Models for automatic generation of safety-critical real-time systems

Christian Buckl and Matthias Regensburger and Alois Knoll and Gerhard Schrott
Robotics and Embedded Systems
Department of Informatics
Technische Universität München
Email: {buckl,regensbu,knoll,schrott}@in.tum.de

Abstract

Model-based development has become state of the art in software engineering. A number of tools, like Matlab/Simulink or SCADE, are available for the automatic generation of application code on basis of models. Unfortunately, system aspects like process management, communication or fault-tolerance mechanisms are not covered by these tools. One main reason is the non-existence of appropriate models with an explicit semantic to allow the automatic code generation. In addition, there is a great need to have the possibility to extend both the model and the code generation abilities to allow a high coverage of the used platforms, since such code is platform dependent. In this paper, we will present an approach applying meta code generators using template-based code generation to achieve this extensibility and will discuss the properties of models required for the use in model-based development of system aspects for safety-critical real-time systems.

1. Introduction

Model-based design [18] has become state of the art in software engineering. Especially the existence of diverse tools for automatic code generation like Matlab/Simulink [1] or SCADE is very attracting. Particularly for the domain of safety-critical applications, where the developers are typically application domain experts with less background in programming fault-tolerant real-time systems [16], the possibility to provide extensive code generation would be crucial.
Unfortunately, the code generation abilities of existing tools typically cover only the functional aspects of the applications like the control functions. System aspects like process management, scheduling, inter-process communication, communication within the distributed system and fault-tolerance mechanisms are not addressed in general. One reason is the absence of adequate models with an explicit semantic. The widely used Unified Modeling Language UML [9] for example lacks the precision and rigor needed for code generation [12]. Only few models, such as class or state machine diagrams, can be therefore used for automatic code generation. A solution to this problem is the usage of domain specific languages (DSL) [6].
Another big problem is the platform dependency of system level code. Since safety-critical real-time software is typically embedded in a larger system, there exists a huge heterogeneity of the used platforms, the combination of the hardware, operating system and programming language [21]. Due to this variety, it is not possible to design a code generator for system aspects that supports a priori all these platforms. Rather, the code generator must support an easy extension as well of the underlying model, as of the code generation ability. Template-based code generators can be used to achieve this extensibility regarding the code generation ability. In this paper, we will present an approach using meta code generators applying template-based code generation to achieve extensibility both on the model and the generation side. In addition, we will discuss the properties of models required for the use in model-based development of system aspects for safety-critical real-time systems.
The paper is composed as follows: section 2 discusses related work with a focus on models used for automatic code generation. In section 3, our approach is outlined and the code generator architecture is explained. Since our tool is intended as an add-on to other model-based development tools like Matlab/Simulink, we discuss the aspects (system level code, fault-tolerance mechanisms) that are covered by our approach in detail in section 4. Afterwards, the different models and their properties are discussed in section 5. For illustration purpose, we use a simple example: a control application using a PID controller, implemented on a Triple-Modular-Redundancy- (TMR-) System. The paper is concluded with a summary of our approach and an overview of the current status regarding the practical realization in section 6. Finally, we close with some remarks about future work.

2. Related work

A number of tools for model-based development of applications exist, but typically these tools focus only on some specific aspects of the whole system, mainly the functional aspects like the control functions. Examples for such tools are Matlab/Simulink by Mathworks [1] or SCADE.
Matlab/Simulink focuses on the development of algorithms, data visualization, analysis and numeric computation. By the use of the Real-Time Workshop, C code can be generated out of the models automatically. Features of distributed systems like networking or process management are not covered by the Real-Time Workshop.
SCADE by Esterel Technologies is a tool for code generation of safety-critical applications based on synchronous languages like Esterel[2] or Lustre [5]. The main idea of SCADE is to generate sequential code to avoid problems like race conditions. Therefore, features like networking and process management are also not covered.
In addition to tools used for specific application domains, there are also several more general tools, typically using the unified modeling language UML [9] as modeling language. While there are some diagram types that are suited for automatic code generation like class or state machine diagrams, most diagram types are only used for modeling the systems, while the implementation has to be done by hand. Reasons are the lack of precision and the ambiguous semantics of these diagrams [12].
A more promising approach is the use of domain specific languages [6] and models. One successful example of this approach is Giotto [10, 11] from the University of Berkeley. Giotto is a time-triggered language used for the specification of distributed real-time systems. The applications are interpreted on two virtual machines: the embedded and the scheduling machine. Therefore the usable platforms are restricted to platforms for which the virtual machines are implemented. In addition, Giotto is not designed to support the automatic realization of fault-tolerance mechanisms.
Regarding the realization of a fault-tolerant communication using model-based development, the time-triggered architecture TTA [13] can be used. TTA is a framework for the design and implementation of distributed fault-tolerant applications with a focus on the automotive and aviation industry. TTA provides different services like predictable communication with small latency, clock synchronization and membership service [14]. The approach is based on a hardware solution, the so-called TTP/C controller [24], running the TTP protocol that realizes time-triggered communication on redundant communication channels. Because TTA concentrates only on a fault-tolerant communication, the implementation of mechanisms for the toleration of other error sources has to be done by the developer itself. Another disadvantage of TTA is the restriction on specialized hardware. Therefore the application area is restricted.
The increase of code reuse is the main goal of component-based approaches [23] and architecture description languages [17]. However, most of these approaches do not provide any possibilities to adapt the behavior of the component to application needs, which is one requirement for safety-critical embedded software. Particularly in the context of fault-tolerance mechanisms, the strong correlation between application design and required component behavior becomes obvious. This problem is addressed within FRACTAL [3], which provides a component model with extensive reflection capabilities. The usability of this approach for embedded software was shown in the project Think [8]. In contrast to our approach, the developer has to have however in-depth knowledge of the available components using FRACTAL. Our approach has to be seen much more as a meta-programming technique: all implementation details are hidden from application programmers, the selection of appropriate components/templates and the adaptation is automated.

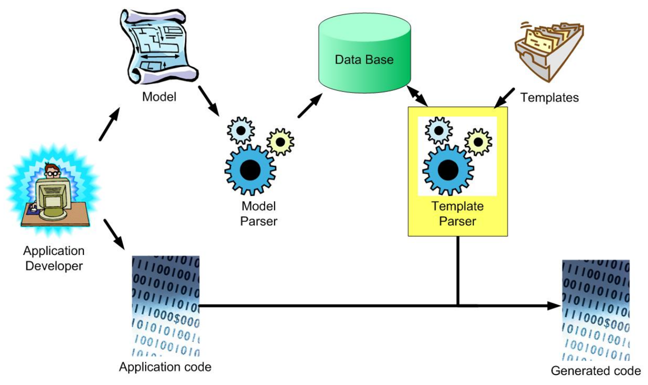

Figure 1. Code Generation Process

3. Code generator architecture

In the previous sections the need for a new code generator architecture for model-based development of safety-critical real-time systems was motivated. Since it is impossible to design a code generator that supports all platforms a priori, the code generator must be designed to support extensibility, even by the user. This extensibility must be realized on two different levels: regarding the meta-model used for code generation and regarding the code generation ability. A solution for this problem is the use of template-based code generators. In such code generators, the generation ability is outsourced to templates. Therefore, the code generator's task is limited to the selection of appropriate templates and the adaptation to the application needs. New templates can be easily added by the developer to sup-

port further platforms or new functionality.
Instead of implementing a new code generator, we use for our approach an existing tool, called openArchitectureWare [7]. OpenArchitectureWare is a modular MDA/MDD generator framework. It is implemented as plug-in for the Eclipse development environment that also provides a graphical interface to design the model. The code generation process is depicted in figure 1. The input of the code generator is the model describing the application and the application dependent code, e.g. the control function.
The code generation is performed by using templates. A single template can offer an application independent solution for one aspect of the system, e.g. scheduling, or may combine other templates. Thus a hierarchical order of the templates is possible. Templates are typically designed for a specific platform, because of the code dependency.
The adaptation of the templates to the model by the code generator is performed using a technique similar to preprocessor macros. A detailed description of this technique can be found in [4].
In case there are no appropriate templates for a specific application available, the application developer can design new templates or change existing ones. Thus the extensibility of the code generation ability to support diverse platforms is guaranteed.
The extensibility issue regarding the meta-model can be solved by the use of a meta-code generator. Figure 2 depicts the process of generating a code generator. Within our system, we are providing a basic meta-model and instructions on how to adjust this meta-model. Therefore the application developer can adjust the meta-model very easily to the application needs, for example by adding a new operating system or new network settings. On the basis of the meta-model, a code generator including a model parser, a graphical modeling environment and a template parser are generated. In addition, a template interface description is generated that can be used to design new templates or adjust existing ones.

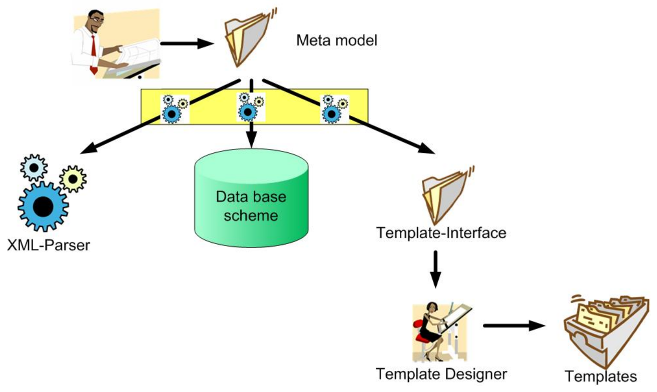

Figure 2. Meta Code Generator

4. Automatically generated features

As seen in section 2, there are a number of tools with code generators that cover the application aspects, but not the system aspects. In this section the features that are covered by our approach are explained in more detail. In addition the requirements on the model to generate automatically these features are discussed.
We assume the application to consist of different tasks executed on a distributed system. On each control unit some real-time operating system, e.g. VxWorks or RTLinux, is operated and the nodes are connected via a network. Each of the control units is equipped with an internal clock and some of the units may have sensors and actuators.
Our goal is to generate automatically the system aspects for this setting, more precisely the application execution including timing, process management and inter-process communication, networking and I/O aspects as well as fault-tolerance mechanisms. These different aspects are explained in more detail in the following subsections.

4.1. Application execution

One main task of the generated code is the coordination of the application execution. This includes the management of the start and end of the different processes as specified in the model, the temporal correct execution of I/O operations and the management of the temporal correct communication between sensors, actuators and the control processes. To allow an automatic generation of these mechanisms, the model must comprise a precise execution model, especially regarding the exact specification of timing issues and the exclusion of race conditions by design.

4.2. Process management

Related to process management, the basic process management (creation, deletion), inter-process communication as well as the scheduling are realized automatically in our approach. Race conditions regarding the inter-process communication can be avoided by clear communication semantics within the model. Violations or inconsistencies must be detectable at model level.
To realize an efficient scheduling algorithm, clear scheduling criterions (for example deadlines) must be available within the model. One prerequisite for an automatic code generation is the possibility to specify details about the used operating systems and programming languages within the model.

4.3. Networking

Besides the realization of the pure network communication, also some means to achieve temporal synchronization of the different nodes in the distributed system must be available. To reduce the network load, this temporal synchronization should be combined with the regular messages of the application. The network type and network settings must be also contained within the model.

4.4. I/O

For several standard devices an automatic code generation would also be very useful. Typical devices can be digital/analog boards, the serial interface or the network. The input and output semantics, e.g. blocking or non-blocking, must be clearly contained within the model to allow an automatic code generation.

4.5. Fault-tolerance mechanisms

Finally the code generator should also be able to generate standard fault-tolerance mechanisms at system level. This includes error detection, e.g. by voting or plausibility checks, hot or cold standby, exclusion of erroneous nodes and the integration of repaired nodes.
Since the fault-tolerance mechanisms are implemented at system level, it is not possible to perform the error-detection and localization at application level. For example, it is possible to detect an erroneous result by using a triple modular redundancy (TMR) architecture, but it may not be possible to detect the reason for this error. In addition, since the reaction to failures are very often application dependent, there must be the possibility for the user to define appropriate reactions. Within our system the standard reactions to errors are to restart the process, to reboot the control unit or to omit any output. Alternatives can be defined and implemented by the user.
To allow the generation of such fault-tolerance mechanisms, there are several requirements posed to the model. The first requirement is related to voting: there must be some mean to achieve replica determinsm [19]. To allow also the automatic realization of reintegration, the state of a unit must be automatically determinable. Therefore, the system's state and the system's behavior must be separated within the model.

5. Models

As seen in the previous section, several requirements are posed to the model:

1. Suitability for replica redundancy
2. Separation of functional behavior and state
3. Clear execution and timing semantic
4. Details about platform: hardware architecture, operating system, programming language

Different techniques are used to meet these requirements, which will be described in the following. Since the system must be described in several dimensions, we have chosen to split up the model into four distinct sub models: an application model describing the software architecture, a system model to describe the hardware architecture, a fault model containing the assumptions about faults and the fault-tolerance model describing the fault-tolerance mechanisms and the reactions to faults. In the following each model is explained in more detail and the purpose of the models is illustrated by the use of a simple control application, implementing a PID controller on a TMR architecture.

5.1. Application model

The application model contains information about the tasks of the application, the temporal requirements, the communication and the input and output. The main requirements as stated above concerning the application model are the need to accomplish replica redundancy within this model and a clear execution and timing semantic.
These requirements are accomplished by the use of the time-triggered paradigm [13]. In such models, all executions are triggered by the advancement of time rather than by internal or external events. The developer has to specify the points in time when tasks should logically start, by reading some input, and when they should logically stop their computation, by publishing the results. The concrete scheduling of the tasks itself becomes transparent to the user. Since all communication is also performed in a time-triggered manner, race conditions can be excluded by design and violations can be automatically detected.
Replica determinism can be achieved by using the knowledge about the execution times [20] : at specific points in time a deterministic behavior of the system is guaranteed, while between these points in time, the process execution and scheduling can be carried out in different ways on the individual units.
The time-triggered paradigm has also the advantage that there are previously known points in time, when the execution of voting and temporal synchronization algorithms has to be performed. This is the prerequisite for a successful implementation of distributed voting and synchronization algorithms. The messages in the time-triggered communication can also be used for the synchronization algorithm as required: by means of the expected and the actual arrival time of the voting messages, a logical global clock can be

computed [15, 22]. The precision of the temporal synchronization is limited by the maximal network message delay and by the precision of the system clock. Within our tests, we achieved maximal synchronization errors below 200 μs using standard Ethernet.

Another requirement on the model was to support automatic state synchronization and voting. The state synchronization functionality is necessary to allow a repaired unit to reintegrate into the system during system execution. The generated run-time system supports the state synchronization and voting by separating the functionality of the application, reflected in the concept of tasks, from the application's state, reflected in the concept of ports. Thus, these states can be simply compared during voting, while an integration is possibly by copying the state of a fault-free unit to the integrating unit.

To support simplicity and a fast learning process, the model consists of only seven different objects that are explained in the following.

Port All communication in the application model is performed via ports. A port is a unique space in memory with a predetermined size and a specified representation. The values of the ports represent the state of the electronic control units. Therefore, the voting on units and the state synchronization can be based on the values of these ports. For a successful synchronization, it is required that the application functions have no internal states. An easy example is a PID controller: while such a controller is often implemented with two internal variables to store the current integral and derivative part, these variables must be realized by using ports in our model. Otherwise a generic state synchronization could not be achieved.

In the following, the attributes of ports are described. Ports are persistent, that means a port keeps its value over time until the port is updated. The update access is performed time-triggered and has to be deterministic: the access times are previously known and simultaneous write accesses by more than one unit are not allowed. This condition is checked by the code generator while parsing the functional model and in addition at run-time.

Task Tasks are periodically called functions that realize the actual functionality of the application. Tasks can be executed in parallel, but an interaction between the executions of two tasks is not allowed, since this would contradict the requirement for determinism in the execution. Since synchronization points are not allowed, the task functions represent simple sequential programs. Thus, the implementation of these task functions is simplified and race conditions are excluded by design.

The communication of the tasks between each other and with the environment is exclusively performed via ports. The access of tasks on ports occurs in a time-triggered manner: at the beginning of every invocation the task reads the values of its input ports, at the end of the invocation the results are written into the output ports of the task. Here, the begin and the end refers to the invocation period as specified in the functional model. The port access is realized by the generated run-time system and is performed in logical zero time.

The actual execution of the task on the CPU is scheduled by the run-time system and is transparent to the developer. Nevertheless, the developer has to guarantee that the worst-case execution times (WCETs) of the tasks allow a completion of the tasks satisfying the temporal restrictions as specified in the functional model.

Sensor and Actor Sensors and actors realize the communication of the application with the environment and should not be mistaken for the hardware devices. Sensors are functions that are executed to read values from the environment and to write these values into ports, actors are functions to read values from the port and write these values to the environment.

The execution of the sensor and actor functions is also performed time-triggered. The execution frequency has to be specified by the developer. The sensor execution takes thereby place at the begin of each interval, the actor execution at the end of each interval. Both executions are regarded as instantaneous. To legitimate this assumption the functions must represent short sequential code without synchronization points and blockages. For example, in case of a network device the sensor function may check the arrival of a message and copy the message into a port, but a blockage until the receive event of a new message is not allowed.

Mode Applications can have different operation modes. To support this feature, our model includes modes. A mode is a set of tasks, sensors and actors that is currently executed. In addition, a mode cycle duration is assigned to every mode. Within each mode cycle the tasks, sensors and actors are executed according to their frequency as specified in the mode declaration.

Modechange To enable the switch between different operation modes, modechanges can be used. A modechange is a function implemented by the developer that evaluates, if a mode should be switched or not. The developer has to specify the target mode and a non-empty set of source modes within the modechange declaration. The evaluation of the function, which is based on the values of the assigned ports, takes place always at the end of the source mode cycles. Mode switches must be deterministic; this means that for

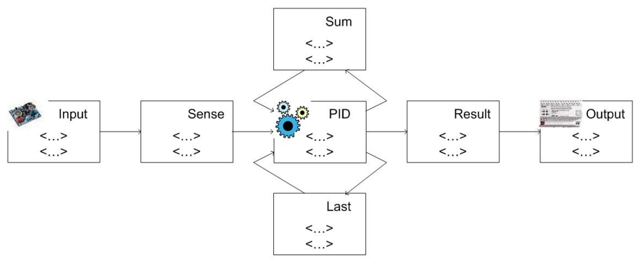

Figure 3. Application model: a PID controller

every achievable configuration (port values and modes) at most one assigned modechange can reach a positive evaluation for a modechange. This condition is checked the generated run-time system at run-time.

Guard Guards are another possibility to change the behavior of the application. Guards are similar to modechanges functions based on port values, but while modechanges should be used for different operation modes, guards can be used to control individual tasks. For this reason the guard is assigned to a certain task. At the begin of every invocation of this task, the guard function is evaluated and only in case of a positive evaluation the according task is started. The main advantage of guards over modechanges is their flexibility. A guard can be used also within a mode cycle and not only at the end of the mode cycle as in the case of modechanges.

Example An exemplary model of a control application implementing a PID controller is depicted in figure 3. A sensor is executed every 1ms and the result of the invocation is written into the port $senseResult$. The task performing the PID control algorithm reads the value of this port as well as two other auxiliary ports, used for the calculation of the integral part and the derivative part, and writes the results to the result port and updates the auxiliary ports. An actor outputs the value of the result port periodically.

5.2. System model

The system model describes the hardware architecture of the system including the electronic control units, the network, the sensors and actuators. The model contains most of the information that is necessary for the appropriate template selection. Due to the vast heterogeneity, this model is also the model that needs to be augmented most likely for future applications.
The model contains amongst others, the used operating systems and programming languages, the timer capabilities of the control units, network settings like IP addresses and ports for Ethernet and so on. An easy extensibility is guaranteed by an extension guide containing several examples.
Figure 4 shows an example for a triple modular redundancy (TMR) architecture: three redundant electronic control units (ECU), each equipped with a sensor and one actuator (single point of failure) for the output. Within the description of the actuator the developer can specify that the actuator has three input channels with exactly once semantics. This means that no voting is performed by the actuator and that only one connected electronic control unit must perform the output at a time. Therefore the only task of the actuator is to delegate the output value to the environment, the voting and the guarantee that exactly one correct value is send to the actuator must be realized within the automatically generated software.

5.3. Fault model

Within the fault model, all possible faults and their effects on the system are specified by the developer. In addition, the fault model contains also information which faults should be tolerated. If possible, the developer can also state the probability of the different faults. To simplify this specification typical faults are suggested to the developer on the basis of the system model. The fault model covers only faults that can be detected at system level. Therefore, identifying the task that causes the fault is the minimal granularity of faults that can be achieved within the application code. Other fault sources are the communication, missed deadlines and errors related to the sensors and actuators.
For our PID controller example, we assume that all single faults within one sensor or electronic control units must be tolerated. In addition, the use of an unreliable network medium is indicated: within each communication round one message loss should be tolerated. The actuator is assumed to be error-free.

5.4. Fault-tolerance model

Within the fault-tolerance model, the developer can specify mechanisms used to achieve fault tolerance. The initial mechanisms to detect defects are plausibility checks, check points and voting. Typical fault-tolerance mechanisms are hot- and cold-standby, fault masking and backward recovery. To repair erroneous units the mechanisms that are realized within the generated code are the restart of the system or to reboot the unit. Additional mechanisms can either be added by augmenting the model and implementing appropriate templates (in case the functionality can be reused for several applications) or be implemented within the application code.
Since replica non-determinism can also be the result of

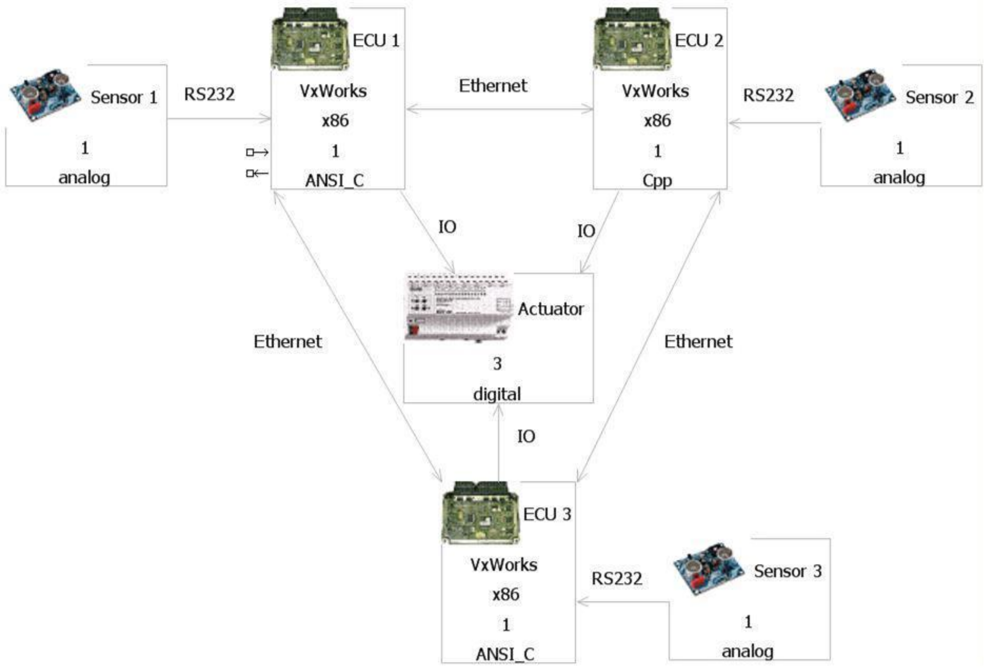

Figure 4. System model: a TMR system

small clock differences (since the synchronization algorithm can only guarantee a deviation of the local clock from the global clock smaller than ε), of N-Version programming or of imprecision in results of measurement, the correct port values are typically situated in a small interval. To overcome this problem, interval decisions can be used for voting.

The fault-model and the fault-tolerance model can be checked automatically for consistency by checking if all faults that should be tolerated are covered by the applied mechanisms. The specification of the applied fault-tolerance mechanisms involves also the combination of the application and the system model. All software objects (tasks, sensors, actors) are assigned to the appropriate hardware object (ECU, sensor, actuator). Depending on the applied fault-tolerance mechanisms, some software objects may be assigned to more than one hardware object. In our example the sensor and task execution is performed redundant on the three ECUs of the TMR-system. Similarly, the voting is realized as a distributed algorithm. It is executed on all three ECUs and performs two communication rounds to cope with the unreliable network.

6. Conclusion

Within this paper, we have presented an approach to use model-based development and code generation also for system level code. One main goal of our approach was the extensibility of both the underlying meta-model, as well as the code generation ability. By the use of a meta-code generator and template-based code generation this goal could be accomplished.

Since standard modeling languages like UML lack the precision and rigor to generate code, a domain specific model had to be designed. The usage of the time-triggered architecture was very useful to achieve the main requirements posed to enable an automatic code generation of fault-tolerance mechanisms: determinism in the execution, replica determinism and the possibility of state synchronization and automatic voting.

Besides the application model, it was also necessary to use other models describing the hardware and the fault-tolerance mechanisms that should be applied.

The approach is already tested in some lab applications. We have implemented for instance the described PID application to control a rod by switched solenoids with control response times of 1ms. The automatic code generation rate reached up to 95% of the total code [4].

For this realization, a sub set of the current system was used. The model consisted only of the application model and some minor information about the platform. As fault-tolerance mechanisms, we implemented voting and failure masking on a triple-modular redundancy system. The used templates are designed for the operating system VxWorks and the programming languages C and C++.

Currently we are implementing additional templates that allow also the generation of other fault-tolerance mechanisms as already foreseen in the model. In addition, the approach will be tested for feasibility by applying the code generator in industrial development processes. This latter project is funded by the German ministry for education and research BMBF.

References

[1] P. Barnard. Software Development Principles Applied to Graphical Model Development. In *AIAA Modeling and Simulation Technologies Conference and Exhibit, San Francisco*, Aug. 2005.

[2] G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[3] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The fractal component model and its support in java. *Software - Practice And Experience*, 36(11-12):1257–1284, 2006.

[4] C. Buckl, A. Knoll, and G. Schrott. Model-based development of fault-tolerant embedded software. In *Second International Symposium on Leveraging Applications of Fomal Methods, Verification and Validation (IEEE-ISoLA)*, pages 113–120, 2006.

[5] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: a declarative language for real-time programming. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 178–188. ACM Press, 1987.

[6] A. Deursen, P. Klint, and J. M. Visser. Domain-specific languages. Technical report, CWI (Centre for Mathematics and Computer Science), 2000.

[7] S. Efftinge, M. Volter, A. Haase, and B. Kolb. *openArchitectureWare - Documentation*.

[8] J. Fassino, J. Stefani, J. Lawall, and G. Muller. Think: A software framework for component-based operating system kernels. In *Usenix Annual Technical Conference, Monterey (USA)*, Monterey (USA), June 2002.

[9] O. M. Group. *OMG Unified Modelling Language Specification*, 1.3 edition, June 1999.

[10] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Embedded control systems development with giotto. *Proceedings of the International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 64 – 72, 2001.

[11] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the First International Workshop on Embedded Software (EMSOFT)*, pages 166 – 184, 2001.

[12] I. Johnson, C. Snook, A. Edmunds, and M. Butler. Rigorous development of reusable, domain-specific components, for complex applications. In *CSDUML'04 - 3rd International Workshop on Critical Systems Development with UML*, 2004.

[13] H. Kopetz and G. Bauer. The Time-Triggered Architecture. *Proceedings of the IEEE*, 91(1):112 – 126, Jan. 2003.

[14] H. Kopetz, G.Grnsteidl, and J.Reisinger. Fault-tolerant membership service in a synchronous distributed real-time system. In *Dependable Computing for Critical Applications*, pages 411–429, 1991.

[15] L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *J. ACM*, 32(1):52–78, 1985.

[16] P. A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1990.

[17] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering*, 26(1):70–93, 2000.

[18] J. Miller and J. Mukerji. *MDA Guide*. Object Management Group, Inc., June 2003. Version 1.0.1 (omg/03-06-01).

[19] S. Poledna. Replica determinism in distributed real-time systems: A brief survey. *Real-Time Systems*, 6:289–316, May 1994.

[20] S. Poledna, A. Burns, A. Wellings, and P. Barrett. Replica determinism and flexible scheduling in hard real-time dependable systems. *IEEE Transactions on Computers*, 49:100–110, Feb. 2000.

[21] S. Sastry, J. Sztipanovits, R. Bajcsy, and H. Gill. Scanning the issue - special issue on modeling and design of embedded software. *Proceedings of the IEEE*, 91(1):3–10, 2003.

[22] U. Schmid and K. Schossmaier. Interval-based clock synchronization. *Real-Time Systems*, 12(2):173–228, 1997.

[23] C. Szyperski. *Component Software*. Addison-Wesley Professional, nov 2002.

[24] TTTech Computertechnik AG. Time Triggered Protocol TTP/C High-Level Specification Document. 2003.