Fakultät für Informatik
der Technischen Universität München
Informatik VI: Robotics and Embedded Systems

Developing dependable real-time systems

Christian Buckl
TU München, 85748 Garching b. München, Germany
buckl@in.tum.de
http://www6.in.tum.de

A growing number of safety-critical systems is controlled by computer systems. In the context of several research projects solutions were suggested how to reduce the implementation effort for dependable real-time systems. Unfortunately most of these approaches are based on special hardware solutions or restricted to specific application domains. In addition most of the application realize only fault-tolerance concerning communication errors, whereas fault-tolerance at application level is left to the developer.
Within the Zerberus project a development process is suggested that is not restricted to a special kind of hardware, operating system or programming language. The main idea of Zerberus is the separation of the application functionality from the fault-tolerance mechanisms. The fault-tolerance mechanisms are then generated automatically by the Zerberus code generator. As basis for this generation a high-level platform independent description of the application provided by the developer is used. This so called functional model describes the application tasks, the interaction between these tasks, the timing constraints and the input and output of the system and can be specified by using the Zerberus language.
Due to the automatic code generation of the fault-tolerance mechanisms the implementation effort is restricted to the absolute application dependent code. The automatic generation of the FT-mechanisms is based on preimplemented, application-independent templates that are adopted to the application during the code generation process. Templates are currently offered for the operating system VxWorks 5.5 and the programming languages C and C++.

1 Introduction

The unit numbers of safety critical computer systems are growing permanently, especially in the market of embedded systems. Although the main fault-tolerance mechanisms are known for more than 20 years [1], there are no development tools available that allow an automatic generation of the fault-tolerance mechanisms.
Instead the application functionality and the fault-tolerance are mixed, thus complicating the whole development process. This leads to a time-consuming and cost-intensive process. Another big disadvantage is the need of developers with expert knowledge in fault-tolerance in addition to the application expert. Especially this need for experts of different domains is one main error source.
Several research projects addressed this problem and tried to find solutions. Unfortunately most of these approaches are based on special kind of hardware or restricted to specific application domains. In addition most of the solutions offer only fault-tolerance concerning communication errors, whereas fault-tolerance at application level is left to the developer.
The research project Zerberus from the chair for Embedded Systems and Robotics of the Technical University in Munich offers a solution without a restriction on a certain platform. A platform int the sense of Zerberus is the combination of used hardware components, operating system and programming language. Within Zerberus a development process with tool support is suggested that enables the application expert to design dependable real-time systems without expert knowledge in fault-tolerance mechanisms.
The main concept of Zerberus is the separation of application functionality from the fault-tolerance mechanisms. While the application-dependent code has to be implemented by the developer, the FT-mechanisms are automatically generated by the Zerberus code generator. As basis for this generation a functional model, describing the application tasks, the interaction between these tasks the temporal constraints as well as the input and output of the system, is used. For the platform independent specification of the functional model, the Zerberus language [2] is introduced.
Due to the automatic code generation of the fault-tolerance mechanisms the implementation effort is restricted to the absolute application dependent code. To achieve the fault-tolerance a triple-modular-redundancy (TMR) system is used. The main advantages are the possibility to implement voting, failure masking, exclusion and reintegration algorithm nearly almost in an application independent way. In addition only TMR systems enable the toleration of all possible error types if hardware diversity (affordable due to the use of commercial-off-the-shelf hardware components) and N-version programming techniques (manageable due to the restriction on application dependent code) are used. The automatic generation of the FT-mechanisms is based on preimplemented, application-independent templates that are adopted to the application during the code generation process. Templates are currently offered for the operating system VxWorks 5.5 and the programming languages C and C++.
This paper is structured as follows: in section 2 other approaches for the development of dependable real-time systems are discussed. Section 3 gives an overview over Zerberus and describes the suggested development steps, the Zerberus language as well

as the used fault-tolerance mechanisms. In section 4 a test application developed with Zerberus is described. The whole paper is summarized in section 5 and possible future work is presented.

2 Other Research Projects

Different research groups have observed the demand for a development process for safety critical real-time systems. Most of these solutions are based on the time-triggered paradigm [3]. The time-triggered approach guarantees one important aspect that is absolutely necessary for fault-tolerance mechanisms: determinism. Due to the application of the time-triggered paradigm, there are previously known points in time, when distributed fault-tolerance mechanisms can be performed.

One important representative for the time-triggered approach is TTP/C [4]. TTP/C, the Time-Triggered Protocol, is a TDMA protocol designed to handle highly dependent real-time applications implemented in distributed networks. The protocol offers clock synchronization, clique avoidance, deterministic message sending and membership services. The TTP/C protocol itself offers nevertheless no built-in fault-tolerance mechanisms at application level. Several other projects, like DECOS [5] addressed this problem, but all these approaches have one major drawback in our opinion: the restriction to special hardware (like TTP/C controllers), programming languages or operating systems.

Our attempt was to design a development process that allows the usage of commercial-off-the-shelf hardware and that has no constraints towards programming languages and operating systems. This approach is shared with the research project Giotto [6], from the University of California at Berkeley. On the one hand, Giotto is based on the time-triggered approach, but on the other hand it also uses results of the research on synchronous languages like Esterel [7]. Like the synchronous languages Giotto introduces an abstraction level that separates the software design process from the actual hardware. By using the concept of FLET (Fixed Logical Execution Times), the applications designed with Giotto are not only deterministic regarding the values of the results (like Esterel, Lustre), but also have a deterministic temporal behavior. Thereto Giotto offers a language for the specification of the platform independent functional model for distributed real-time applications. The mapping of the platform independent functional model to executable code is realized by a code generator. Since Giotto was designed primarily for the use in distributed systems Giotto has no built-in fault-tolerance.Within our project we developed the Zerberus Language, which is based on Giotto, to describe the functional model of the safety-critical system.

Several goals of the Zerberus System are also shared with Erlang [8]. 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. In contrast to the Zerberus System, Erlang

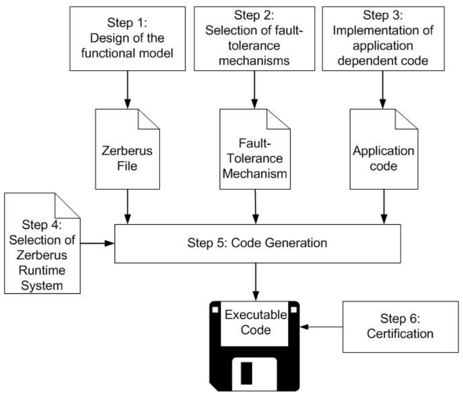

Step 1: Design of the functional model
Step 2: Selection of fault-tolerance mechanisms
Step 3: Implementation of application dependent code
Zerberus File
Fault-Tolerance Mechanism
Application code
Step 4: Selection of Zerberus Runtime System
Step 5: Code Generation
Executable Code
Step 6: Certification

Figure 1: Development steps

was designed only for soft real-time systems. Another difference is the programming extent: while Erlang is used for implementing the whole application, the Zerberus Language is only used for the specification of the functional model. For the implementation of the pure application code the developer can use a common, familiar programming language like C or C++.

3 Zerberus

3.1 Development steps

The development with Zerberus is partitioned in six steps as illustrated in figure 1. In the beginning the developer has to specify the functional model that describes the application tasks, the interactions between these tasks and with the environment as well as the temporal constraints. The specification of the functional model is realized with the Zerberus Language.
In the next step the fault-tolerance mechanisms that should be applied must be chosen. Currently only failure masking on the base of a TMR-system is available. In addition the developer can use hardware diversity and N-version programming to allow also the toleration of design errors. During this step the developer has also to select the used platforms (hardware, operating systems, programming languages).
After the specification of the functional model and the used platform the developer has to implement the pure application dependent functionality. Due to this limitation the application of N-Version programming techniques becomes a matter of choice. The implementation is of course platform dependent and thus must be performed for each used platform.
The functional model and the code implemented by the developer are the inputs of the code generation process. The code generator parses the functional model, extracts all the information necessary and transforms the application independent, pre-implemented run-time system templates into executable code. The run-time system templates are ei-

ther provided by Zerberus or can be implemented by the developer in case he needs to use a platform that is currently not supported. Zerberus offers possibilities to implement own run-time system templates application independent so that it is possible to reuse these templates in other projects.
The result of the code generation process is executable code. Since for most safety-critical applications a certification by a control organization like the German TÜV is needed, Zerberus tries to accelerate this process by the reuse of code. In case the run-time system templates are already certified, the results of this certification process can be reused. Of course this is only true in case the templates are already certified for the desired application domain [9].

3.2 Zerberus Language

The Zerberus Language allows the 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 [3]. Similar to the approach in [10] replica determinism can be achieved by using the knowledge about the execution times: the points in time when the distributed synchronization and voting algorithms take place are known in advance and between these points in time the execution and scheduling of the different processes can be carried out in different ways on the individual Zerberus units.
The time-triggered paradigm has also the advantage that there are previously known points in time when the execution of voting and time synchronization algorithms must 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. During the voting algorithm the state is simply compared. In case of an integration the integrating unit simply needs to copy the state of the running units into its own state.
The Zerberus Language is designed in a very simple way to allow a fast learning process. Due to this reason the language consists of only seven different objects that are explained in the next paragraphs.

Task Tasks represent the actual functionality of the application and are periodically called functions, that can be executed in parallel. The execution of tasks is performed autonomous, that means that no interaction between tasks is allowed. The execution of tasks is performed time-triggered: the developer has to assign a frequency to the task. The task execution starts logically at each begin of the period by reading the inputs and

ends at the end of the period by writing the outputs. The actual execution of the task on the CPU is scheduled by the Zerberus 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.
The tasks must comply to the paradigm of referential transparency like in functional programming. The code must be therefore sequential code without any access to state variables. This demand is derived from the need to separate the application functionality from the application state.

Port The communication between tasks 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 application´s state and can be used for a reintegration of one unit during the system execution.
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. Tasks can read the values of ports at the begin of their execution period. The results of a task are written to the according port at the end of the task period.
Due to small time deviations on the different units, to different algorithm implementation and to the usage of different hardware deviations of the port values on the different units are possible. To enable Zerberus to perform nevertheless a correct voting execution the application developer can specify interval decision functions that are used during voting to determine the correctness of the different units.

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 assumptions 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 the Zerberus Language introduces modes. A mode is a set of tasks, sensors and actors that is currently active on the Zerberus units. In addition, a mode cycle duration is assigned

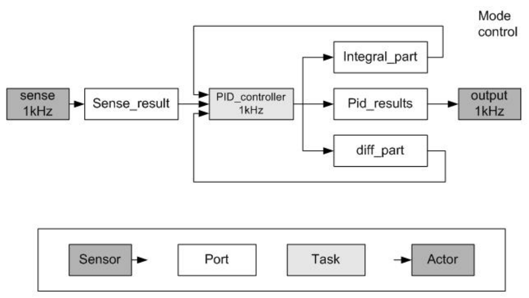


Figure 2: The functional model of a PID controller

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 every achievable configuration (port values and modes) at most one assigned modechange can reach a positive evaluation for a modechange.

Guard Guards are another possibility to change the behavior of a Zerberus program. 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 therefore their flexibility. A guard can be used also within a mode cycle and not only at the end of the mode cycle.

Example: functional model for a PID-controller Figure 2 shows the functional model of a PID controller. 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 that store the deviation read in the last round (to calculate the differential part) and the integral part. The results of every PID controller deviation are written to the result port as well as to the ports used for the integral and differential part.
For this example we assume that the set point is constant and can be therefore stored within the controller function. In case the developer wants also to have the possibility to change this set point, he would have to use one additional port.

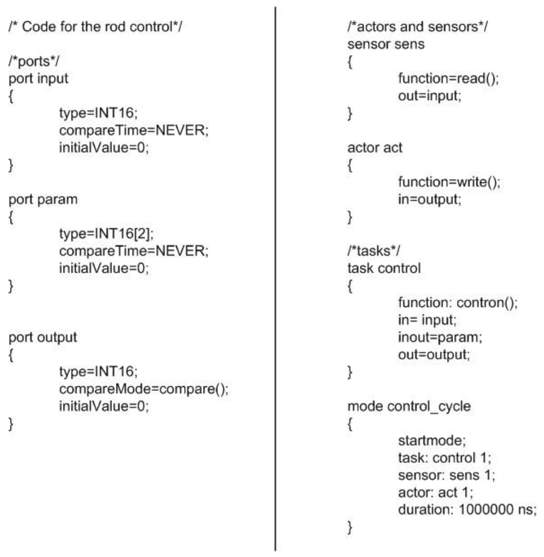
3.3 Fault-tolerance Mechanisms

As already mentioned Zerberus offers currently failure masking and voting as fault-tolerance mechanisms based on structural redundancy. At least three redundant units must be used, but all algorithms are designed in a way that they can be used also with arbitrary n-of-m-systems. This includes voting, exclusion of erroneous units, reintegration of repaired units as well as temporal synchronization. The voting is performed at least whenever the system wants to execute an actor, but the developer has also the possibility to specify a higher voting rate. The voting itself is performed in two rounds to allow also the usage of unreliable communication channels: in the first round each computer sends the values of the ports to the other units. To limit the network traffic the developer has also the chance to restrict the transmitted ports to the ports used by the actor. Each unit performs then the voting algorithms and sends the results in the second round to the other units. Only units that agree with the majority of votes are allowed to perform the output. The decision if all correct units or only one correct unit performs the output can be specified by the user. Units that do not agree with the majority of votes are excluded from the execution and can perform application dependent error recovery algorithms. After a successful completion the repaired units can be reintegrate into the running system by listening to the voting messages and adopting to the current application state. In case not all port values are submitted in the voting messages the integrating unit can also request for the other values. An integration is only allowed if the unit receives consistent states of the majority of votes. Both, the voting and the integration, algorithms are based on algorithms suggested in [11].
The temporal synchronization at system start is similar to the algorithm used in TTP [4]. 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 [12, 13].

4 Experiment

For demonstration we have implemented a system to balance a rod under the control of switched solenoids. For a stable control, sample rates in the range of few milliseconds are necessary. The system's setup consisted of three computers equipped with AMD Athlon processors, connected with switched ethernet and equipped with an AD/DA-board. As real-time operating system we used VxWorks and C as programming language.

The implementation turned out to be very simple. For the description of the functional model (see figure 3) 30 lines of code were needed. The application consists of three ports for the measurement value, the differential and integral part for the controller and one port for the result. In addition a sensor and an actor, as well as one task for the control function had to be declared. The code to implement the application consisted of less than 70 lines. Thus it was possible to implement a fault-tolerant control system with less than 100 lines of code. With our setup we achieved sample rates of



```
/* Code for the rod control*/

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

port param
{
        type=INT16[2];
        compareTime=NEVER;
        initialValue=0;
}


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

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

actor act
{
        function=write();
        in=output;
}

/*tasks*/
task control
{
        function: contron();
        in= input;
        inout=param;
        out=output;
}

mode control_cycle
{
        startmode;
        task: control 1;
        sensor: sens 1;
        actor: act 1;
        duration: 1000000 ns;
}
```

Figure 3: Functional model

1000 Hz.

5 Conclusion and Future Work

We presented within this paper an approach to develop safety-critical real-time applications. The separation of the application functionality from the fault-tolerance mechanisms and the automatic realization of these mechanisms promises a fast and error minimizing development process. The realization of the application on a specific platform is realized by application independent run-time system templates.
Nevertheless there are several possibilities to augment Zerberus to increase the benefit of the developers using Zerberus. One main job for the future is to integrate Zerberus into a complete development tool chain. Currently only a graphical interface for the code generator is offered. The Zerberus Language must be coded using a standard editor. The integration of the Zerberus Language into UML, like [14], or the development of an own graphical development environment to provide a drag-and-drop functionality like in Matlab are desirable and currently planned.
Another possibility to ameliorate Zerberus is to widen the application range from rather simple control applications to distributed systems with some needs of fault-tolerance mechanisms. In case the system becomes more complex the utilization of triple modular redundancy may be too expensive. To provide nevertheless the possibility to use Zerberus, Zerberus must be extended with further fault-tolerance mechanisms to abstain from TMR and must be altered in a way that these fault-tolerance mechanisms can be used in a more flexible way. By using this approach there would be also the possibility to integrate monitoring, logging and user interface units into the system which is essential for the design of larger distributed systems.

To point out the feasibility of Zerberus and to learn more about further improvements, the application of Zerberus in two industrial projects is planned.

References

[1] Pradhan, D.K.: Fault-Tolerant Computer System Design. Prentice Hall (1996)

[2] 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)

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

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

[5] Website DECOS: (http://www.decos.at/)

[6] 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

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

[8] Armstrong, J.: The development of erlang. In: ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming, New York, NY, USA, ACM Press (1997) 196–203

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

[10] 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

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

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

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

[14] Majzik, I., Pintér, G., Kovács, P.T.: UML Based Design of Time Triggered Systems. In: Proc. The 7^{th} IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC-2004), Vienna, Austria (2004) 60–63