Model-Based Development of Fault-tolerant Embedded Software

Christian Buckl and Alois Knoll and Gerhard Schrott
Robotics and Embedded Systems
Department of Informatics
Technische Universität München

Email: {buckl,knoll,schrott}@in.tum.de

***Abstract*— Model based development has become the state of the art in software engineering. Unfortunately there are only few model-based tools available for the design of fault-tolerant embedded software: while there exist many different code generators for application code, the generation of system aspects like process management, communication in a distributed system and fault-tolerance mechanisms is very complex due to the heterogeneity of the embedded systems.**

We think that the design of an all-embracing code generator, that supports a priori all platforms (the combination of hardware, operating system and programming language) is impossible. Rather it is necessary to concentrate on a code generator architecture that allows an easy extension of the code generation ability.

In this paper we present one possible solution: generating the code on the basis of templates, that solve different recurring aspects of safety-critical embedded software. By the use of a technique similar to preprocessor macros, these templates can be implemented in an application independent fashion. The code generator can then adapt these templates to the application by extracting the necessary information out of the model provided by the application developer.

A first realization of this approach is also mentioned in this paper.

I. Introduction

The development processes for classical software and fault-tolerant software differ significantly. Fault-tolerant software is typically embedded within a distributed system consisting of different hardware and the software has to deal with diverse sensors and actuators. In many cases even different operating systems are applied within one embedded system. This heterogeneity leads to the fact that classical approaches of software engineering, like the model based paradigm [1], can not be applied [2]. Especially tools that allow an automatic code generation or are based on libraries are typically not suited for this heterogeneity. While there exists a huge variety of tools that cover the functional aspects of applications, e.g. Matlab/Simulink [3], there are only very few tools available that cover system aspects, like communication within a distributed system, process management and scheduling or fault-tolerance mechanisms. In addition these tools are typically designed only for a limited number of platforms. To our understanding a platform is the combination of hardware, operating system and programming language.

On the other hand there is a great need for exactly this kind of code generation. Since safety-critical software typically encapsulates domain expertise the software is often designed by engineers who are experts in the application domain and not in the domain of safety critical software and real-time systems. Therefore tools that help to generate automatically large parts of the system are desirable.

It is obvious that it is impossible to design such a code generator that supports a priori all platforms. Rather the code generator must be designed in a way that the generation ability is easy to extend, even by the user. In this paper we present such a code generation architecture: we separate the code generator functionality from the code generator core. The generation functionality is based on templates that can be edited or extended; even new templates can be added. A single template can provide a solution for an aspect of the system or can be used to combine other templates to form a run-time system. Examples for the different aspects of the system are templates for fault-tolerance mechanisms (e.g. voting), process management (scheduling, inter process communication), mechanisms within the distributed system (communication, temporal synchronization) and time or event management. The code generator core can adapt the templates to the application requirements by extracting the necessary information out of the application model specified by the application developer.

We provide such a code generator with a basic set of templates in the project Zerberus [4]. In case the provided templates are not applicable for the desired application due to platform constraints, new templates can be added or existing ones can be changed. In many cases changing a template can be done very easily. One example is the usage of another operating system. In case only standard system calls are used within the component, the changes are typically restricted to the renaming of the system calls and the adaptation of the arguments. The possibility to add new or change existing templates leads to a maximum of flexibility and simultaneously allows a maximal rate of automatic code generation. By generating source code rather than machine code a possible certification of the applications is simplified. Following the certification guide lines, like DO-178B [5], a code generator has to be certified to avoid the certification of the generated code. Since a certification of machine code is very complex, such a certification should be

978-0-7695-3071-0/07 $25.00 © 2007 IEEE
DOI 10.1109/ISoLA.2006.22

IEEE computer society

avoided. By generating source code the certification becomes much easier. The mapping to machine code can be done by the use of existing qualified compilers.

In the following sections our approach is discussed in more detail. Section II gives a summary of related approaches to develop safety-critical embedded software. Afterwards we give more details about our approach, the template-based code generation, in section III and describe one first realization in section IV. The paper is summarized in section V and future work is described.

II. Related Work

Different research projects focus on the issue of modeling and design of embedded fault-tolerant software. Unfortunatly most of these projects are restricted to a specific platform and therefore the application range is limited [6].

One of the most successful projects in the domain of fault-tolerant applications is the time-triggered architecture (TTA) [7]. 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 [8]. The approach is based on a hardware solution, the so-called TTP/C controller [9], running the time-triggered protocol (TTP) 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. This constraints the application area.

Another approach is to use libraries that provide functions to solve recurring problems in the domain of fault-tolerant computing, like synchronization and voting. One representative of this approach is Erlang [10]. Erlang is a programming language designed for programming real-time control systems. The language offers many features that are more commonly associated with an operating system than a programming language like concurrent process, scheduling or garbage collection. Fault-tolerance, fail-over, take-over is built right into the platform and concurrent processing is one of its strengths. A disadvantage of Erlang is the necessity to use Erlang as programming languages. Like other approaches based on libraries the restriction on a specific programming languages reduce the options for the implementation. Another big disadvantage of libraries are problems concerning a certification. For the certification process the source code of the libraries must either be available or the libraries must be already certified. But since the requirements regarding the certification differs for each application area [11], the existence of a certification in the specific application area is very unlikely.

The issues of certification are not considered in most approaches based on code generation. A qualifiaction of the code generator is desirable since this would limit the certification effort to a certification based on the model and the application functionality. Unfortunatly the code generators are typically very complex systems that are very hard to verify. Within our approach the code generation functionality lies within the templates, while the code generator itself does only perform simple adaptations. Thus a certification of the code generator becomes easier and the code generation ability can be easily expanded by introducing new templates.

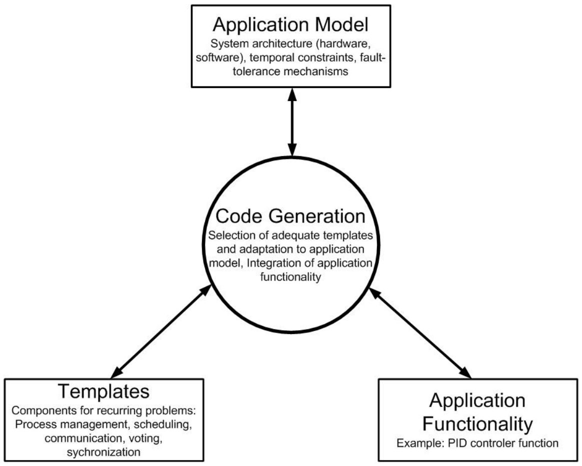

Application Model
System architecture (hardware, software), temporal constraints, fault-tolerance mechanisms
Code Generation
Selection of adequate templates and adaptation to application model, Integration of application functionality
Templates
Components for recurring problems: Process management, scheduling, communication, voting, sychronization
Application Functionality
Example: PID controler function

Fig. 1. Code Generation Architecture

III. Template-Based Code Generation

As already mentioned in the preceding section there are many recurring problems in the context of fault-tolerant embedded software like process management, scheduling, communication or fault-tolerance mechanisms. Solutions for these problems already exist but the heterogeneity of embedded systems contradicts the attempt to reuse components solving these probems.

In this paper we present a solution to this issue, called template-based development. Instead of generating machine code directly from an application model like other approaches or trying to reuse preimplemented components/libraries, we use application-independent templates that can be automatically adapted to the application requirements on the base of the model. The big advantage of this approach is the flexibility regarding the code generation since the generation ability can be extended very easily by new templates. In case a new platform should be supported, existing templates can be adopted to this platform very often with little effort. For example in case a new operating system needs to be supported, the changes are typically restricted to the adaptation of the system calls. A simple example for such a template is depicted and explained in section IV-D.

Templates are already used in many other areas of development processes: one example is the development of graphical user interfaces. State-of-the-art development tools allow a graphical design (model) of the GUI. The developer can modify the design by drag-and-drop functionality and specify the actions, e.g. the effect when a button is pressed. Subsequent the development tool can automatically generate source code

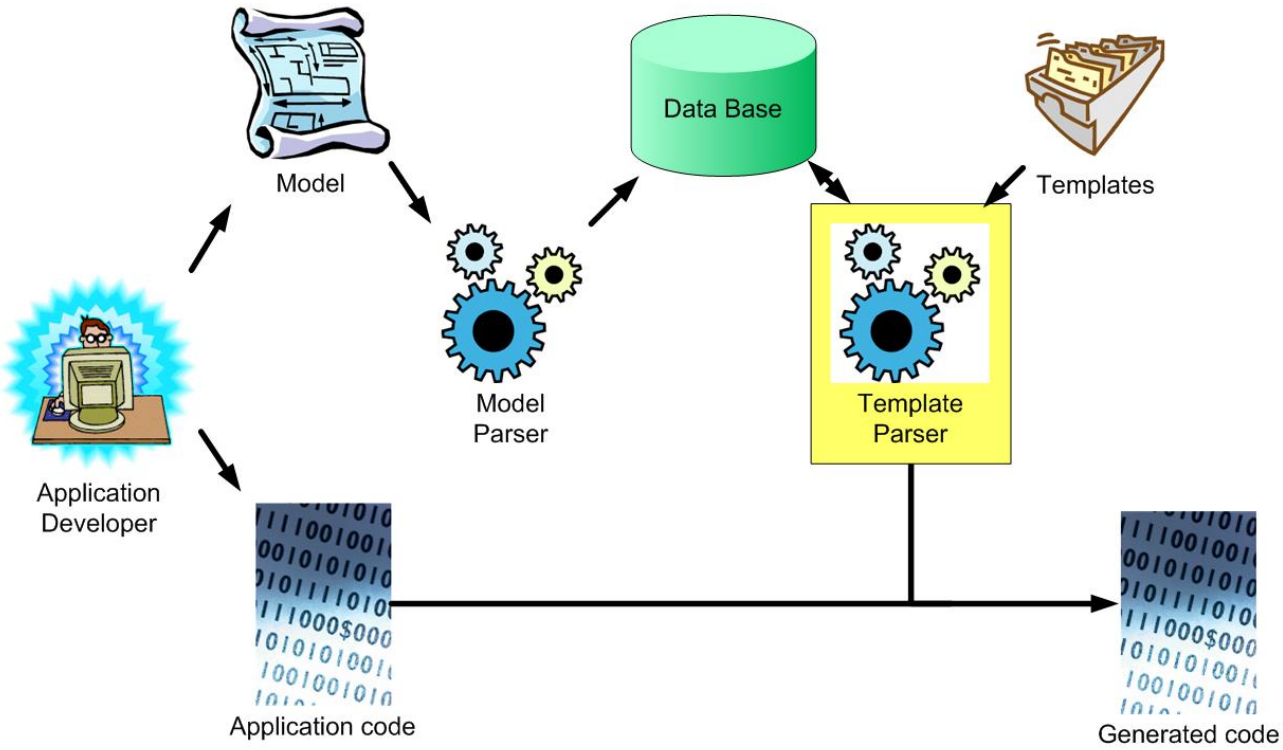


Fig. 2. Code Generation Process

out of the graphical design/model that the developer can modify to adopt to specific application-dependent problems. Another example is the generation of class templates out of UML class models.

In comparison to other approaches template-based development has the advantages that is is not restricted on a specific platform, that all recurring problems within the domain of fault-tolerant embedded software can be addressed and that there are in principle no constraints regarding the application area.

The code generation process is depicted in figure 2. In the first step, the application developer has to specify the application model. This model contains for example information about the software architecture, the tasks of the application, the communication between tasks and the environment and timing constraints. In addition information about non-functional system aspects like the used hardware, operating system, programming language and network settings are neeeded for an appropriate selection of the right templates. We provide a model parser to check the correctness of the model and store the information within a database. There are also some tests performed to check whether the required templates are available or whether the developer has to implement new ones. The second step for the application developer is then to implement the application code, like control functions. This job can be also performed by using other model-based tools like Matlab/Simulink. Finally the template parser adapts the templates with the information contained in the database and combines the generated code with the code implemented by the application developer. The result of the generation process is source code that can be compiled and executed immediately on the desired platform.

IV. Zerberus

Within the Zerberus project [4] we have developed software engineering tools that exemplifies the realization of our approach. As hardware architecture we have chosen a triple-modular-redundancy (TMR) system built with standard components. This architecture allows the application of voting and failure masking as fault-tolerance mechanisms. The main advantage of these mechanisms is the possibility to implement them in an nearly application-independent way. In addition all error types can be covered if N-Version programming techniques and hardware diversity are applied [12].

The intended applications are simple control applications with real-time constraints, that could be implemented in a non-fault-tolerant way on a standard computer. Applications we have in mind are for example the control of wind mills, of industrial robots or control applications in the medical domain.

The main goal of Zerberus is to reduce the development effort to the implementation of the pure application functionality. The realization of the fault-tolerance mechanisms, as well as the process management (timing, scheduling) and the communication between processes, is realized automatically by pre-implemented templates (run-time systems) that are adopted to the application by the Zerberus code generator.

The adaptation is based on a functional model that must be described by the application developer. The functional model contains in Zerberus the specification of the application tasks, the interaction between these tasks, the I/O of the system, as well as the timing constraints. Zerberus provides with the Zerberus Language [13] a possibility to describe that functional model.

In the next two subsections we give a short introduction into the Zerberus Language and we explain the applied fault-tolerance mechanisms.

A. Zerberus Language

The Zerberus Language allows a simple specification of the functional model. For an appropriateness for the use with failure masking and voting several requirements are posed on the language. First of all the language must be suited for replica determinism. This is a non-trivial issue since different platforms and implementations of the application can be used within one system. To achieve replica determinism nevertheless, the Zerberus Language is based upon the time-triggered paradigm [7]. Our approach resembles Giotto [14], a time-triggered language used for the specification of distributed real-time systems. In contrast to our approach applications in Giotto are interpreted on two virtual machines: the embedded and the scheduling machine, while in Zerberus executable code is generated. Another difference is the focus of the two projects: Giotto concentrates on distributed systems, while we are focussing on fault-tolerant applications. Due to this differences an automatic generation of fault-tolerance mechanisms is not foreseen within the context of Giotto.

Using the time-triggered approach, replica determinism can be achieved by using the knowledge about the execution times [15] : 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 have to be performed. This is the prerequisite for a successful application of distributed voting and synchronization algorithms.

The second requirement on the language is the support of an automatic state synchronization and voting. This state synchronization is necessary to allow a repaired unit to reintegrate into the system during system execution. Zerberus supports the state synchronization and voting by separating the functionality of the application from the application's state. 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 language consists of only seven different objects that are explained in the following (a comprehensive description can be found in [13]):

Tasks represent the application functionality, e.g. a control function, and consist of sequential code that is executed in a time-triggered manner. All tasks are executed periodically and the developer can specify the logical start and end time. At the logical start the tasks reads the inputs and at the end of the logical execution the results are output. The actual execution of the task on the CPU is scheduled by the Zerberus runtime system and is transparent to the developer. The input and output of the tasks is performed by using **ports**. A port is a global variable that can be accessed in time-triggered manner. The values of the ports represent the application state and can be therefore used for voting and integration.

The interaction of the system with the environment is also performed via ports. While **sensors** are functions to read inputs from the environment and store these in ports, **actors** are functions to output values of ports to the environment. Both sensors and actors are also executed time-triggered.

To allow also an adaptation of the applications behaviour to the applications mode, **modes**, **modechanges** and **guards** can be specified. Using these mechanisms the execution of the tasks, sensors and actors can be steered.

Example: functional model for a PID-controller For illustration purpose we use the functional model of a PID controller, see fig. 3. The PID controller uses the results of a sensor that is invoked every millisecond and that stores the result in the sensor result port. In addition the PID controller uses the values of two ports to calculate the differential part and the integral part. The results of the PID controller execution are written to the result port as well as to the ports used for the integral and differential part. Within this example we assume that the set point is constant and can be therefore stored within the controller function. In case the developer also wants to have the possibility to change this set point, he would have to use one additional port. The real Zerberus code is depicted in figure 4.

B. Fault-Tolerance Mechanisms

Based on the TMR architecture, Zerberus realizes fault-tolerance mechanisms like failure masking and voting. The hardware redundancy allows the toleration of one arbitrary failure within one of the redundant units. Design errors within the software can be tolerated if N-Version programming is applied. Although N-version programming is typically not applied due to the high development costs, the restriction to the implementation of only the application-dependent code by using Zerberus makes N-Version programming a matter of choice. In case the output is performed by only one unit at a time, errors within the actuator can not be tolerated. We assume within our application that the actuator meets the safety requirements, but nevertheless we have realized a functionality to supervise the output, so that at least reactions to output errors by the system are possible.

The realization of the fault-tolerance mechanisms is based on well-known algorithms and we realized functions for voting, exclusion of erroneous units, reintegration of repaired units as well as temporal synchronization.

The voting is performed at least everytime before the system performs an output, but the developer can also specify a higher

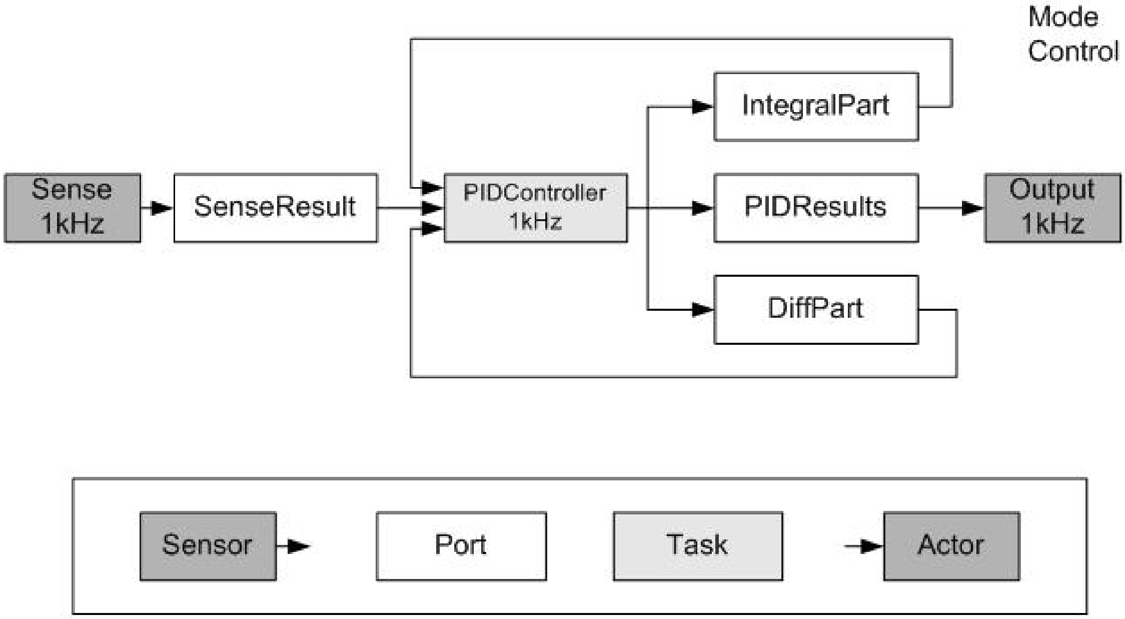


Fig. 3. The functional model of a PID controller: a graphical notation

voting frequency. The voting itself is executed in two rounds to allow the usage of unreliable communication channels: in the first round each computer sends the state information (values of the ports, the current mode and the mode unit) to the other computers. To limit the network traffic the developer has also the chance to restrict the number of transmitted ports. The received state information of the other redundant units are compared with the own state. Due to timing differences within the allowed temporal synchronization interval and to measurement errors the values of the ports may not be deterministic. To handle this issue, Zerberus also supports interval voting for ports. In case interval voting is applied, the developer has to specify the valid bounds for a specific port. The results of the voting are transmitted to the other units within the second round, thus enabling the reconstruction of missing messages. The results of the voting are the partition of the redundant units into correct and erroneous units, as well as the selection of the unit that has to perform the output, in case only one unit should perform the output. A unit is classified as erroneous in case it does not agree with the majority of votes. In this case this unit is excluded from the execution and can perform application-dependent error recovery algorithms.

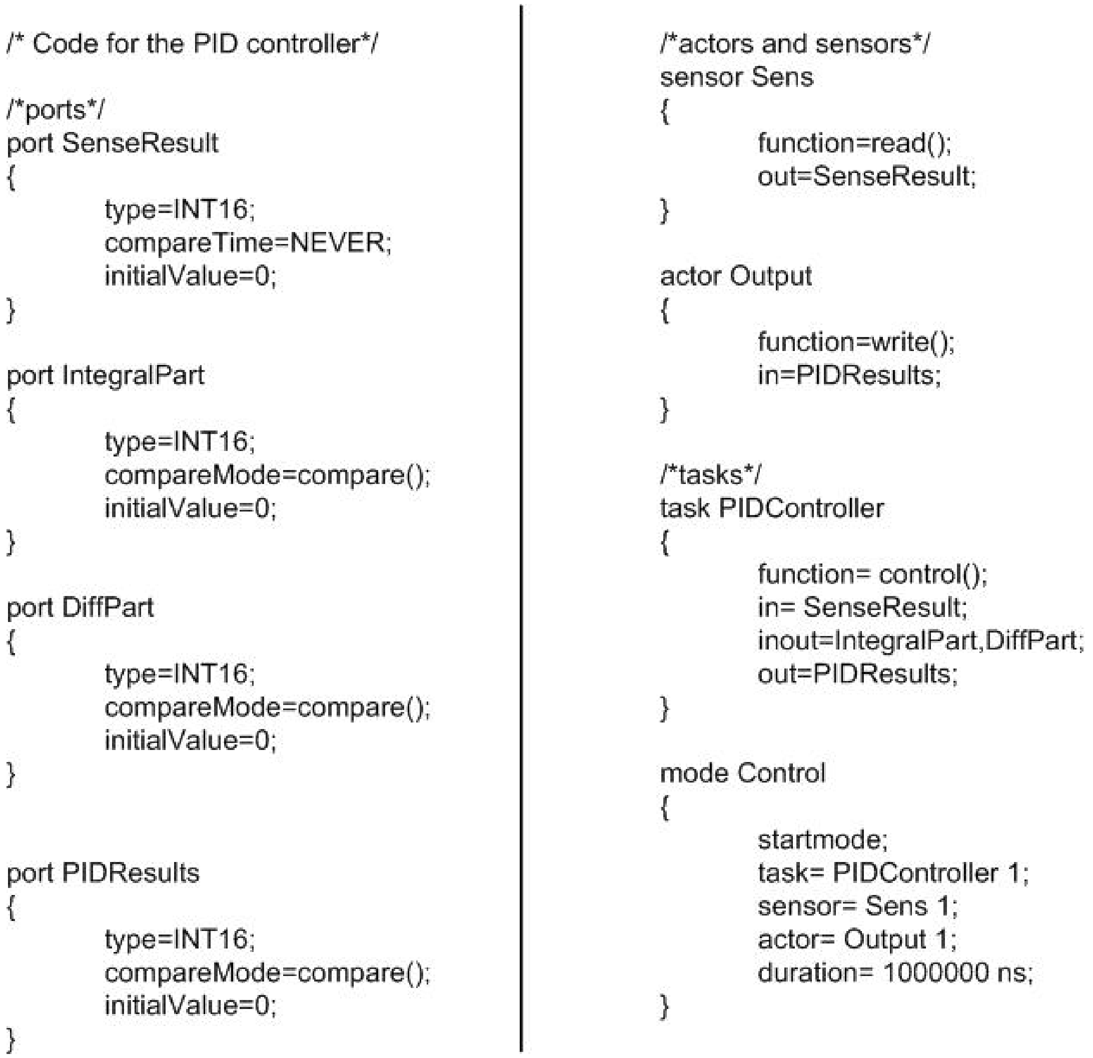
After a successful completion the repaired units can reintegrate into the running system. The reintegration can take place in the next voting round at the earliest by listening to the voting messages and adopting the current application state. In case not all port values are submitted in the voting messages, the integrating unit can also send a request for transmission of the remaining port values. An integration is only allowed if the unit receives consistent states of the majority of units. Since the system state is influenced by the values of the ports and by the results of running tasks, a reintegration is only allowed in case no task is currently running. This is true at the beginning of a new mode round. Both algorithms, the voting and the integration, are based on algorithms suggested in [12].

The temporal synchronization at system start is similar to the algorithm used in TTP [9]. During system execution the voting messages are also used for the synchronization algorithm: by means of the expected and the actual arrival time of the voting messages a logical global clock can be computed [16], [17]. 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.

C. Run-time System

Instead of providing multiple templates that solve parts of the system we developed a combined run-time system template. We currently offer two such run-time systems for the programming languages C and C++ both using VxWorks. These templates can be transformed into application-dependent code during the code generation process.

The run-time system covers all system aspects of the fault-tolerant application: a control layer realizes the reaction to failures within the system, the system layer executes the application as specified in the functional model and executes the fault-tolerance mechanisms like voting, exclusion of erroneous units and reintegration. The communication within the TMR system and the temporal synchronization is performed by the communication layer. Finally the scheduling layer realizes an



```
/* Code for the PID controller*/

/*ports*/
port SenseResult
{
        type=INT16;
        compareTime=NEVER;
        initialValue=0;
}

port IntegralPart
{
        type=INT16;
        compareMode=compare();
        initialValue=0;
}

port DiffPart
{
        type=INT16;
        compareMode=compare();
        initialValue=0;
}


port PIDResults
{
        type=INT16;
        compareMode=compare();
        initialValue=0;
}
```

```
/*actors and sensors*/
sensor Sens
{
        function=read();
        out=SenseResult;
}

actor Output
{
        function=write();
        in=PIDResults;
}

/*tasks*/
task PIDController
{
        function= control();
        in= SenseResult;
        inout=IntegralPart,DiffPart;
        out=PIDResults;
}

mode Control
{
        startmode;
        task= PIDController 1;
        sensor= Sens 1;
        actor= Output 1;
        duration= 1000000 ns;
}
```

Fig. 4. The functional model coded in the Zerberus Language

Earliest-Deadline-First scheduling [18], [19] of the application tasks realized in the task layer.

D. Zerberus Tags

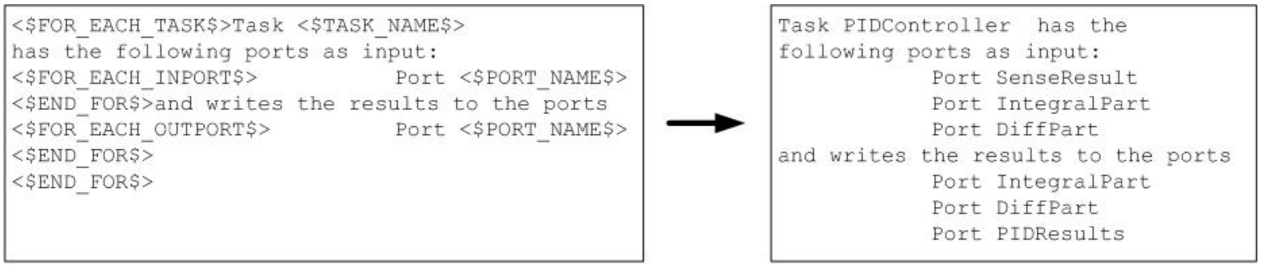
We use a technique similar to preprocessor macros to allow the implementation of application-independent templates. All application-dependent data is replaced by so-called Zerberus Tags. There are two different types of tags: simple tags and control flow tags. While simple tags can be replaced directly by application-dependent data, control flow tags manipulate a code range. Two different control loop tag types are offered: for-each tags and if tags.

Thus a templates consists of source code augmented with Zerberus tags. These tags are replaced with application specific content during code generation. Because the usage of the Zerberus tags is very hard to depict within one simple figure using source code, we use a simple example based on natural language illustrated in figure 5. The use of natural language demonstrates the fact that the tags are not based on a certain programming language, which is also necessary in the context of comments or documentation.

In our example we want to enumerate the different tasks with their ports: by using the <FOR_EACH_TASK> tag, the code until the corresponding <END_FOR> tag is written into the output file for each task available. The effect of tags is always context related: the succeeding <FOR_EACH_INPORT> tag is interpreted in the context of the current task.

E. Simple Control Application Example

We have tested the run-time system in the context of a simple control example: the control of a rod by switched solenoids, see in figure 6. This rather simple application demonstrates the advantages of our approach. The whole code including the functional model as depicted in figure 4 that had to be implemented by the developer consists of less than



```
<$FOR_EACH_TASK$>Task <$TASK_NAME$>
has the following ports as input:
<$FOR_EACH_INPORT$>         Port <$PORT_NAME$>
<$END_FOR$>and writes the results to the ports
<$FOR_EACH_OUTPORT$>        Port <$PORT_NAME$>
<$END_FOR$>
<$END_FOR$>
```

```
Task PIDController  has the
following ports as input:
            Port SenseResult
            Port IntegralPart
            Port DiffPart
and writes the results to the ports
            Port IntegralPart
            Port DiffPart
            Port PIDResults
```

Fig. 5. Zerberus tags and the generated code

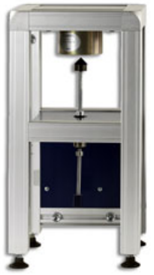


Fig. 6. Rod controlled by switched solenoids

100 lines of code. We achieved control response times of 1MHz with our setup (AMD Athlon processors,ethernet). To achieve better control response times a faster communication medium and a better clock resolution, for example by the use of external timers, would be necessary.

We also used different run-time systems written in C and C++ to demonstrate the possibility to use N-Version programming techniques within Zerberus. The integration was performed smoothly due to the strict adherence of the protocols offered by Zerberus.

V. Conclusions and Further Research

Classical software engineering tools based on code generation, middleware approaches or libraries can not be applied within the context of safety critical embedded software or are limited to only a specific application domain due to inflexibility. The requirements in the context of fault-tolerant systems like generality concerning the usable platforms, automatic code generation of standard system functionality, flexibility in the sense that the system can be adopted to application requirements in all phases of the engineering process and support concerning certification issues are not satisfied by existing tools.

Within this paper we presented an approach to fulfill these requirements. The use of application-independent templates that are mapped automatically to directly compilable source code offers diverse advantages. The templates can be easily adopted and extended or new templates can be implemented to extend the application area. The automatic code generation reliefs the developer of implementing great parts of the system. Certification issues are mitigated since all source code is available to the developer.

A first realization of our approach was done with Zerberus. Based on the simple case of TMR-systems the advantages of our approach could be shown. The next steps within our research will be the extension of our approach to arbitrary distributed system architectures, the modularization of the Zerberus run-time systems and the support of further fault-tolerance mechanisms that are not based on TMR-systems. We are also planning to apply our approach within real industrial projects to point out the feasability.

References

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

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

[3] Barnard, P.: Software Development Principles Applied to Graphical Model Development. In: AIAA Modeling and Simulation Technologies Conference and Exhibit, San Francisco. (2005)

[4] Buckl, C., Knoll, A., Schrott, G.: Development of Dependable Real-Time Systems with Zerberus. In: 11th International Symposium, Pacific Rim Dependable Computing, PRDC 2005, Changsha, China, December 12-15, 2005, Proceedings, IEEE (2005) 404–408

[5] RTCA DO-178B: Software considerations in airborne systems and equipment certification. (1992)

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

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

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

[9] TTTech Computertechnik AG: Time Triggered Protocol TTP/C High-Level Specification Document. (2003)

[10] Armstrong, J.: Erlang — a Survey of the Language and its Industrial Applications. In: INAP'96 — The 9th Exhibitions and Symposium on Industrial Applications of Prolog, Hino, Tokyo, Japan (1996) 16–18

[11] Saglietti, F.: Licensing reliable embedded software for safety-critical applications. Real-Time Systems **28** (2004) 217–236

[12] Echtle, K.: Fehlertoleranzverfahren. Springer Verlag (1990)

[13] Buckl, C., Knoll, A., Schrott, G.: The Zerberus Language: Describing the Functional Model of Dependable Real-Time Systems. In: Dependable Computing, Second Latin-American Symposium, LADC 2005, Salvador, Brazil, October 25-28, 2005, Proceedings. Lecture Notes in Computer Science, Springer (2005) 101–120

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

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

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

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

[18] Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment. Journal of the ACM **20** (1973) 46–61

[19] Buttazzo, G.C.: Rate monotonic vs. edf: judgment day. Real-Time Syst. **29** (2005) 5–26