The IEEE International Conference on
Industrial Informatics (INDIN 2008)
DCC, Daejeon, Korea July 13-16, 2008

A Code Generation Tool for Embedded Automotive Systems Based on Finite State Machines

Felix Lindlar
Daimler Center for Automotive
Information Technology Innovations
Technische Universität Berlin, Germany
Email: felix.lindlar@dcaiti.com

Armin Zimmermann
Real-Time Systems and Robotics
Technische Universität Berlin, Germany
Email: zimmermann@cs.tu-berlin.de

***Abstract*—This paper introduces a tool for the automatic code generation of automotive embedded systems. Electronic control units (ECU) are supplied with multiple software tasks, which are organized by an OSEK/VDX operating system. The code is generated based on an annotated finite state machine. Further features of the tool include a structural validation of the state machine and a schedulability check to guarantee that all deadlines of the system are met.**

***Index Terms*—Source Code Generation, Finite State Machines, Deadline Monotonic Analysis, XSLT, OSEK, Model Based Code Generation**

I. Introduction

Modern vehicles rely to an increasing degree on safety-critical systems containing software. A few years ago the malfunction of a software system in a car may have led to the failure of a simple function only. Nowadays this could impose a safety thread to passengers. Embedded control in automotive applications is usually implemented as a distributed set of microcontrollers, called electronic control units (ECUs), which are connected by specific bus systems such as controller area network (CAN) or FlexRay.

Functions of these controllers depend on each other and are often implemented by different teams or even companies. Integration of software and hardware as well as testing different configurations has become a major issue in automotive system engineering. Conventional manual implemention of software often results in a higher error rate, because not all interdependencies are understood. Model-based techniques in software engineering can help to overcome this situation [5], [14].

Based on a formal model describing the different states (or modes) of the system and its state changes, it is possible to check certain properties and to finally generate control code. Automated code generation avoids human errors in this step and thus decreases the risk of a system malfunction. The fact that many similar projects are implemented can be supported by using templates for typical functions and to combine them in a parameterized way.

The main reason to implement a model-based code generator is to shift the focus on design rather than implementation — it is desirable to derive the implementation directly from the specification. Productivity, quality and coping with complexity are expected to improve thereby.

In the industrial application at Continental AG reported in this paper, software configurations for electronic projects in the field of hybrid-electric power train had to be generated automatically. The framework is described by a state machine and comes into operation in an environment managed by an OSEK operating system. OSEK is a static operating system for embedded real-time systems often applied to automotive problems. It allows the effective utilization of resources for automotive software and other embedded systems.

Because of its field of application, OSEK has to provide support for real-time software. Through modularity and a high degree of customizability it can be adapted to different application areas. Both low-end micro processors and complex systems are supported. Since the configuration of an OSEK system is carried out statically, system components cannot be created dynamically at runtime. This leads to an increase of reliability and reactivity.

In the software configurations to be generated there are sets of functions mapped to every state of the underlying state machine. Those process functions implement the actual controlling functionalities but are not part of the generated code and thus not the focus of this work. The code to be generated implements the state-dependent invocation (scheduling) of these functions. Moreover, there are initialization and de-initialization functions which have to be called when the system state changes. The purpose of these functions is to prepare the system for a new state.

The classic approach would be to implement the application code by hand. Manual translation of the state machine into source code and mapping of the functions would be time-consuming and error-prone. To improve and automate the process, model-based code generation and verification of properties at the model level are necessary. Unfortunately, current modeling tools were not satisfactorily able to handle the specific needs of the engineers. Thus a specific software tool – the *State Machine Code Generator* (SMCG) – has

978-1-4244-2171-8/08/$25.00 ©2008 IEEE

been implemented, which is described throughout the paper. The SMCG generates the required source code without the necessity to implement a single line of configuration code by hand.

The underlying state machine is verified automatically and dependencies between functions are checked. It is thus guaranteed that the sequence of initialization, execution and de-initialization is followed for every individual function.

Another issue of automotive embedded systems is timing and schedulability. Due to its field of application in the automotive context, all deadlines of the multi-tasking real-time system have to be guaranteed. An additional feature of the SMCG tool is thus to perform a schedulability test – preventing the system from crashing once the roll-out to the target platform has been performed. For that matter, Deadline Monotonic Analysis (DMA) is used, resulting in the worst-case response time of tasks. Our tool thus allows to check, if the planned priority-based multi-tasking system can be scheduled.

The SMCG communicates with the user through a graphical user interface and produces the desired source code. Some documentation reports are generated as well. The proposed tool has been developed for the automotive context, where low memory consumption and strict real-time capabilities are especially important. The complexity of the automatically generated code is kept to a minimum. Furthermore the tool performs a schedulability test and thus makes it easy for the developer to optimize the system's runtime behavior. It has been shown in several projects that the development time – especially for testing – is comparatively low.

There is a body of tools generating code from state machines. Examples include Rational Rose Technical Developer by IBM, a model-driven development tool based on UML 1.4. It does not, however, directly support OSEK/VDX as a target platform. Other examples are UML 2.1-based Rhapsody by Telelogic and TargetLink by dSpace, both with OSEK/VDX support.

Behavioral models of software are useful to design complex real-time systems. Typically, states and state transitions are elements of these discrete model types. Well-known examples include Statecharts [7], [8], Modecharts [11], Augmented State Transition Diagrams [9], and behavioral diagram types of the Unified Modeling Language [13]

The remainder of this paper is organized as follows. Software configurations considered in this paper as well as the chosen state machine model type are described in the subsequent section. Section III shows how some important properties of the modeled system can be checked at the model level, before the actual code is being generated from it automatically. How this is done is covered in Section IV. Section V describes architecture and use of the implemented software tool and briefly reports on an application examples. Finally, conclusions are given.

II. Software Model

A classical state machine consists of a finite set of states and a set of transitions between them [8]. Conditions can be assigned to transitions. Thus a change from one state to another is only carried out if the associated condition is true. Furthermore, specified actions can be executed as the result of a state change. One of the states has to be the initial state, assigning one or more final states is optional. Originating from the initial state a state machine is always in exactly one state. A state machine can be represented graphically by a state diagram or in table form by a state matrix.

State machines referred to in this paper have been extended by features not contained in the classical definition. For every state of the state machine there exists a set of functions for every task of the system. Depending on the current system state and the currently active task all the linked functions are being executed one after another. A function set of this type is called *cyclic function container* in this document. It refers to the set of functions that need to be executed (scheduled) with a predefined frequency in each state.

Additionally there are sets of functions linked to transitions. These sets are called *transition function container*. Every time the system state is changed the functions in the *transition function container* are executed respectively. A state change happens when the boolean condition evaluates to *true* in a state.

Fig. 1 shows a part of an example. There are three boxes, one for each task of the system, assigned to state *Run*. Every time the OSEK operating system triggers either the $1ms$, $10ms$ or $100ms$ alarm, the corresponding functions in the cyclic container are executed. Functions corresponding to the illustrated state transitions are called when the system changes from *Run* to *Standby* or vice versa.

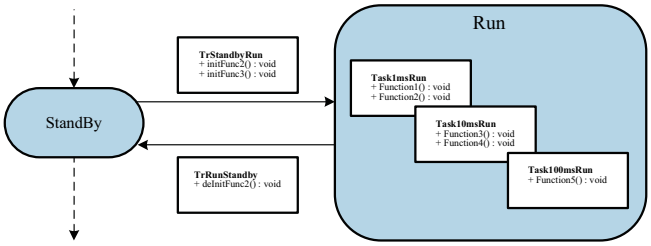

Fig. 1. Example of an annotated state machine

III. Verification

This section introduces the techniques used for verification by the SMCG tool prior to generating source code.

A. Checking for a Valid State Machine Structure

To ensure the system has the desired behavior, state machines are validated to certain criteria before the actual code generation is allowed to take place. The SMCG only accepts connected state machines, since there would be states that could never be reached and would result in dead code.

In addition there can only be one initial state – a state with outgoing links only. Every state has to be reachable from the initial state. One final state has to be defined. Loops are allowed but there has to be at least one path from every state that leads to the final state.

B. Checking of Dependencies between Functions

A *cyclic function container* is assigned to each state and task. The functions in a specific container are always executed when the system is in the according state and the OSEK operating system triggers the corresponding alarm. As a general rule, these functions need to be initialized before their first execution by an initialization function. Additionally some of the functions need to release resources following their last execution by a de-initialization function.

Transition function containers are assigned to transitions. A state change is carried out when the condition of a transition leading away from the currently active state evaluates to *true*. The functions in the target state are executed afterwards. A special case of a *transition function container* is the *reset container*. The *reset container* is always processed once when the system starts. There are two types of functions in *transition function container*: Initialization and de-initialization functions.

Every initialization functions is associated to one or more functions referenced in *cyclic function containers*. The tool has to assure that the individual initialization function is executed before one of the associated functions is called. Since there may be several paths to a state in which a function is called, the initialization function has to be called at least once on every path, otherwise a stable performance of the system cannot be guaranteed.

Starting at the initial state the SMCG uses a *breadth first search* for every function addressed in a *cyclic function container* to verify that no uninitialized function is called in the generated source code. Each search path terminates if the appropriate initialization function is found in a *transition function container* or if the final state is reached. An error is detected if the function under observation is reached, since no initialization function would have been called for that function.

Fig. 2 shows an example of a faulty configuration. No initialization function for *Function2* in *State 2* is called. *Function2* relies on *InitFunc2*. Moreover, *Function3* in *State 4* is not initialized on every path leading to the state. The system is thus potentially unstable and no code should be generated before the errors are fixed.

In addition for some of the functions it has to be verified that a deinitialization function is called after the last execution. Some modules share resources which need to be released in correct order for other modules to use them. All transitions leaving a state where a function is addressed in a cyclic function container that implies the call of a deinitialization function have to be checked to ensure a stable system.

C. Schedulability Check

The SMCG tool also provides functionalities to warn the user about possible deadline exceedings. To achieve that, it performs a schedulability check prior to code generation. For a higher processor usage than with static scheduling, tasks are planned dynamically based on priorities. The precondition is, that the tasks are independent and a priority is assigned to each of them. The responsibility of the preemptive operating system (here: OSEK) is to assign the processor to the task with the highest priority of all the tasks that are ready to run. As soon as a task with a higher priority than the task that is currently running is ready, the operating system has to give the processor to that task.

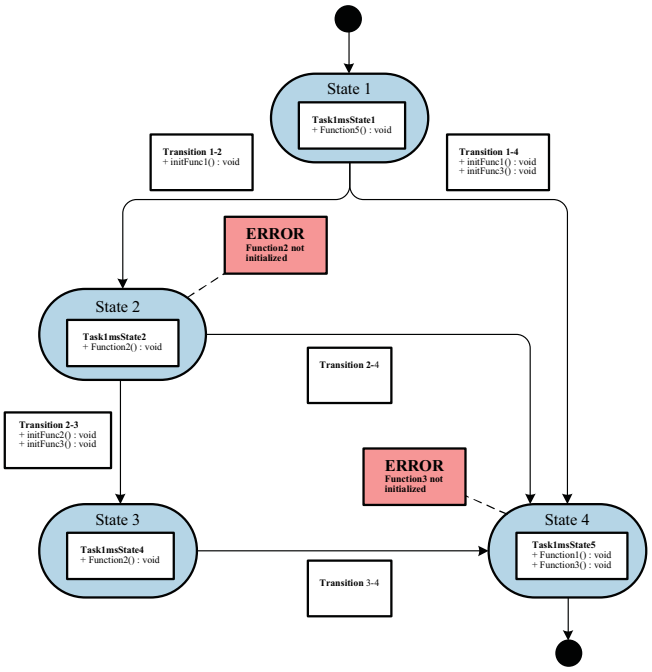

Fig. 2. Example of an erroneous model

This type of system has a faster response time than its static correspondent. Higher-prioritized tasks and interrupts do not have to wait for tasks with lower priority to terminate. Furthermore, task periods do not have to be selected based on harmonic values to keep the cycle time short.

To implement this type of scheduling the worst-case response time (WCRT) has to be calculated for all tasks. The WCRT is defined as the longest response time from the moment a task is ready to run until its completion. The worst-case execution time (WCET) is the maximum execution time a task needs excluding interrupts from higher prioritized tasks. Hence the WCRT is the sum of the WCET and the interference of higher prioritized tasks. For a real-time system with hard deadlines it has to be guaranteed, that the WCRT for all tasks is shorter than their deadline.

Deadline Monotonic Analysis (DMA) [12] is a method to calculate the worst-case response time (WCRT). A priority-based scheduling can be implemented based on the results of DMA [2], [3]. In theory, a processor load of 100% can be achieved based on the results.

The WCRT R_k is the sum of the worst-case execution time

and the interference of tasks with higher priorities:

$$R_k = C_k + I_k. \tag{1}$$

C_k is the Worst-Case Execution time of task k and I_k is the interference of tasks with a higher priority than task k. The interference in (1) is calculated as follows:

$$I_k = \sum_{\forall i \in hp_k} \left\lceil \frac{R_k}{T_i} \right\rceil * C_i. \tag{2}$$

T_i is the period of task i and hp_k is the set of all tasks with a higher priority than task k.

Since there are several states in our software model, while the system can only be in one state at a time, the DMA has to be performed for all states individually. The WCET of a function set is the sum of the WCET of every single function in the set. From the result of the DMA, the SMCG tool determines if all deadlines will be maintained in the final system.

This does of course require knowledge about the worst-case execution times of all called functions. To approximate these execution times, the SMCG tool uses the *Trace32* tool. Its task is to trace the execution of a software system and produce the desired run time data. An experimental setup is necessary perform the measurements, but data can be reused for future projects, because many functions remain the same.

Substituting (2) into (1) and transforming it results in the recursive formula

$$R_k^{n+1} = C_k + \sum_{\forall i \in hp_k} \left\lceil \frac{R_k^n}{T_i} \right\rceil C_i. \tag{3}$$

The formula converges for an initial value of zero for R_k. If the WCRT for every task is smaller than its corresponding deadline, the system is guaranteed to not exceed any deadlines.

D. Processor Load

To gain another significant metric, the processor load is calculated for each state. The processor load is defined as the ratio of utilization time and total runtime of the processor. The maximum processor Load $PLoad_j$ in a state j is:

$$PLoad_j = \sum_{\forall k \in K} C_k * 1/T_k. \tag{4}$$

K is the set of all tasks.

Changing the task period affects the processor load. A shorter period of a task leads to a higher execution frequency and thus to a higher load. For example a system with a single task with a period of $1ms$ and a WCET of $10us$ has a processor load of 10%. Changing the period to $0.5ms$ increases the load to 20%.

The following has to be true for the processor load in all states:

$$\forall j \in J : (\forall k \in K : R_{k,j} < D_k). \tag{5}$$

The processor load gives an indication of how critical the runtime in a single state is. Integrating runtime analysis and code generation in a single application makes the development process more efficient. Changes can be applied more easily and the immediate effect can be observed.

IV. Code Generation from System Model

On the basis of a valid state machine that passed all checks – functions initialized and deinitialized if required and no deadline exceedings - the source code can be generated. The SMCG uses XSLT (Extensible Stylesheet Language Transformation) for that purpose. The required data set consists of states, transitions and transition conditions gathered from the XML state machine representation. This set is further supplemented by function mappings and task periods.

The SMCG needs to preprocess the project data for later use by the XSLT processor. XSLT is a Turing complete programming language which main purpose is to transform a XML document into other documents. The hierarchically structured input document is not altered in the process. Rather a new document is constructed. This document is often another XML document but XSLT can also be used to create documents of various other types.

The most important characteristics of XSLT are:

- One or more XML input files are required,
- Parts of the input document are addressed using XML path language, and
- XSLT-Stylesheets define the conversion from input to output documents.

Fig. 3 shows an example of the code generation process. The two input files for the XSLT processor are shown on top. The XML file contains the data and the XSLT files the rules to be applied to the data. After processing of the input files, the processor generates the code.

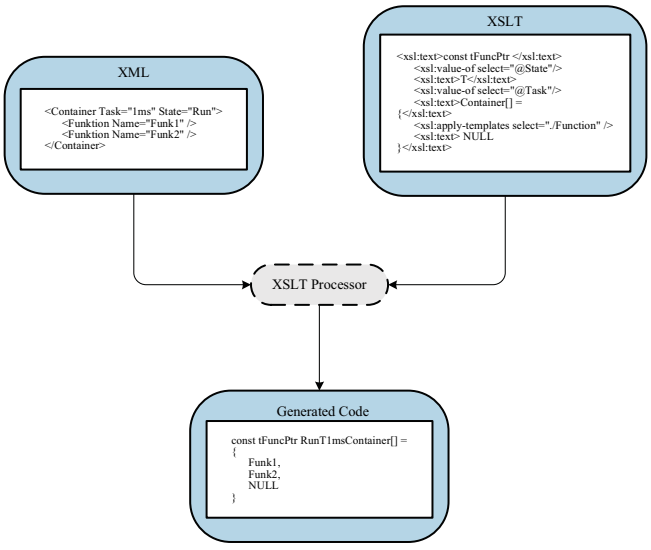

Fig. 3. Code generation with XSLT

The principles of the transformation can be summarized as follows:

- The input XML document is parsed and interpreted as a tree,
- The XSLT document is parsed and interpreted as a tree,
- Both trees are read by the XSLT processor, and
- Both trees are transformed according to the instructions in the XSLT document into the output document/file.

XSLT provides the necessary functionalities to handle complex requirements regarding code generation. It allows for example to link statements to conditions and to use loops to process any number of nodes. A further advantage is the option to reuse XSLT code even from different programming languages. Modifying the XSLT files changes the output source code without the necessity to recompile the program, since XSLT files are read at runtime. The use of schemas helps to ensure that the input XML format always conforms to the requirements. Thus it can be guaranteed that the generated output meets the specification.

V. Software Tool and Case Studies

The implemented SMCG software tool is described in the following.

A graphical User Interface (GUI) enables easy operating of the SMCG. The main features are:

- Management of projects,
- Graphical mapping of functions to states and transitions,
- Display of runtime analysis results,
- User access authorization to projects and
- Support in acquiring data for projects.

Fig. 4 shows a screen shot of one of the tool's dialogs where the user can map functions to transitions. The set of available functions is extracted from source code by an internal parser, while the state machine model is imported from an external modeling tool.

Data storage and manipulation is done with an SQL database. To reduce the size of the database and the time spent to update the database, two types of data are distinguished. First, there is project-specific data which can only be used by a single project. This data set includes among others:

- Project details,
- Project configuration,
- Change log, and
- Function Mappings.

Second, there is data which can be reused by several projects. Thus only links to the reusable data have to be stored for a certain project and not the whole data set. The main components of the set are:

- Function details,
- Function runtime data, and
- Function interdependencies.

The tool uses XML and text parsers to minimize the effort the user has to spend to accumulate data required by the SMCG. XML parser are used to read the state machine representation generated by a graphical UML editor. Information

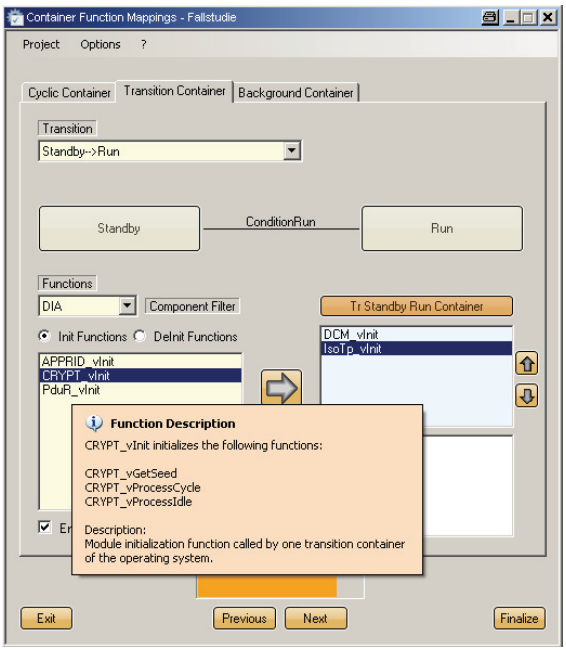

Fig. 4. GUI dialog of the SMCG

about functions are extracted from source code using a text parser.

To further optimize the process, reports for projects are generated automatically. The SMCG uses the project data to create well arranged text documents which makes it easy for reviewers to understand and continue to work on projects.

Fig. 5 shows an overview of the tool's data flow. The XML representation of the state machine has to be created with an external tool (e.g. Enterprise Architect) and is then parsed by the SMCG. The text representation of the state machine, function details, and runtime values serve as further input for the tool. Once the user has completed configuring a project using the graphical user interface – the code generation process can be activated. Output files (source code, documentation and project log) are only generated if no errors are detected by the application and the results of the schedulability check are satisfying. In the case that there are faults, the SMCG assists with correcting them.

A. Tool Use and Case Studies

After a new project is set up based on a finite state machine, function mappings have to be added to states and transitions. The SMCG tool provides information about the effect on schedulability. In addition to that, interdependencies between functions are monitored and warnings and errors generated to inform the user about inconsistencies. Functions that need to be mapped to states or transitions that have not been added to the database can be supplemented using a parser which extracts the required information directly from source code files. The runtime values appertaining to the function also have to be added either by hand or by another integrated text parser. If no deadline violations are detected and functions are

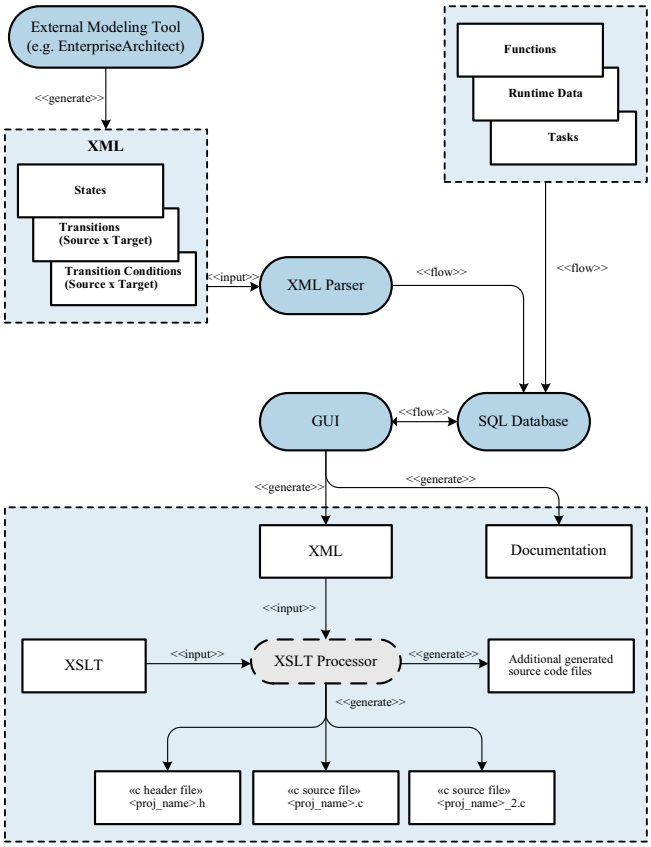

Fig. 5. Data flow of the SMCG

initialized and deinitialized as required, output files are finally produced.

In particular the tool was used for generating code for embedded devices in the field of hybrid-electric power train projects. Several application examples were considered twice, i.e., manually and using the tool, to evaluate the advantages. The total amount of code generated for each case study covered several thousand lines of code. The case studies showed that the development process was sped up significantly by using the SMCG. Many errors the C compiler was unable to detect led to a lot of time spent on debugging for projects implemented the conventional way. Those errors were most frequently violations of function interdependencies but also included, i.e., omitting of terminal symbols in a function pointer array and wrong or missing includes. However, none of those errors occurred in the automatically generated code. Moreover, no effort was required to keep documentation and code consistent. Finally, the integrated schedulability check proved very useful with respect to efficient hardware use and meeting of deadlines.

VI. Conclusion

Customized software for particular problems are important in automotive embedded system design. Memory and runtime optimization and a high degree of reliability are key factors. Model-based techniques help to avoid errors in the software engineering process. The paper presented an industry tool, which checks properties such as correct execution order and schedulability on a state machine variant. When modifying a project, the developer can observe the effect on the system's performance immediately. The actual code for execution on automotive ECUs running OSEK is generated automatically.

The SMCG tool has been tested in several industry projects. A significant decrease in development time, number of errors and required time for testing was observed. Since part of the documentation is generated automatically, the developer saves further valuable time, while consistency between documentation and source code is guaranteed.

Acknowledgment

Part of the reported work has been performed during a research internship at Continental AG in Berlin. The authors would like to thank Jörg Scheiner and Ralph Kottman from Continental AG for their support.

References

[1] M. Arbib, *Theories of Abstract Automata*. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1969.
[2] N.C. Audsley, *Deadline Monotonic Scheduling*. Dept. of Comp. Sci., Univ. of York, York, Y01 5DD, England, 1990.
[3] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings, *Hard Real-Time Scheduling: The Deadline-Monotonic Approach*. Dept. of Comp. Sci., Univ. of York, York, YO1 5DD, England, 1991.
[4] A. Burns, *Scheduling Hard Real-Time Systems: A Review*. Software Eng. Journal, 6, pp. 116-128, 1991.
[5] J. Cooling, *Software Engineering for Real-Time Systems*. Addison Wesley, 2003.
[6] A. Gill, *Introduction to the Theory of Finite-state Machines*. McGraw-Hill, 1962.
[7] D. Harel, Statecharts: A visual formalism for complex systems. Science of Computer Programming, 8:231-274, 1987.
[8] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, *STATEMATE: A working environment for the development of complex reactive systems*, IEEE Transactions on Software Engineering, 16(4):403-414, 1990.
[9] C. S. Hendricksen, Augmented state-transition diagrams for reactive software. ACM SIGSOFT Software Engineering Notes, 14(6):61-67, 1989.
[10] ISO 17356-3:2005, *Road vehicles - Open interface for embedded automotive applications - Part 3: OSEK/VDX Operating System (OS)* 2007.
[11] F. Jahanian, A. K. Mok, Modechart: A specification language for real-time systems. IEEE Transactions on Software Engineering, 20(12):933-947, 1994.
[12] J. Liu, *Real-Time Systems*. Prentice Hall, 2000.
[13] Object Management Group, Unified Modeling Language Specification v.2.0. www.omg.org, August 2005.
[14] W. Wolf, *Computers as Components*. Academic Press, 2001.