Development of Dependable Real-Time Systems with Zerberus

Christian Buckl and Alois Knoll and Gerhard Schrott
Department of Informatics
TU München
Garching b. München, Germany
{buckl,knoll,schrott}@in.tum.de

Abstract

Although the main fault-tolerance techniques are known for a long time, there exists no consistent approach for implementing dependable applications in the sense that the fault-tolerance mechanisms are generated automatically by development tools.

With the Zerberus System we developed such a tool. The main concept of Zerberus is the platform independent specification of the functional model (application tasks, interaction with environment, temporal constraints) by the developer using the Zerberus language. Based on this functional model the code for the fault-tolerance mechanisms is generated automatically for the desired platform using pre-implemented templates.

Due to this automatic code generation the development process is accelerated and the developers are enabled to implement dependable application without expert knowledge of fault-tolerance techniques. Our approach offers also the possibility to accelerate the certification process by using certified templates for the fault-tolerance mechanisms.

1. Introduction

Although the main fault-tolerance mechanisms are known for a long time [3], there exists no general development model especially designed for dependable systems. Therefore most of the systems are built from scratch which leads to a time-consuming and cost-intensive development process. In addition the fault-tolerance mechanisms and the application functionality are often mixed up thus complicating the certification process.

Within Zerberus a development process is suggested to the user that attempts to reduce the development times and costs, while increasing the reliability and safety of the software. The main concept of Zerberus to achieve these features is to separate the functional design of the application from the platform dependent implementation and to provide a set of pre-implemented fault-tolerance mechanisms in the form of templates. The separation is realized by the specification of the functional model of the application. On the basis of the functional model, Zerberus is enabled to generate automatically the necessary fault-tolerance mechanisms. Thus the task of the developer can be minimized to the implementation of the application-dependent code.

The fault-tolerance techniques currently supported by Zerberus are based on structural redundancy. At least three redundant units (in the following denoted as Zerberus units) are executing the application in parallel. To reduce the costs for system design commercial-off-the-shelf hardware components are supported. The system generates automatically facilities for temporal synchronization of the units, voting on the states of the units, exclusion of erroneous units and the reintegration of the repaired Zerberus units.

The focus of this paper is laid on the Zerberus language used for the platform independent specification of the functional model. The functional model of an application specifies the systems tasks, the interaction among each other and with the environment as well as the temporal constraints. Since the functional model is used as base for the realization of the fault-tolerance mechanisms, the main properties required were the suitability for replica determinism and the existence of previously known points in time for the execution of distributed voting and synchronization algorithms. These properties could be fulfilled by the appliance of the time-triggered paradigm. To allow a automatic generation of voting and reintegration algorithms, the functional model must separate the application's state from the application execution.

The paper is structured as follows: first of all related work is summarized in sec. 2. A short overview of the development process is given in sec. 3. In sec. 4 the requirements on the Zerberus language are elaborated and the concepts of the language are described. Sec. 5 describes a case study to point out the benefits of our approach. At the end this paper is summarized and we give a short overview of the planned future research.

2. Related work

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 [6]. The time-triggered approach guarantees the important aspect of determinism of the system execution since all points in time when system components are interacting are known in advance.

One important representative for the time-triggered approach is TTP/C [10]. 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 addressed this problem (MARS [5] or DECOS [11]). 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 [4], 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 [1]. 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), but also have a deterministic temporal behavior. Giotto offers a language for the specification of the platform independent functional model for distributed real-time applications. A platform in the sense of Giotto (and in the sense of Zerberus) comprises the hardware, the operating system and the programming language. 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, based on Giotto, to describe the functional model of the safety critical systems.

3. Development Process

The Zerberus System suggests six steps in the development process for dependable systems. In each step the system assists the developer to accelerate the process (for example by automatic code generation) and to improve the results by tool support or by providing guidelines. The individual steps to produce executable code are illustrated in fig. 1 and are described below. Since for most of the safety-critical systems a certification by an authority is required, this problem is also addressed.

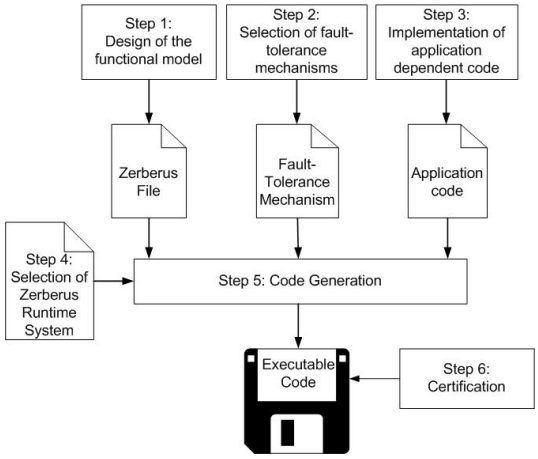

Figure 1. Development process

Specification of the Functional Model Within this step the user has to specify platform independent the functional elements of the application , their relationship towards each other and to the environment as well as the temporal constraints. The specification is realized by the use of the Zerberus Language, which is described in more detail in sec.4.

Analysis of the Requirements on the Dependability Currently Zerberus offers active structural redundancy as fault-tolerance mechanism. At least three Zerberus units compute the application in parallel. At specified points in time the units perform a distributed voting and synchronization algorithm. Erroneous units are excluded from the computation and can perform error recovery algorithms. Since error recovery algorithms are most times application dependent the current run-time systems offers only a restart or a reboot of the system. In addition the developer can specify further fault reactions and recovery algorithms. After a successful completion, protocols allow the reintegration into the running system.

Since a replication of identical units permits no toleration of design errors, the system also supports diversity in hardware and software. While hardware diversity leads to no or only few additional costs as a result of the support of COTS hardware, N-Version programming is often not considered due to the extra effort necessary for the implementation of the individual versions. Since the implemen-

tation effort in Zerberus is reduced to the implementation of the pure application functionality, this disadvantage of N-Version programming can be decreased.

The voting in Zerberus is performed in two rounds to additionally support the usage of a non-reliable communication network and is based on the voting algorithms as suggested by Klaus Echtle [3]. 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 [7, 9]. The initial clock synchronization at start up is based on the algorithm implemented in the TTP/C [10] protocol.

Implementation of Application Dependent Code In this step the developer has to implement the code for the application. As already implied in the previous section this code is restricted to the pure application dependent functionality of the main parts which were identified within the design process of the functional model. By this restriction, the implementation effort can be reduced to a minimum. The implementation step is platform dependent. This implies that for every platform used, the code has to be reimplemented by the developer.

Selection of run-time systems Run-time systems realize the execution of the application on the individual platform and provide the fault-tolerance mechanisms. Several run-time systems are provided by Zerberus, but to guarantee the generality of our approach the developer can also design his own run-time system, e.g. if the desired platform is not supported. All run-time systems are implemented independent of a particular application in form of templates so that for each platform a run-time system must be implemented only once.

Code Generation The transformation of the functional model, the application dependent code and the selected fault-tolerance mechanisms into executable code is performed automatically by the Zerberus code generator. Both the functional model and the run-time systems are parsed by the code generator and syntactic and semantic checks are performed.

Certification of the Zerberus System By using pre-implemented templates for the fault-tolerance mechanisms, the certification effort can be reduced to the certification of the application code implemented by the developer, once the templates and the generation tools are initially certified. For this reason we are currently collaborating with the German control authorization TÜV to get a initial certification for one application and for our tools.

For a successful certification the system must of course apply to the certification standards. These standards differ from the fields of application. In general this means that the system must be re-engineered for each such standard. In case a certification is achieved, the system can be reused for applications of the same domain without a repeated certification of the templates. We intend to achieve such a certification for the medical domain.

4. The Zerberus language

After the overview of the development model the Zerberus language is explained in more detail. While this chapter explains the language in a general way, the syntax of the language is shown in the next chapter 5 in the context of one example. The concrete syntax is specified in a technical report [2]. The main feature of the language to support voting, synchronization and integration algorithms is replica determinism. This is a non-trivial issue since different platforms can be used to achieve fault-tolerance. This includes the simultaneous use of different hardware, operating systems, programming languages and control algorithms in one control system. To achieve replica determinism nevertheless the Zerberus Language is based upon the time-triggered paradigm [6]. Similar to the approach in [8] 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.

To avoid a long learning process the language was designed very simple and consists of only seven different elements described in the following paragraphs.

Port All communication in the Zerberus System is performed via ports. A port is a unique space in memory with a predetermined size and a specified representation. To guarantee the platform independence the possible port types are not based on a specific programming language or hardware. The realization of the different port types for a given platform is the task of the run-time systems.

The values of the ports represent the state of the Zerberus units. Therefore a comparison of the different Zerberus units can be based on the values of these ports. It is required that there are no spaces in memory to store internal states besides the ports. Thus the state synchronization can also be based on the values of the ports during the reintegration of a Zerberus unit. The platform independent specification of the size and the representation of the port values is the foundation to enable the use of N-Version programming using different programming languages and operating systems.

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.

Replica non-determinism can also be the result of 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. Due to these effects the correct port values are typically situated in a small interval. To support this fact the comparison of ports can also be based on an interval decision. This can be done by declaring a voting function for the port that has to be implemented by the developer. In case no voting function is specified the voting of the port values is based on the bit-by-bit comparison.

The voting on the value of a specific port takes place at least every time an output is performed based on this port value. For a faster detection of errors the developer can also specify shorter voting intervals.

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 execution 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.

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 the 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 Zerberus run-time system and is performed in logical zero time.

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.

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 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. This condition is checked by Zerberus at run-time.

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.

5. Case study

For demonstration we have implemented a system to balance a rod under the control of switched solenoids. For a

```
/* 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 2. Functional model

stable control, sample rates in the range of few milliseconds are necessary. The system‘s setup consisted of three computers equipped with AMD K6 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.

While the implementation of a non fault-tolerant application took about two weeks, the conversion of this version for a single machine into a fault-tolerant application could be realized within two hours. The students that implemented the application needed less than 100 lines of code. For the description of the functional model (see figure 2) 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.

In addition four functions had to be implemented by the developers: *read()*, to read the current value from the AD/DA board, *control()*, the real control function realizing a PID controller, *compare()* realizing an interval voting, *write()* for output on the AD/DA board. The code for these functions consisted of less that 70 lines of code.

The addition of the fault-tolerance mechanisms (voting, synchronization, integration), the communication between tasks, sensors and actors, as well as the scheduling is realized by the system. We achieved a sample rate of 1000 Hz for this setup.

This example proves the applicableness for small control applications. However we are currently working on two pilot projects with the industry. The goals of these projects are on the one hand to point out the feasibility, but on the other hand also to adopt industrial standards in Zerberus to increase the acceptance rate in the industry.

6. Conclusions and Further Research

We have introduced in this paper the Zerberus System that helps the developer to accelerate the development process for dependable real-time systems. The main idea of Zerberus is the automatic generation of fault-tolerance mechanisms using templates. This generation can be realized on base of the functional model of the application described in the Zerberus language. Due to its platform independence the system is not restricted to special hardware, a certain programming language or an operating system. Currently there exist two different run-time systems for the operating system VxWorks and the programming languages C and C++.

In the future work we will concentrate on an advanced support of the fault-tolerance mechanisms. We intend to offer a tool for the specification of points during the execution when fault-tolerance mechanisms should be executed (events) and exception handlers to address the occurrence of failures. Thus we want to add new fault-tolerance techniques besides error masking like forward or backward recovery, plausibility and liveliness checks, to enable the user also to design fault-tolerant systems without structural redundancy.

References

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

[2] C. Buckl. Zerberus Language Specification Version 1.0. Technical Report TUM-I0501, TU München, Jan. 2005.

[3] K. Echtle. *Fehlertoleranzverfahren*. Springer Verlag, 1990.

[4] 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.

[5] H. Kopetz. The distributed, fault-tolerant real-time operating system mars. *IEEE Operating Systems Newsletter*, 6(1), 1992.

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

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

[8] 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.

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

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

[11] Website DECOS. http://www.decos.at/.