The Zerberus Language: Describing the Functional Model of Dependable Real-Time Systems

Christian Buckl, Alois Knoll, and Gerhard Schrott

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

Abstract. A growing number of safety-critical systems is controlled by computer systems. Currently these systems are often built from scratch. The Zerberus System assists the developer in the design and implementation process. Main features of the Zerberus System are generality, dependability, real-time predictability, the ability to be certified and cost-efficiency.

The main concept of the Zerberus System is the platform independent specification of the functional model by the developer. The functional model specifies the functional elements (tasks), the relation between these elements, the interaction of the system with the environment and the temporal constraints. On the base of the functional model the Zerberus System automatically generates the fault-tolerance mechanisms. Thus the task of the developer is restricted to the implementation of the application-dependent code.

In this paper we present one major part of the Zerberus System: the Zerberus Language that is used to specify the functional model of the control applications.

1 Introduction

Many safety-critical control systems are automated by the use of computer systems. Although the main fault-tolerance mechanisms are known for a long time [1,2] a general approach in the sense of reusing fault-tolerance mechanisms is missing. Most systems are therefore built from scratch and the application functionality is mixed with the fault-tolerance mechanisms. This leads to a time-consuming and cost-intensive development process.

Within the Zerberus System 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 Zerberus System emphasizes five different features: generality (by supporting the development of computer systems for various applications and domains, e.g. space, medical and traffic engineering), dependability (by providing fault-tolerance mechanisms to comply with the safety and availability requirements), real-time capability (by enabling the satisfaction of hard real-time constraints), the ability to be certified (by meeting

C.A. Maziero et al. (Eds.): LADC 2005, LNCS 3747, pp. 101–120, 2005.
© Springer-Verlag Berlin Heidelberg 2005

certification standards e.g. DO-178B [3], IEC 61508 [4] and assisting the certification process by the system's architecture) and cost-efficiency (by supporting commercial-of-the-shelf (COTS) hardware and by accelerating the development process).

The main concept of the Zerberus System 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. This separation is realized by the specification of the functional model of the application. This model specifies the functional elements, the relation between the elements, the interaction of the system with the environment and the temporal constraints. On the basis of the functional model the Zerberus System 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 automatic code generation of the fault-tolerance mechanisms is performed by using templates that are implemented independent from a certain application. The templates are carefully designed and coded and we intend to obtain a certification for these templates from the German certification authority TÜV. By reusing certified templates for the fault-tolerance mechanisms the development process can be accelerated and the error rates in comparison to a repeated reimplementation of these mechanisms can be reduced.

The fault-tolerance mechanisms that are currently supported are based on structural redundancy. At least three redundant units are executed in parallel. In the following we denote the redundant units as Zerberus units. The system offers facilities for synchronization, voting, exclusion of erroneous units and reintegration of repaired Zerberus units.

In this paper we focus on the Zerberus Language. This language is used for the specification of the functional model by the developer. The language had to be designed in a way that the fault-tolerance mechanisms could be realized based on this model. Therefore the main goals for the language were the suitability for replica determinism (to enable a comparison of the states of the redundant Zerberus units for an error detection) and the existence of previously known points in time for voting (to enable the implementation of distributed voting and synchronization algorithms).

The paper is structured as follows: section. 2 discusses related work, section. 3 introduces the development process proposed by the Zerberus System to clarify the role of the Zerberus Language. In addition the requirements on the language are elaborated. The main concepts of the Zerberus Language are then described in section. 4 in an informal way, while the exact semantics are specified in section. 5. At the end of the paper the concrete syntax of the Zerberus Language is pointed out for a concrete control program in section. 6 and the work is summarized in section. 7.

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 [5]. The time-triggered approach guarantees one important aspect that is absolutely necessary for fault-tolerance mechanisms: determinism.

One important representative for the time-triggered approach is TTP/C [6]. 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 [7]. The TTP/C protocol itself offers nevertheless no built-in fault-tolerance mechanisms at application level. Several other projects addressed this problem (MARS [8] or DECOS [9]). 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 [10,11], 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 [12] or Lustre [13]. 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. 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, which is based on Giotto, to describe the functional model of the safety critical system.

Another tool intended for modeling and implementation of embedded systems is TIMES [14]. Within TIMES the developer models a system and the abstract behavior of the environment. By using a simulator the user can validate the dynamic behavior and verify the schedulability [15] of the system. A code generator for the synthesis of C-code on a LegoOS platform is provided. Like Giotto the tool TIMES was not intended for the use for dependable systems.

Several goals of the Zerberus System are also shared with Erlang [16,17]. 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 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.

3 Development Process

The Zerberus System suggests different 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.

The description of the individual steps is focused on the requirements towards the Zerberus Language.

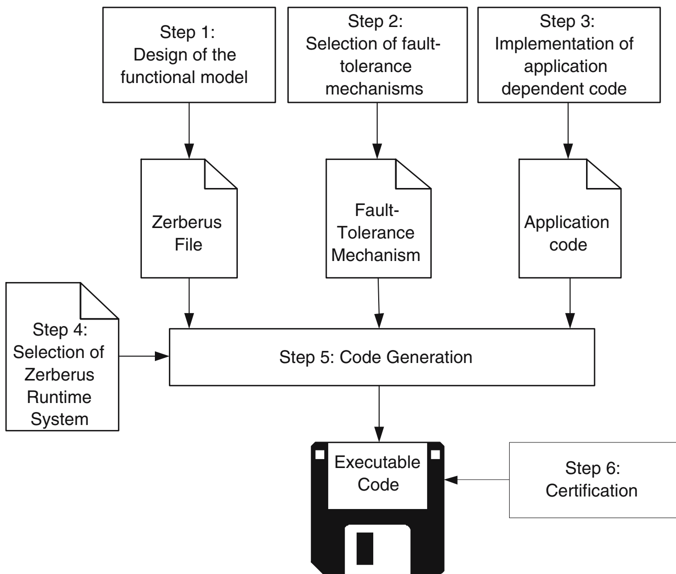


Fig. 1. Development process

3.1 Specification of the Functional Model

Within this step the user has to specify 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. Since the specification of the functional model should be independent of a specific platform, the Zerberus Language has to be designed in a way to support this independency. A platform in the context of the Zerberus System is understood as the hardware, the operating system and the programming language.

The Zerberus Language was designed very simple and intuitive to avoid an error source and a long-lasting learning process. The language is not based on a certain programming language or operating system to comply with the generality requirement of the Zerberus approach.

3.2 Analysis of the Requirements on the Dependability

Currently the Zerberus System 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 in most times application dependent the current run-time systems offer only a restart of the system or a reboot. 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 allows no toleration of design errors, the system also supports diversity of hardware and software. While hardware diversity leads to no or only few additional costs as a result support of COTS hardware, N-Version programming is often not considered due to the extra effort necessary for the implementation of the individual versions.

As a result of these considerations several requirements are posed to the Zerberus Language. First of all the language must support the replica determinism: during the system execution it must be possible to compare the states of the individual Zerberus units for error detection. Especially due to the support of N-Version programming this is not a trivial requirement.

Another requirement that arises due to the voting is the existence of deterministic points in time when the voting should be performed. The existence of deterministic points in time is on the one hand the main requirement to allow the implementation of a distributed voting algorithm, on the other hand it also allows the implementation of a distributed clock synchronization algorithm.

The voting in the Zerberus system 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 [18]. The voting messages are also used for the synchronization algorithm [19,20,21]: by means of the expected and the actual arrival time of the voting messages a logical global clock can be

computed. The initial clock synchronization at start up is based on the algorithm implemented in the TTP/C [6] protocol.

To support the re-integration of a previously excluded Zerberus unit, the system must offer facilities for state synchronization. Since the algorithms are realized automatically by the system a derivation of the state of the individual units must be possible out of the functional model.

Finally, in order to achieve a reduction of the implementation effort for N-Version programming the code that has to be implemented by the developer should be restricted to the pure application dependent code.

3.3 Implementation of Application Dependent Code

In this step the developer has to implement 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 formal 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.

3.4 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 the Zerberus System, 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. To avoid a repeated implementation of such run-time systems, Zerberus offers a way to code such run-time systems application independent. By the use of other means, the Zerberus tags, locations in the code that have to be replaced with application dependent data can be marked. The replacement takes place in the code generating process.

To enable the simultaneous use of run-time systems implemented by the developer and of run-time systems provided by Zerberus in a N-Version programming system all protocols for the fault-tolerance mechanisms are provided. Thus the design effort for a new run-time system is also minimized.

3.5 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. The whole code generation process is depicted in fig.2. Both the functional model and the run-time systems are parsed by the code generator and syntactic and semantic checks are performed. Afterwards the code generator replaces the Zerberus tags by application data and produces executable code.

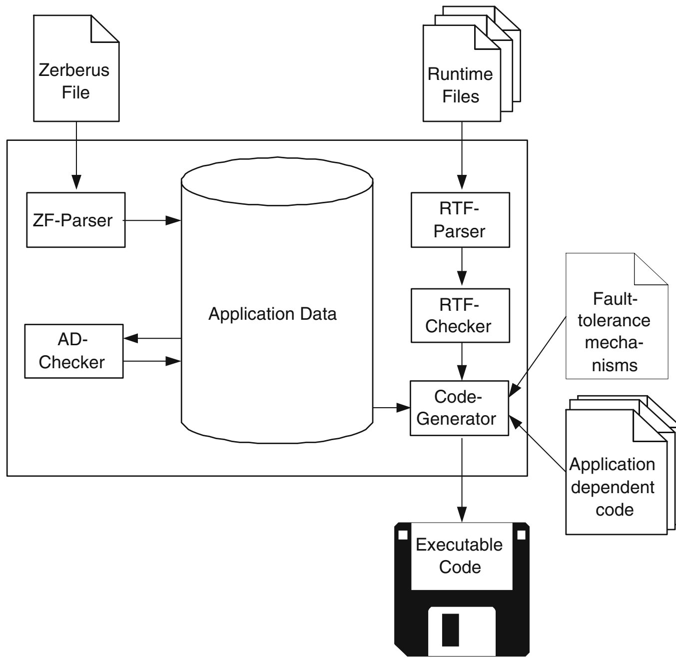


Fig. 2. Code generation process

3.6 Certification of the Zerberus System

The certification of an application developed with the Zerberus System can be split up into three distinct parts:

1. Certification of the Zerberus System approach
2. Certification of the Zerberus run-time system
3. Certification of the application-dependent code

Certification of the Zerberus System Approach. In a first step the Zerberus System approach has to be certified. This certification includes the Zerberus Systems concept (including voting, synchronization, integration algorithms), the Zerberus Language, the code generator and the Zerberus Tags. All tools are currently available as prototypes. For a successful certification these tools have to be re-engineered according to the standards proposed by the certification authorities (RTCA,FDA,TÜV).

Certification of the Zerberus Run-Time Systems. In the second step the certification of the run-time systems is performed. This includes tests of the successful implementation of the proposed algorithms, the successful execution of functional models and the conformance with the proposed standards of the certification authorities. Currently two prototype implementation for VxWorks 5.5 and the programming languages C and C++ are available.

Certification of the Application Dependent Code. For the certification of an application developed with the Zerberus System only a certification of the functional model, the code implemented by the user and the compliance with the Zerberus run-time system should be necessary. To achieve this minimization a strong partitioning among the different integrated modules must be ensured. This separation is another requirement towards the Zerberus Language.

For a successful certification the system must of course apply to the certification standards. These standards differ from the fields of application [22]. 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 steps one and two. We intend to achieve such a certification by the German certification authority TÜV for the medical domain.

4 Informal Description of the Zerberus Language

In the previous section the requirements on the Zerberus Language were discussed in the context of the different development process steps. In this section the Zerberus Language is described informally and it is shown that the requirements can be satisfied by the Zerberus Language. The language was influenced by the language Giotto introduced in Berkeley [10]. Giotto was changed and extended in a way that the resulting Zerberus Language was suited for the use for fault-tolerant applications.

The main attribute 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 [5]. Similar to the approach in [23] replica determinism can be achieved by using the knowledge about the execution times. In the context of control applications the execution times can be related to the frequency of control cycles.

Basing the voting, synchronization and integration algorithms on the frequency of control cycles has different positive outcomes: by specified frequencies of control cycles in the functional model there exist on the one hand deterministic points in time, when the synchronization and voting algorithms can take place. On the other hand the execution and scheduling of the different processes can be carried out in different ways on the Zerberus units between these points.

The existence of deterministic points in time allow the application of distributed voting and synchronization algorithms. In this way a single point of failure can be avoided.

To achieve the claimed simplicity of the language, the Zerberus Language consists of only seven different object types: ports, actors, sensors, guards, modes and modechanges. In this section the different object types are explained informally.

4.1 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. Port types are the only element of the Zerberus Language, that refer directly to a specific platform. To guarantee the platform independence the port types are platform independent, but are based on the fundamental types of the most common programming languages.

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 has to be performed deterministically: it is not allowed that more than one write access is performed at a certain point in time. This condition is checked by the code generator while parsing the functional model and in addition at run-time (necessary due to the possible usage of guards, see section.4.6).

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) or of N-Version programming. 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.

4.2 Task

The separation of the pure functionality of the application and the run-time system including the fault-tolerance mechanisms is realized by tasks. Tasks are periodically called functions and realize the actual control system functionality. The simultaneous execution of different tasks is allowed, but to achieve determinism in the execution the tasks have to be independent of each other and synchronization points are not allowed. Thus the implementation of the task functions is simplified and accelerated since they represent only sequential programs and the requirement of the strict partitioning of the integrated modules to reduce the certification effort is satisfied.

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.

4.3 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.

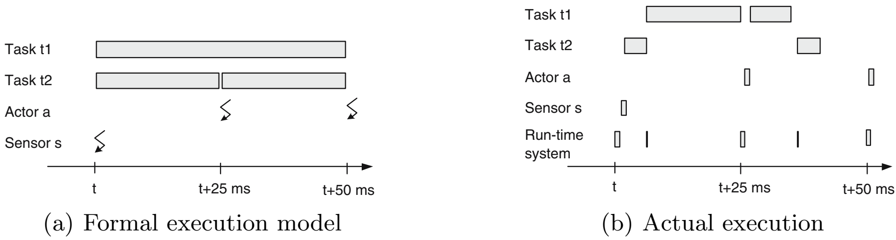
The execution of the sensor and actor functions is also performed time-triggered. The execution frequency has to be specified by the developer. The sensor execution takes thereby place at the begin of each interval, the actor execution at the end of each interval. Both executions are regarded as instantaneous. 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 functions 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.

4.4 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.

```
mode m
{
        task= t1 1,t2 2;
        actor= a 2;
        sensor= s 1;
        duration= 50000000 ns;
}
```

Fig. 3. Mode declaration



(a) Formal execution model (b) Actual execution

Fig. 4. Execution model for mode m

Figure 3 shows the declaration of an example mode m in the Zerberus Language. m contains two tasks t1 with frequency 1 and t2 with frequency 2, a sensor s with frequency 1 and an actor with frequency 2. The duration of one mode cycle is set to 50 ms.

The formal execution model is depicted in figure 4(a) under the assumption that the mode cycle starts at time t. At time t the function of sensor s is executed and the tasks t1 and t2 are started. At time t+25ms the task t2 is stopped and the actor function is executed. Afterwards the task t2 is started for a second time. At the end of the mode cycle at t+50ms both tasks are stopped and the actor a is executed a second time. The execution of the sensor and actor functions appear instantaneous in the execution model.

Figure 4(b) shows a possible actual execution of the mode cycle on the machine. In addition to the task execution also the time required for the actor and sensor function execution, as well as the time consumed for run-time system execution have to be considered. The run-time system realizes the scheduling of the tasks, the port accesses and the voting and synchronization with the other Zerberus units.

The scheduler used in the example of fig. 4(b) uses a Earliest-Deadline-First strategy for the task execution. Sensors and actors are executed within the run-time system context.

4.5 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 always place 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 in the Zerberus System at run-time.

4.6 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. Thereto the guard is assigned to a certain tasks. 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 Formal Description of the Zerberus Language

The concrete language specification is given in [24]. In this chapter we describe the language in a more abstract way. A Zerberus program computes on the base of some inputs by the environment the output to the environment. In the following we refer to *Input* for the values of the environment inputs and *Output* for the values of the output to the environment.

A Zerberus program consists of:

1. A set of port declarations: A port declaration $(p, type, init, comp)$ consists of a port name p, a type *type*, an initial value $init \in type$ and a compare mode *comp*. The set of allowed types are the basic types of common programming languages (abstracted to achieve platform independence) and arrays of fixed size of these types. Every port declaration must also contain an initial value to achieve a common start configuration for all units.

   The developer can specify how a port is treated within the voting algorithm. These possibilities range from the denial of comparisons, a bit-by-bit comparison to an user-defined comparison (typically an interval test). The denial of comparisons is only valid if the port is not read by an actor.

   Port names must be uniquely declared: that means if $(p, *, *, *)$ and $(p', *, *, *)$ are distinct port declarations, then $p \neq p'$.

   We refer to the set of declared ports by *Ports*, to the initial value of a port p by $init[p]$ and to the values of a set of ports $P \subseteq Ports$ by $Vals[P]$.
2. A set of actor declarations: An actor declaration (a, f, P) consists of an actor name a, an actor function name f and a set $P \subseteq Ports$ of input ports. Actor names must be uniquely declared: that means if $(a, *, *)$ and $(a', *, *, *)$ are distinct port declarations, then $a \neq a'$.

   The developer has to implement an actor function with the name f for each platform used. The function must be of the form $f : Vals[P] \rightarrow Output$ and is executed every time the actor is invoked synchronously within the system's context. We write *Actors* for the set of declared actors and f_a for the function of an actor a.
3. A set of sensor declarations: A sensor declaration (s, f, P) consists of a sensor name s, a sensor function name f and a set $P \subseteq Ports$ of output ports. Sensor names must be uniquely declared.

The developer has to implement a sensor function with the name f for each platform used. This function must be of the form $f : Input \rightarrow Vals[P]$. The sensor function is executed every time the sensor is invoked synchronously within the system's context. We refer to $Sensors$ for the set of declared sensors, to f_s for the function of a sensor s and to $res_s[p]$ for the results regarding port $p \in P$ of the sensor function.

4. A set of guard declarations: A guard declaration (g, f, P) consists of a guard name g, a guard function name f and a set $P \subseteq Ports$ of evaluation ports. Guard names must be uniquely declared.

   The developer has to implement a guard function with the name f for each platform used. This function must be of the form $f : Vals[P] \rightarrow \mathbb{B}$. Guard functions are invoked every time the assigned task should be started. The execution of the guard function takes place synchronously within the systems context. We write $Guards$ for the set of declared guards, f_g for the function of a guard g, $p[g]$ for P and $res_g(Vals[P])$ for the results of one function invocation based on the current values of the assigned ports.

5. A set of task declarations: A task declaration $(t, f, g, In, Out, Inout)$ consists of the task name t, the task function name f, optionally a guard $g \in Guards$ and a set of Ports $In \cup Out \cup Inout \subseteq Ports$. Task names must be uniquely declared.

   The set of assigned ports is subdivided into three classes: In, Out and $Inout$. These classes refer to the access type of the task to the port. Every port used in the task must belong to exactly one class.

   The developer has to implement the task function with the name f for each platform used. The function must be of the form $f : Vals[In \cup Inout] \rightarrow Vals[Out \cup Inout]$ and is performed every time the task is invoked by the system. The execution takes place asynchronously to the system's context.

   We write $Tasks$ for the set of declared tasks, $res_t[p]$ for the results of the current function invocation of task t concerning one assigned output port $p \in In \cup Out$ and f_t for the function of a task t.

6. A set of mode declarations: A mode declaration $(m, start, T, A, S, d)$ consists of a mode name m, a boolean value $start$, task assignments T, actor assignments A, sensor assignments S and a duration d. Mode names must be uniquely declared.

   Within the application exactly one mode must be declared as start mode m_{start}, that means $start = true$. The system will start the operation in this mode.

   A task assignments $(t, freq)$ consists of a task $t \in Tasks$ and a related frequency $freq \in \mathbb{N}$. The frequency determines the number of the task invocations within one mode cycle (except if a related guard evaluates false). In the following we will refer to the frequency $freq$ of a task t in mode m by $freq(t, m)$. The sensor and actor assignments are similar.

   The duration (s, ns) consists of the number of seconds $s \in \mathbb{N}$ and the number of nanoseconds $ns \in \mathbb{N}$ (to confirm with the POSIX standard) and determines the duration of one mode cycle.

   We write $Modes$ for the set of declared modes.

7. A set of modechange declarations: A modechange declaration $(mc, f, P, Source, target)$ consists of the modechange name mc, a modechange function name f, a set $P \subseteq Ports$ of evaluation ports, a set of source modes $Source \subseteq Modes$ and a target mode $target \in Modes$. Modechange names must be uniquely declared.

   The developer has to implement a modechange function with the name f for each platform used. The function must be of the form $f : Vals[P] \rightarrow \mathbb{B}$. A modechange is evaluated always at the end of a mode $m \in Source$. If the function result is true the new mode executed by the system will be $target$. We write $Modechanges$ for the set of declared modechanges, f_{mc} for the function of a modechange mc, $p[mc]$ for P and $res_{mc}(Vals[P])$ for the results of one function invocation.

In the following the semantics of the Zerberus Language are described. The realization of the fault-tolerance mechanisms is mentioned but the focus lies on the functional semantics.

The voting algorithm has three results: the state of the system $res_{sys} \in \mathbb{B}$, the state of the own unit $res_{unit} \in \mathbb{B}$ and the acting unit id $act \in \mathbb{N}$. The result of the synchronization is the temporal correction value Δ_{cor}. In addition we assume that the developer has decided to use the port values for voting only in case they are used for an actor output.

For simplicity reasons possible occurrences of errors during the application execution are ignored. These errors can be time violations or simultaneous write attempts on one port. In all such cases the normal execution is aborted at once and fault reaction algorithms are executed.

A program configuration $C = (id, s_sys, s_unit, m, \delta, v, \sigma_{active,}, \tau)$ consists of the unique Zerberus unit ID id, states of the system s_sys and of the own unit s_unit, a current mode $m \in Modes$, a mode unit $\delta \in \mathbb{N}$, a valuation $v \in Vals[Ports]$ for all ports, a set of active tasks $\sigma_{active} \subseteq Tasks$ and a time stamp $\tau \in \mathbb{Q}$. The set of active tasks σ_{active} contains all tasks that are logically running, whether or not they are physically running by expending CPU time. The mode unit δ represents the current internal point of the mode cycle. The number of internal points within one mode cycle of mode m is determined by the least common multiple $\omega[m]$ of the frequencies of the tasks, actors and sensors assigned to m.

At start-up each Zerberus unit has to determine if the system is currently running or if an initial synchronization procedure must be started. This is realized by a function of the run-time system that observes the network. An operating system can be recognized by voting and synchronization messages.

In case the system is already running another run-time system function is executed that allows to obtain the states of the other Zerberus units. One requirement for a state synchronization is that the system is currently at the beginning of one mode cycle ($\delta = 0$). In this case no tasks are active on the other units and an integration can be successful. Another requirement is that the majority of Zerberus units agrees in their states. If both requirements are met the configuration is updated to the state of the majority and the integration was successful.

If on the other hand the system is not running an initial synchronization procedure is started. The goal of this procedure is to obtain a global time base. In case of a successful synchronization the initial configuration is set to $C_{init} = (id, true, true, m_{start}, 0, v_{init}, \emptyset, \tau_i)$ where τ_i is the result of the initial synchronization and $v_{init}[p] = init[p]$.

The internal points represent the points in time, when the synchronization and voting algorithms are executed. At each internal point the following steps are performed by the run-time system on the basis of the current configuration $C = (id, s_sys, s_unit, m, \delta, v, \sigma_{active,}, \tau)$:

1. Copying of task results: Let $\sigma_{completed}$ be the set of tasks $t \in Tasks$ that are completed. A task t is completed if $t \in \sigma_{active}$ and if δ is an integer multiple of $\omega[m]/freq(t, m)$ at configuration C. For all ports $p \in Ports$: if $p \in inout[t] \cup out[t]$ of a task $t \in \sigma_{completed}$ then define $v_{stop}[p] = res_t[p]$, else $v_{stop}[p] = v[p]$. Let C_{stop} be the new configuration that agrees with v_{stop} in the values of ports and with the set of active tasks $\sigma_{stop} = \sigma_{active} \backslash \sigma_{completed}$ and otherwise agrees with C.
2. Voting and synchronization: Let $a_{execute}$ be the set of actors to be executed. An actor a is executed if δ is an integer multiple of $\omega[m]/freq(a, m)$ at configuration C_{stop}. Let p_{vote} be the set of all ports read by the actors $a \in a_{execute}$. The voting and synchronization algorithms are then invoked with the parameters $v_{stop}[p_{vote}]$, the mode m and the mode unit δ. Let res_{system}, res_{unit} and act be the results of the voting algorithms and Δ_{cor} be the result of the synchronization algorithm. If $((res_{system} \wedge res_{unit}) = false) \vee |(\Delta_{cor}| > \epsilon)$ then the normal system execution is aborted and error reaction and recovery algorithms are invoked. Otherwise let C_{vote} be the new configuration that agrees with $s_sys_{vote} = res_{system}$, $s_unit_{vote} = res_{unit}$, $act_{vote} = res_{act}$ and $\tau_{cor} = \tau + \Delta_{cor}$ and otherwise agrees with C_{stop}.
3. Execution of actors: Let $a_{execute}$ be the set of actors to be executed. For all actors $a \in a_{execute}$ the actor function f_a is executed if $id = act_{vote}$. If $id \neq act_{vote}$ the unit only controls the correct output (performed by another unit). In case errors are detected by the system error recovery algorithms are executed. The execution of the actor functions takes places synchronously within the run-time system execution that means that the run-time system waits for the completion of the actor function. Let C_{actor} be the new configuration that agrees with C_{vote}.
4. Evaluation of modechanges: If $\delta = 0$ modechanges have to be evaluated. The set of modechanges mc_{eval}that needs to be considered consists of all modechanges mc with $m \in source(mc)$. For each modechange $mc \in mc_{eval}$ the corresponding function is evaluated and if $f_{mc}(v_{stop}[p[mc]]) = true$ then $m' = target(mc)$. The developer has to guarantee that at most one modechange evaluates true at a time. The run-time systems checks this condition and creates an internal error in case of a violation of this rule. In the latter case the system execution is stopped and fault reactions are started.

If no modechange evaluates true, then $m' = m$. Let $C_{modechange}$ be the new configuration that agrees in m' as new operating mode and otherwise with C_{actor}.

5. Execution of sensors: Let $s_{execute}$ be the set of sensors s to be executed. A sensor is executed if δ is an integer multiple of $\omega[m']/freq[s, m']$. Let p_{sensor} be the set of ports that are written by a sensor $s \in s_{execute}$. For each port $p \in Ports$: if $p \in p_{sensor}$ $v_{sensor}[p] = res_s[p]$, else $v_{sensor}[p] = v_{stop}[p]$. Let C_{sensor} be the new configuration that agrees with v_{sensor} in the values of the ports and otherwise with $C_{modechange}$.
6. Invocation of tasks: Let t_{start} be the set of tasks t to be started. A task t is started if δ is an integer multiple of $\omega[m]/freq[t, m']$. In addition if the task has a guard the evaluation must be positive: $res_g(v_{sensor}[p[g]]) = true$. For every task $t \in t_{start}$ the function f_t is invoked with the specified parameters based on the values v_{sensor}. Let C_{start} be the new configuration that agrees with the set of active tasks $\sigma_{start} = \sigma_{stop} \cup t_{start}$ and otherwise with C_{sensor}.
7. Advance time: If $\delta = \omega[m] - 1$ then $\delta' = 0$ otherwise $\delta' = \delta + 1$. Let $\tau' = \tau_{cor} + d[m']/\omega[m]$. The next time the program is invoked with step 1 is at time τ'. Let C_{succ} be the new configuration that agrees with δ' and τ' and otherwise with C_{start}.

6 Case Study

For demonstration we have implemented a system to balance a rod under the control of switched solenoids, see figure 5. For a stable control sample rates in the range of few milliseconds are necessary. As device an AD/DA-board was used to connect the experimental setup with the three computer units. The computers were equipped with AMD Athlon processors and they were connected by switched ethernet. As real-time operating system we used VxWorks and as programming language C.

Fig. 5. Balanced rod

```
/* 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;
        compareTIME=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;
}
```

Fig. 6. Functional model

The implementation of the control program was done by two students. It took two weeks to implement the PID controller on a single computer. The conversion of the code to the Zerberus System and the addition of the fault-tolerance mechanisms could be realized within two hours using the code for the single-machine version. The code that had to be implemented for the fault-tolerant controller was less than 100 lines of code.

For describing the functional model of the control application 30 lines of code in the Zerberus Language were needed. The code is depicted in figure 6. Three ports had to be declared: one port for the systems input (the deviation of the current position from the desired position), an array of two integer values for the differential and integral part and one port for the result. Only the port for the result was used for the voting algorithm. Also the rest of the functional model was very simple: a sensor was used to read the current position of the rod, a

task was needed for the control computation and an actor was used for writing the output to the environment.

In addition to the functional model four functions were needed for the control program:

- *read()*: The sensor function was used to read the current value from the AD/DA-board.
- *control()*: This function implemented the PID controller. As input the function uses the current position of the rod. The function computes the necessary control output for stabilizing the rod at the desired position. To achieve this goal the function uses two further ports to obtain also the differential part and the integral part of the controller.
- *compare()*: The function *compare()* is used within the execution of the voting algorithm of the run-time system. Due to synchronization differences and to sensor imprecision a binary compare of the result of the *control()* function was not possible. Therefore the two students implemented an interval decisions: two results were assumed to be correct if the difference between both values was less than 0.1 V (allowed voltage range was -10..10 V).
- *write()*: The actor function was used to write the value of the port output to 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 was realized by the system. The sample rate for this control example was 1000 Hz.

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 the Zerberus System to increase the acceptance rate in the industry.

7 Conclusions and Future Research

In this paper we have introduced the Zerberus Language. This language enables the developer to design the functional model of the control application. The design of the language was guided by the different requirements on the language and the development system.

To achieve a general applicability the constraints by the language should be minimized. This was realized by the independency from a certain platform and by the time-triggered approach which is suitable for most control systems.

For the use with fault-tolerance mechanisms and especially with active redundancy the language must provide features for replica determinism. By the time-triggered approach this requirement is satisfied. In addition deterministic points in time for the execution of voting algorithms are available and also a synchronization of the different units can be achieved. The state synchronization

during the reintegration phase is enabled by separating the inner state (ports) from the functionality (tasks).

One main aspect of supporting the acceleration of the certification process is the strict separation of the different integrated modules. This separation is realized by the task concept of the Zerberus Language. In case operating systems are used that support memory protection, it can be guaranteed that the run-time system is not influenced by the tasks except in the predefined way.

The Zerberus Language is therefore suited for the use within the Zerberus System. A code generator is available to support the transformation of the functional models designed in the Zerberus Language into executable code. Within one small case study we demonstrated the usage of the Zerberus System.

To point out the applicableness within industrial projects we are currently working on two pilot projects with the industry. Within these projects we also plan to adopt the recommended development process and the tools to industrial standards. In addition we want to support further fault-tolerance mechanisms despite active structural redundancy. Therefore we intend to introduce another language to specify points within the execution when fault-tolerance mechanisms should be executed (events) and exception handlers to address the occurrence of failures. The goal is to provide a set of standard fault-tolerance mechanisms to the user. To assist the developer in choosing adequate mechanisms, guidelines will be developed.

Another research area will be an advanced support of the user in the certification process. Document output automated by the used tools and the compliance of tools and run-time systems with the relevant development standards are planned. Within one project for a medical control system in cooperation with the German certification authority TÜV we want to exemplify our approach.

References

1. Pradhan, D.K.: Fault-Tolerant Computer System Design. Prentice Hall (1996)
2. Lee, P.A., Anderson, T.: Fault Tolerance: Principles and Practice. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1990)
3. RTCA DO-178B: Software considerations in airborne systems and equipment certification (1992)
4. International Electrotechnical Commission: IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems. (1998)
5. Kopetz, H., Bauer, G.: The Time-Triggered Architecture. Proceedings of the IEEE **91** (2003) 112 – 126
6. TTTech Computertechnik AG: Time Triggered Protocol TTP/C High-Level Specification Document. (2003)
7. 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
8. Kopetz, H., Fohler, G., Grünsteidl, G., Kantz, H., Pospischil, G., Puschner, P., Reisinger, J., Schlatterbeck, R., Schütz, W., Vrchoticky, A., Zainlinger, R.: The distributed, fault-tolerant real-time operating system mars. IEEE Operating Systems Newsletter **6** (1992)

9. Website DECOS: (http://www.decos.at/)
10. 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
11. Henzinger, T.A., Horowitz, B., Kirsch, C.M.: Embedded control systems development with giotto. Proceedings of the International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES) (2001) 64 – 72
12. Berry, G., Gonthier, G.: The esterel synchronous programming language: Design, semantics, implementation. Science of Computer Programming **19** (1992) 87–152
13. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.A.: Lustre: a declarative language for real-time programming. In: POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, ACM Press (1987) 178–188
14. Amnell, T., Fersman, E., Mokrushin, L., Pettersson, P., Yi, W.: Times - A Tool for Modelling and Implementation of Embedded Systems. In: Joint European Conferences on Theory and Practice of Software, ETAPS 2002. Lecture Notes in Computer Science, Springer-Verlag (2002)
15. Krcal, P., Yi, W.: Decidable and Undecidable Problems in Schedulability Analysis Using Timed Automata. In: Joint European Conferences on Theory and Practice of Software, ETAPS 2004. Lecture Notes in Computer Science, Springer-Verlag (2004)
16. 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
17. 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
18. Echtle, K.: Fehlertoleranzverfahren. Springer Verlag (1990)
19. Lamport, L., Melliar-Smith, P.M.: Synchronizing clocks in the presence of faults. J. ACM **32** (1985) 52–78
20. Lundelius, J., Lynch, N.A.: A new fault-tolerant algorithm for clock synchronization. In: Symposium on Principles of Distributed Computing. (1984) 75–88
21. Schmid, U., Schossmaier, K.: Interval-based clock synchronization. Real-Time Systems **12** (1997) 173–228
22. Saglietti, F.: Licensing reliable embedded software for safety-critical applications. Real-Time Systems **28** (2004) 217–236
23. 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
24. Buckl, C.: Zerberus Language Specification Version 1.0. Technical Report TUM-I0501, TU München (2005)