]Universität Duisburg-Essen, Fachbereich Ingenieurwissenschaften, Institut für Medientechnik und Software-Engineering, Germany, email: denis.hatebur@uni-duisburg-essen.de and Institut für technische Systeme GmbH, email: d.hatebur@itesys.de

validation conditions state necessary semantic conditions that the developed artifact must fulfill in order to serve its purpose properly.

Table 1 shows an agenda that precisely describes how to carry out and validate the all the steps of the development process. In the following, each step is motivated and explained in more detail.

Table 1: Agenda for model-based development

No.	Description	Result	Validation
1.	Describe problem	system mission statement (SM), glossary with definitions and designations, requirements (R), domain knowledge (D), assumptions (A) in natural language and a context diagram (see [Jac01])	in Step 2
2.	Consolidate requirements	set of consolidated requirements (R), distinguished between ”need to have” and ”nice to have”	$D \wedge A \wedge R$ are consistent; $D \wedge A \wedge R' \Longrightarrow SM$; determine set R' ($R' \subseteq R$) of mission-critical requirements
3.	Decompose problem using D, A and R	set of problem diagrams with associated sets of requirements (R)	consistent with SM and context diagrams of Step 1; all requirements have to be captured
4.	**For all subproblems**: derive specification S using R, D and A	specification S of machine to construct (in natural language)	$D \wedge A \wedge S$ are consistent; $D \wedge A \wedge S \Longrightarrow R$
5.	**For all subproblems**: express system behavior, using specifications from Step 4	sequences of interactions between machine and environment (UML 2.0 sequence charts)	- all requirements must be captured - in the charts exactly the phenomena of the problem diagram are used - the direction of signals must be consistent with control of shared phenomena as specified in problem diagram - signals must connect domains as connected in problem diagram
6.	Design system architecture using results of Step 5	- system architecture (UML 2.0 composite structure diagram) - perhaps subcomponents (recursively) - all interfaces between the components (UML) - technical description of hardware interfaces	- all interfaces must be captured - all subproblems must be captured by at least one component

7.	**For all components**: derive interface behavior using results from Steps 5 and 6	interface behavior of all complex components, expressed as UML 2.0 sequence charts (test specification)	consistent with input
8.	**For all software components**: design software architecture using results from Step 6, phenomena of problem diagrams from Step 3 and reusable components from other projects	layered software architecture (UML 2.0 composite structure diagram), interfaces between software components (UML)	phenomena of problem diagrams are interfaces of the application layer; there must be one hardware abstraction layer for each external interface
9.	**For all software components**: develop specification, using results from Steps 7 and 8	component description consisting of: - component overview description (UML 2.0 class diagram with ports and lollipops) - data types (UML-Class-diagrams) - for all operations: pre- and postconditions (OCL or formulas) - invariants (OCL or formulas) - state machine (UML 2.0 state machine diagram)	consistent with interface behavior, completeness of state machines (implies error-cases for user interaction)
10.	**For all software components**: implement software components and test environment for software, using results from Step 7 for tests and Steps 8 and 9 for machine	test environment and software	run tests
11.	Integrate hardware and software using results from Step 10	system and test environment, including test interfaces	run test with hardware and software

Step 1 of the agenda is a creative process. In contrast to other work, we distinguish between requirements and a mission statement. This helps us to classify the requirements in ”need to have” and ”nice to have”. The system mission statement describes the purpose of the system in general terms. The requirements, in contrast, describe in more detail how the environment will behave after the developed system is integrated in it. The requirements are supposed to be a refinement of the system mission. Domain knowledge consists of facts that are true no matter how the embedded system is built. Assumptions are usually rules how users should behave, but which cannot be enforced[1]. The informal way of description used here is helpful to communicate with customers.

In **Step 2**, the consistency between the system mission and the requirements is checked. In particular, the domain knowledge, the assumptions, and the requirements should not be contradictory, and they should

[1]For more details, see [ZJ97, HS99].

suffice to accomplish the system mission. In most cases, domain knowledge, assumptions and further requirements have to be added to successfully perform the check. If there are requirements that are not needed to show that the system mission is accomplished, then either these requirements are not mission-critical, or the system mission is incomplete. Requirements not being mission-critical can be analyzed to decide if the added value for the customer is higher than the estimated cost to develop feature in question.

In **Step 3**, the problem is divided into subproblems, as described by Jackson [Jac01]. Each requirement must belong to the requirements of some subproblem. The subproblems are represented as *problem diagrams* (see [Jac01]).

In **Step 4**, specifications of all the subsystems to be developed (called *machines* by Jackson) are derived. Specifications are implementable requirements. Requirements that are not implementable are transformed into specifications using domain knowledge and assumptions. For an example, see [JZ95]. The specification is a description of the machine that contains all necessary information for its construction. It must be shown that, when the machine fulfills S, then the requirements are satisfied. For that proof, domain knowledge and assumptions can be used.

Step 5 uses the problem diagrams from Step 3 and the specifications from Step 4. For each subproblem, the desired behavior of the corresponding machine is specified using sequence charts. For the machine and for each domain in the problem diagram, one lifeline is included in the sequence-chart. The asynchronous signals between the lifelines are annotated with elements of the specified phenomena. This step is equipped with various validation rules that can be used to check the consistency between the problem diagrams an the sequence charts.
Experience from many projects has shown that sequence diagrams can easily be discussed with managers and customers that do not have technical knowledge. Loops, states, references and coregions do not cause any problems, while the other new constructs of UML 2.0 such as parallelism, continuation and considered signals should be used with care. The specifications developed in this step can be used as a basis for manual tests.

In **Step 6**, the system architecture is designed. The architecture of the embedded system is expressed as a composite structure diagram. This diagram uses objects for the components, whose ports are connected as described in [OMG03]. The connections are used to transmit the signals of the annotated interfaces between the components. The interfaces with their signals are specified using interface classes. The architecture can be specified recursively, i.e., components can have their own architecture, consisting of sub-components. The external interfaces of the components have to cover the interfaces of all problem diagrams. The architecture must cover all specifications developed in Step 5. This architecture is the starting point for the further development (hardware- as well as software development).

Step 7 refines the sequence diagrams from Step 5 for all complex components of the system architecture. Here, the signals specified in the interfaces of the architecture are used to annotate the sequence charts. These sequence diagrams are a concrete basis for the test implementation for all software components.

In **Step 8**, the software architecture for all components containing software is designed. The architecture of embedded software should be a layered architecture. The lowest layer is the *hardware abstraction layer*. This layer covers all interfaces to the external components in the system architecture and provides access to these components independently of the used controller or processor. For porting the software to another hardware platform, only this part of the software needs to be replaced.
The hardware abstraction layer is used by the *interface abstraction layer*. This layer provides an interface that includes the monitored and controlled variables (see [HJL96]) of the system. These variables can be derived from the context diagram or the problem diagrams. It is possible that these variables have to be calculated from the values of several hardware interfaces. For safety-critical software components, the interface abstraction layer will usually make use of redundant arrangements of sensors and actuators.
The highest layer of the architecture is the *application layer*. This layer only has to deal with variables from the problem diagram. Therefore, the system requirements can be directly mapped to the software requirements of the application layer, as described by Parnas [DLP95].
The software architecture is expressed as a composite structure diagram. To perform this step, the components specified in Step 9 of other projects can be reused.

In **Step 9**, the software components are specified as classes, taking a white-box view. These specifications have to be consistent to Step 7 with respect to the behavior of data types and state machines. The state machines must be complete, i.e., there must be a specified reaction to each possible input event. The specifications must have the same interfaces as in the component diagram designed in Step 8. In this step, we also have to decide if the component is an active (e.g., behaves like hardware) or passive (e.g., calculation-routine) component. The result of this step forms the basis for the implementation phase.

In **Step 10**, the test environment for all software components is implemented, using the test specification from Step 7. In addition, time frames must be be added, specifying when an event is expected to occur.
The system components are implemented using the results of Step 9, using some simple heuristics. The components have to be connected as specified in Step 8. For embedded systems, usually a static connection between components is established. This agenda allows to develop statically linked software components with the capability of reuse. To validate the results of this step, tests may be run in an emulation environment.

In **Step 11**, hardware and software components are integrated. The test of the whole embedded system, consiststing of hardware as well as software, is performed.

In a full paper, we will illustrate the application of our development process by means of concrete examples from the protocol converter.

3 Tool Support

We plan do equip the process described in Section 2 with tool support. To this end, models developed with specification tools must be exported to be used by validation tools. In particular, we started to extend the free specification tool ArgoUML with the new UML 2.0 composite structure diagram. The standardized XMI file format will then be used to check the consistency between several models created during the development process:

- Steps 6 and 7: It will be checked if the events in the sequence diagram are exactly those specified in the interfaces of the architecture.
- Steps 8 and 9: The interfaces of the architecture and those in the overview specification of each component must be the same.
- Step 9: Only those events specified in interfaces and the operations of the data types are allowed to be used in the state machine diagram.

We also intend to further enhance our process by using formal methods. Then, it should be possible to export the models to formal verification tools such as Atelier B, FDR, SPIN or SVM. For hardware-software-codesign, export from and to VHDL is planned.

4 Discussion

We now recall the most important characteristics of our development process for embedded systems.

The process proposed here is **model-based**. Modeling is used for problems, specifications, architecture and component behavior. Consistency checks between the several views of the machine are possible (independently from the used tool), because UML provides a standardized XML-based file format that can be parsed easily.

The process covers not only software but the whole system, consisting of **software and hardware**. Within the process, the hardware-software-partitioning problem is addressed. System and software are specified using the same notation. Therefore, the specification can be refined on the system level (Step 6) if more behavioral information is required before the hardware-software-partitioning is possible.

The process is **tailored to embedded systems**. The application domains of many embedded systems can be covered by the four-variable-model proposed by Parnas [DLP95]. Apart from the hardware abstraction layer, the four-variable-model is the most important design criterion for the layered architecture proposed in our development process.

The proposed process supports the **reuse of components** already in the specification phase (see Step 8). Reuse can further be supported by using design patterns.

In large parts, the process makes use of **UML 2.0**. UML 2.0 combines the advantages of the widely known UML and the Specification and Definition Language (SDL) that is used for telecommunication protocols. In contrast to UML 1.4, our layered architecture can be expressed adequately with UML 2.0. In contrast to SDL, UML 2.0 allows a much more flexible structure of components that allows better reuse of components.

The development of **test cases** is an elementary part in our process. The development of test cases is structured, problem-based and requirement-based. The test specifications are expressed as sequence charts, and test cases can be derived (or generated) from these charts just by replacing points of time with time frames expressing when desired events are expected.

For each step of the development process, we have defined **validation conditions**. These conditions can be checked using reviews and inspections. However, for many of the validation conditions, formal proof or demonstration is also possible.

The process is defined in such a way that **tool support** can be added in a modular way, based on existing tools.

Finally, our process has been developed in an industrial context, and it was **successfully applied in practice** in several projects for developing security- and safety-critical systems.

References

[BP03] Manfred Broy and Wolfgang Pree. Ein Wegweiser für Forschung und Lehre im Software-Engineering eingebetteter Systeme. *Informatik Spektrum*, 18:3–7, Februar 2003.

[CC99] Common Criteria for Information Technology Security Evaluation, 1999. aligns to ISO/IEC 14508:1999, see http://www.commoncriteria.org.

[DLP95] J. Madey D. L. Parnas. Functional documents for computer systems. In *Science of Computer programming*, volume 25, pages 41–61, 1995.

[Hei98] M. Heisel. Agendas – A Concept to Guide Software Development Activites. In R. N. Horspool, editor, *Proc. Systems Implementation 2000*, pages 19–32. Chapman & Hall London, 1998.

[HJL96] C. Heitmeyer, R. Jeffords, and B. Lebaw. Automated Consistency Checking of Requirements Specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.

[HS99] M. Heisel and J. Souquières. A Method for Requirements Elicitation and Formal Specification. In J. Akoka, M. Bouzeghoub, I. Comyn-Wattiau, and E. Métais, editors, *Proceedings 18th International Conference on Conceptual Modeling, ER'99*, LNCS 1728, pages 309–324. Springer-Verlag, 1999.

[Int98] International Electrotechnical Commission. Functional safety of electrical/electronic/programmable electronic safty-relevan systems - Part 1: General Requrements, 1998.

[Jac01] M. Jackson. *Problem Frames. Analyzing and structuring software development problems*. Addison-Wesley, 2001.

[JZ95] M. Jackson and P. Zave. Deriving Specifications from Requirements: an Example. In *Proceedings 17th Int. Conf. on Software Engineering, Seattle, USA*, pages 15–24. ACM Press, 1995.

[OMG03] Object Management Group OMG. UML 2.0 Infrastructure Specification, 2003.

[ZJ97] P. Zave and M. Jackson. Four dark corners for requirements engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30, January 1997. Also availble under http://www.research.att.com/˜pamela/ori.html#fre.