The XBW Model for Dependable Real-Time Systems

Vilgot Claesson
Chalmers University of Technology
vilgotc@ce.chalmers.se

Stefan Poledna
TTTech
poledna@tttech.com

Jan Söderberg
Mecel AB
jan.soderberg@mecel.se

Abstract

This paper presents a new conceptual model, the XBW-Model. Distributed computing is becoming a cost effective way to implement safety critical control systems. To support the development of such systems the XBW conceptual model was developed. The model describes the time behavior and distribution properties of a system in such a way that static scheduling and systematic fault tolerance can be applied. The conceptual model also enables the definition of an appropriate fault model. This fault model along with the XBW-model allow efficient and systematic use of well known software based error detection methods. A distributed steer-by-wire control system is described, which is developed according to the model. The XBW-Model is developed within in the European Brite-EuRam III project X-By-Wire.

1. Introduction

Computer control in hard real-time systems is becoming common in safety critical systems. At the same time the use of distributed architectures is increasing. This introduces new requirements on software design regarding areas such as timing predictability and fault-tolerance. Timing is no longer a node internal issue since each node has to synchronize its actions with other nodes. Fault tolerance is of course a major concern in safety critical systems. Mechanisms for fault tolerance has to be considered in an early stage of the design and needs to be handled in a structured and systematic way throughout the development process. To handle the inherent complexity of developing a safety critical system, systematic fault tolerance is desirable. Other important properties of such systems are testability and reusability which both can be addressed in a systematic way. New application areas where the production cost is crucial, like the automotive, favor solutions that can be implemented with a minimum of hardware redundancy, opting for software based solutions. Software based fault tolerance also offers a reliability gain while using a minimum of hardware resources.

To support the development of safety critical hard real-time systems in a distributed environment a new conceptual model for software design, called the XBW-model, has been developed. This model has been developed with the experience from the partners in the X-By-Wire project. The XBW-Model is designed to support a time-triggered architecture and to take advantage of the inherent predictability following such a system. The intention with this model is to obtain high reusability, testability and maintainability without losing in efficiency and reliability. The model is influenced by the Basement [5] and DFR [16] concepts.

The basic idea of software development with the XBW model is to connect constructional elements, called BPEs (Basic Processing Elements), to establish the functions of the application. The interface of a BPE is input and output messages with state semantics that are used to communicate with other BPEs, where the communication is restricted to the initial and final parts of the BPE. The simple and well-defined interface increases the reusability and testability of the BPE. To assist structuring of functions and subsystems, the BPEs can be composed into CPEs (Complex Process Elements). To view the timing constraints on the functions and precedence between the BPEs a PED (Processing Element Directed acyclic graph) is used. The PED has exactly one trigger condition, which is normally a time-trigger, (the XBW-Model also allows event-trigger conditions), and a deadline associated with it. All output messages from the BPEs in the PED have to be sent within this deadline.

This conceptual model is used to derive a fault model where the constructional elements (BPEs) defines the fault abstraction level and region of fault containment. Based on this fault model well known software based error detection methods are proposed. These methods will be implemented as systematic fault tolerance mechanisms, supported by system services such as an operating system. This will simplify managing fault tolerance during system design. Including these mechanisms when specifying the system in the XBW-model will give a lucid overview of the fault tolerance properties of the system. It may also be

possible to include application specific fault-tolerance in a systematic way by using an assertion framework.

Chapter 2 describes the XBW Conceptual model and its different building blocks. In chapter 3 the fault model is defined and in chapter 4 the strategy for systematic fault tolerance which is used in conjunction with the model is described. As a small sample implementation using the XBW-model a Steer-by-wire function is presented in chapter 5. Finally, chapter 6 gives a summary of the paper.

2. Conceptual model

The goal with this concept is to obtain a model that is analyzable and describes the real-time behavior and also to facilitate systematic fault tolerance. The model will also aid in designing system parts that are highly reusable, testable and maintainable, without losing in efficiency and reliability. The model must also describe the modularization and decomposition of the system. Especially the design of time triggered systems has been considered during the development of this model.

It should also facilitate systematic fault tolerance such that systematic fault tolerant mechanisms and runtime schedules can be automatically generated. The timing properties can then be verified using the information from the model. Similar functionality is currently being implemented in off-line tools for the ERCOS OS [13].

The model has two major constructional elements, the Basic Processing Element and the Complex Processing Element.

2.1 Basic Processing Element

The Basic Processing Element (BPE) is the smallest composable software structure element and communicates with other BPE via messages (Figure 1). The purpose of this element is to serve as a common containment region for processing, distribution, scheduling, and error detection. The main reason for using the BPE for all these purposes is the conceptual simplicity, that a single interface can be used for the error detection and recovery mechanisms.

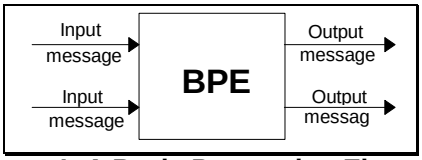

Figure 1. A Basic Processing Element.

Messages sent or received by a BPE have state semantics [19, 15], i.e. all messages are non-destructive read and destructive write.

The internal state between executions of a BPE will be saved internally and is referred to as history-state (H-state). The H-state makes it easier to systematically keep track of information that might be necessary to exchange between redundant instances of the BPE, e.g. for backward recovery actions.

All received messages together with the H-state builds what is referred to as the input vector, and all output messages together with the new history-state builds the output vector, see Figure 2.

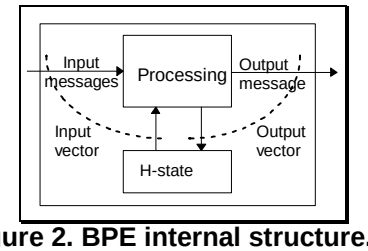

Figure 2. BPE internal structure.

The input vector must be complete before executing the BPE. When processing of the BPE is finished all output messages will be sent and the new history-state saved, i.e. all input and output of the processing are performed prior and after the processing respectively. This internal time precedence of the BPE is described in the figure below.

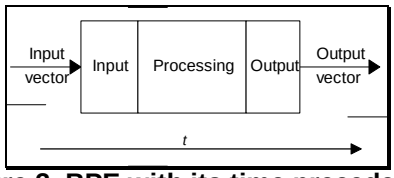

Figure 3. BPE with its time precedence.

The processing is performed non-blocking, i.e. all the conditions for the processing are satisfied after the input, and so describes the smallest unit of scheduling. (This implies no assumption about preemption).

This simple and well-defined interface will increase the reusability and testability of the BPE.

One optional attribute of the BPE is reference input and output vectors which are used for error detection purposes (section 4.1.2).

2.2 Complex Processing Elements

To assist construction and structuring of functions and subsystems the Complex Processing Element (CPE) is introduced. The CPE will also make it possible to view the system in different hierarchy levels.

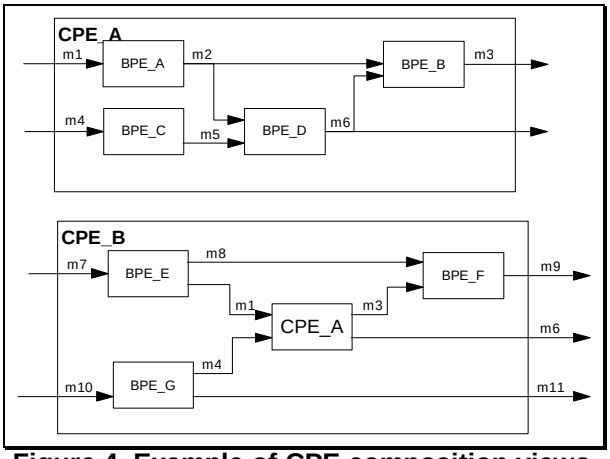

Figure 4. Example of CPE composition views.

BPEs and CPEs, which are functionally or in other ways related to each other, can be composed together in a CPE, see Figure 4.

The message information flow can be one-to-one, (1:1) like message m_5 in Figure 4 or one-to-many, (1:m) like message m_2. Message information flows of the forms many to one (n:1) and many-to-many (n:m) are not supported.

The CPE can be described at hierarchical levels without revealing its internal structure of BPEs and CPEs (this supports the concept of information hiding and abstraction).

2.3 Processing Element Directed a-cyclic Graph

In a PED (Processing Element Directed a-cyclic Graph), which is an extension to the CPE, we can express the precedence relation between different elements in the graph. If there is multiple precedence relations both conditions have to be fulfilled (AND semantics). The precedence relation is indicated with dashed arrows as a distinction to the messages' continuous lined arrows.

Different execution models like periodic or sporadic tasks are modeled by using trigger conditions for the activation of the PED.

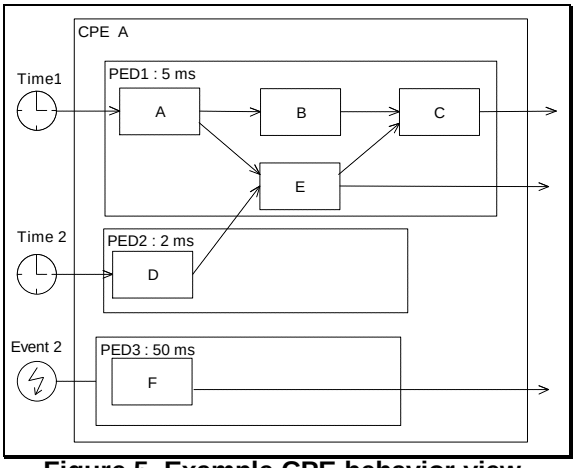

Figure 5. Example CPE behavior view.

The PED is determined by the following characteristics:

- The PED has exactly one trigger element that describes the activation criteria for the PED.
- Each BPE/CPE in the PED can be reached by the precedence edges from the trigger element.
- The PED is associated with a deadline 'D' that is defined relative to the trigger element and the last output message of the PED.
- The PED is further associated with a computation time 'C'. This in turn is based on the computation time of the included BPEs.
- Separate PEDs can be coupled by precedence edges.
- There are no mutual precedence relations between PEDs.
- The trigger element can be a time or an event. In the figure above PED1 and PED2 are time triggered and PED3 is event driven.

Time triggers which are used for periodic activation have the following attributes:

- a time period 'T' and
- a time offset 'r' within the period.

Event triggers which are used for sporadic activation have the following attributes:

- a relation to an external (physical) or internal (symbolic) event including an optional delay r and
- a minimum interarrival interval 'Tmin'.

2.4 Physical distribution

The physical distribution of a system is described by physical node attributes to the BPEs, CPEs or PEDs, which must not be contradictory. The distribution could also be described in a separate graphical view, but this is currently not included in the model.

2.5 Mutual exclusion

The behavior model of the BPE does not include mutual exclusion between BPEs, e.g. to achieve mutually exclusive access to shared resources. A way to include mutual exclusion while conserving the deterministic behavior of the model is to use the *Resource* mechanism [1]. The major advantage of the resource mechanism is that it completely avoids blocking, deadlocks, and livelocks. This allows the analysis of the worst case execution time. The blocking avoidance makes the resource mechanism comply with the input/process/output pattern of the BPE.

Some of the proposed error detection mechanisms in chapter 4.1 excludes the use of resources since it cannot be guaranteed that double update operations are idempotent. This restriction can be relaxed if the implementation guarantees that the double access to the resource leads to no inconsistency.

3. Fault model

The terminology *fault*, *error*, and *failure* used in this paper is based on the definitions given by Laprie [11]: Accordingly a fault is the adjudged or hypothesized cause of an error. An error is the part of the system state which may cause a failure. Errors are manifestations of a fault in the system. A failure is the deviation of the delivered service from compliance with the service specification. Failures are triggered by errors. They in turn may cause faults of systems interacting with the failed systems or at higher hierarchy levels.

The fault model is derived from the conceptual model where the constructional elements define the fault abstraction level and the regions of fault containment. The different fault types are: (1) Data Processing Fault, (2) Data Storage Fault, (3) Data Flow Fault, (4) Control Timing Fault and (5) Control Flow Fault.

Data Processing Faults pertain to the processing part of the BPE, transforming the input vector to the output vector. The Data Processing Fault occurs whenever the processing within a BPE differs from the specified transformation. This means for example that in this model control flow faults within the BPE are classified as Data Processing Faults.

Data Storage Faults pertain to the value of the input and output vectors of a BPE. A Data Storage Fault occurs whenever the message or H-state at the receiving BPE differs from the value of the transmitting BPE. This definition manages transmission faults as storage faults so distributed and non-distributed BPEs can use the same model and detection mechanisms.

Data Flow Faults pertain to the path of the input and output vectors of a BPE. The data flow is static and defined pre runtime in the CPE and PED description. It describes when and where each schedulable unit (BPE) shall read its input information and when and where the result shall be written. A data flow fault has occurred if a BPE, when executed, reads from or writes information into a data storage that differs from the specified ones, or if the timing differs from the specification.

Control Timing Faults pertain to the time domain of the PED. A control timing fault occurs when the actual control timing of a PED is not compliant to the specification, i.e. whenever the dead-line requirement of a PED during run-time is not met.

Control Flow Faults pertain to the precedence order of the PED. A Control Flow Fault occurs whenever the execution order of the BPEs in a PED during run-time differs from the one defined pre run-time. Note that in this model, control flow faults within the BPE is considered as Data Processing Faults.

4. Systematic Fault Tolerance

This chapter presents a framework for the achievement of systematic fault-tolerance. Whereas application-specific fault-tolerance uses knowledge about the application domain to achieve fault-tolerance in an ad-hoc way, systematic fault-tolerance uses replication of components to detect faults and applies redundancy to achieve continued service in the presence of faults.

Although systematic fault-tolerance potentially incurs higher costs than application-specific fault-tolerance it offers a number of advantages as well, specifically [14]: Independence of application knowledge. Broader applicability. Less software complexity. Separation of application and fault-tolerance functionality. Reusability of fault-tolerance mechanisms across applications. Reusability of application software in systems with different degrees of fault-tolerance.

The failure hypothesis of the BPEs is fail silence (FS) in value and time domain. This property is also shared by the CPE and PED. This means that the output vector of a BPE is delivered correctly at the correct time or not at all. Since the output vector is also subject to faults the assumed behavior of the BPE including the output vectors is *extended fail-silence* i.e. the BPE is considered FS even if erroneous vectors are sent, provided this status is detected at the receiving BPE.

To support the FS abstraction, a variety of error detection mechanisms (EDM) must be provided. These mechanisms will be described in section 4.1.

One objective of the X-By-Wire project is to find cost effective fault tolerant (FT) solutions. The chosen FT

mechanisms comply with these requirements. Solutions that use special hardware are not employed.

To obtain the FS property the input and output sections of the BPE (Figure 3) are monitored by software for error detection which is part of the System Services.

An advantage of using the BPE as a basis for the error detection is that the latency requirements for the error detection within a BPE are explicit in the model. Since no output is performed until the output section, there is no requirement for the detection to be effective until then.

In the following sections the EDMs and redundancy strategy is explained and finally some implementation aspects are discussed.

4.1 Error Detection Methods

To achieve a sufficiently high coverage for the fail silence assumption [18] it is necessary to employ extensive error detection strategies. There is a broad variety of error detection methods available. Some of them are hardware based (e.g. signature checking or watchdogs), while others are software based. In the following we are concerned with software based EDMs with high coverage which can be applied systematically, but also standard hardware as watchdog timers. By systematically applicable it is meant that the error detection strategy can be applied to a piece of software without any knowledge of the application domain. This for example excludes all types of plausibility and range checks.

Furthermore these EDMs can be applied locally to a node so no extra inter-node communication bandwidth is occupied. Experimental results have demonstrated a high error detection coverage by these strategies [7]. The EDMs will be described in the following chapters.

The different EDMs accounting for the XBW faults is shown in Table 1.

Fault	Error Detection Mechanism
Data Processing Fault	Double Execution / Double Execution with Reference Check
Data Storage Fault	Message Validity Check: Data redundancy coding
Data Flow Fault	Message Validity Check: Message ID + Message Time-stamp
Control Flow Fault	Block level Signature Check
Control Timing Fault	Watchdog timer

Table 1: Different EDMs handling different faults in the XBW-fault model.

4.1.1 **Double Execution.** For the double execution method (DEX) [22][4][21] the processing part of the BPE is executed twice per activation. After the second execution a generic comparison function is called to check the individual output vectors. If the two output vectors differ an exception is raised to treat this situation, e.g. to be fail silent. If the results are identical then the output of the BPE is made available.

Since the output vectors account for all output from the processing including the internal state of the BPE the error coverage is expected to approach 100% for transient or intermittent faults. Fault injection experiments [7] where DEX were combined with Message Validity Check indeed indicate this.

The BPE concept is very suitable for the DEX method since the output including the internal state is explicitly identified in the model. This means that a code tool can generate the test completely automatically.

Since the processing part of the BPE is executed twice on the same hardware resources, there is no possibility to detect permanent hardware faults. However, detection of transient faults is more important since their likelihood is orders of magnitude higher than the likelihood of permanent faults [22]. This is especially relevant for automotive electronics which must operate under harsh environmental conditions.

```
double execution:
  save in-vector
  exec process
  save out-vector
  restore in-vector
  exec processing
  save out-vector
  generic out-vector compare
  on error raise exception
  send messages (generate output)
```

Figure 6. double execution.

Double execution can only detect transient faults when: (1) The persistence of the transient faults is short enough not to affect both executions or (2) the transient fault affects both executions in a different way so that the output vector of the individual executions are different.

A possible measure to manage long transients is to schedule the two executions with an increased separation in time. This is also done in the Double Execution with Reference Check method presented in chapter 4.1.2.

The execution sequence for double execution is given in Figure 6. The time overhead for the DEX is at least 100%, but the absolute cost is decreasing with higher micro processor/controller performance.

4.1.2 **Double Execution with Reference Check.** Double execution (as presented above) allows for the detection of transient faults under the assumption that the fault does not affect both executions in the same way. Since it is possible that a transient (or permanent) fault, such as a latch-up, affects both executions in the same way it might be desirable to detect this type of faults as well. This can be facilitated by performing an additional execution of the BPE with reference ("golden") data between the first and

second execution. That is, the input vector is initialized with a given set of reference data, the processing part is executed and its output vector is compared against the known correct results for the given set of reference data. Physical fault injection experiments [7] on the MARS system [9] have demonstrated that this EDM might be a way to accomplish 100% coverage at least for transient faults. For reasonably chosen reference input vectors and moderate assumptions (like all user registers tested at least once for 0 or 1, etc.) the coverage for permanent faults should be at least 50%.

The execution sequence for double execution with reference data is given in the following figure:

```
double execution with reference data:
  save original in-vector
  exec process
  save out-vector-1
  in-vector := reference in-vector
  exec process
  compare out-vector and reference out-vector
  on error raise exception
  restore original in-vector
  exec process
  save out-vector-2
  compare out-vector-1 and out-vector-2
  on error raise exception
  send messages (generate output)
```

Figure 7. double execution with reference check.

As for the DEX EDM this method is suitable for automatic generation provided the reference vectors are defined for the BPE.

The time overhead for the method is at least 200%., but as for the DEX this cost is of decreasing importance.

4.1.3 **Message Validity Check.** Since information exchange between BPEs is carried out exclusively by means of message passing it is important to detect message mutilation and timing faults. This can be done by allocating redundant memory for messages. Upon sending a message, the sent message data is stored along with a calculated error detection code. Upon receiving a message, the message data are read and the error detection code is calculated over the message data plus the expected message ID. If there is a difference between the calculated and the stored error detection code an exception is raised since the message has been corrupted. Message validity checks can be implemented for intra- as well as for inter-node communication.

There is a broad variety of error detecting codes which can be used to detect message mutilations. However, depending on the message length different error detecting codes should be used for efficiency reasons. Typically, many messages have a length equal to the processors word length. For these messages it is advantageous to use the two's complement or exclusive-or as an error detection code. For longer messages error detection codes such as CRC are better suited since they have a higher error detection coverage.

The technique described above protects against faults in the value domain. It is possible to extend this check to the time domain by assigning time-stamps to messages. Upon sending a message, the activation time of the sender is saved along with the message. Upon receiving the message the receiver checks whether the message time-stamp is valid. This can be done by calculating the period between the activation time of the receiver and the activation time of the sender. If this period exceeds a specified latency then it is known that a message send operation has been omitted. The error detection techniques for the value and time domain can be combined by calculating the error detection code over the actual message data *and* the time-stamp.

4.1.4 **Block Level Signature Check.** Signature checking is a technique to detect control flow errors [20, 12]. At run-time, a signature is calculated as the processor executes a block of code. For software signature checking without user intervention (insertion of special statements) and without special compiler support the granularity of blocks must be selected in accordance to the granularity of the objects managed by the operating system. The Signature Check is executed in the Input and Output sections of the BPE to assure that the execution order of the BPE is according to the off-line definition and that the output is allowed.

In this signature checking the BPE serves as the block for which the signature is calculated and checked.

The reason why control flow faults within the BPE needs not be considered separately is because the double execution check covers all output from the processing, including the state of the BPE. Any transient control flow fault that does not manifest itself as an error does not need to be accounted for.

4.1.5 **Watchdog Timer.** The Watchdog timer EDM assumes the availability of a hardware watchdog timer on the executing node. For a hard real-time system the worst case computation time should be known. Prior to the execution of a BPE or PED, the timer is initialized with this value. If the watchdog timer expires prior to the termination of the BPE, an exception is raised to handle the situation.

For time-triggered PEDs the entire control timing is defined pre-run-time in the PED description. Consequently, the time when a certain schedulable unit (BPE) is to be invoked and has to be finished is also determined pre run-time.

4.2 Redundancy Strategy

The Redundancy Strategy deals with the tolerance of permanent faults and requires redundant hardware components, i.e. processing and communication hardware resources are replicated.

The replication of hardware components is done at the granularity of nodes and their peripherals—called smallest replaceable units (SRUs). Software components are replicated at the granularity level of subsystems, i.e., replicated and non-replicated BPEs may reside on the same SRU. This differentiation between the granularity level of replicated software and hardware components is advantageous for economic reasons. Such a structure makes it possible to replicate only critical software parts. This allows the application designer to select the necessary level of redundancy without incurring the high hardware costs for the complete replication of SRUs.

4.2.1 **Active Replication.** With active replication a BPE or an entire PED executes on different nodes in parallel in a distributed system. All instances receive the same inputs and generate outputs. If one node fails the remaining instance of the BPE or PED are able to continue their service on the remaining processing nodes. Examples of systems supporting active replication are SIFT [23], MAFT [8], and MARS [9]. The "State Machine Approach", as described by Schneider [19], treats this replication method in a very detailed manner. For hard real-time systems active replication is the technique best suited to mask faults transparently [14] since there is no delay for recovery actions. This means there is no difference in the timing, regardless of whether a node failure had occurred or not. Furthermore, with active replication the individual nodes are not restricted in their failure semantics since all decisions are taken strictly distributed. Due to these advantages active replication is supported as the basic fault tolerance strategy for permanent faults.

The major problem with active replication is the high effort for replica control; all the replicated BPEs must receive the same inputs and they must execute at approximately the same time to deliver the same outputs. However, there are various sources for non-deterministic behavior of replicated BPEs. Among them are slightly diverging inputs from sensors, dynamic scheduling decisions, inconsistent message delivery order, slight differences in the execution timing, time-outs, and uncoordinated accesses to clocks. All these sources may lead to diverging outputs of the replicated BPEs even though all nodes are correct. It is therefore necessary to enforce replica determinate behavior of the replicated BPEs [14].

The enforcement of replica determinism can be decomposed into two parts, The first (1) concerned with external input and (2) processing of information within a node.

(1) Non-determinism of external inputs is caused by the limited accuracy of any sensor. This may lead to a situation where two (or more) sensors map a continuous quantity to different discrete numbers. A divergence of at least one bit cannot be circumvented [14]. This divergence can occur in the value domain as well as in the time domain. It is therefore necessary to provide the application software with a framework that allows replicated subsystems to achieve agreement. This can be achieved by providing the application software with a framework called RDA-Messages [16] (Replica Deterministic Agreed-messages). This message mechanism will allow replicated subsystems to reach agreement on replica non-deterministic values before they are used. Details of this mechanism is out of the scope of this paper.

(2) Actively replicated BPEs can show replica non-deterministic behavior because of slight divergence in the execution timing. To avoid node internal non-determinism it is necessary to achieve total agreement on the order of message receive and send operations. In [17] it has been shown for real-time systems that—except for external events—no communication is necessary at all to achieve agreement. This can be attained by means of Timed Messages which enforces replica determinism on send/receive behavior of replicated processes. For a more detailed discussion the reader is referred to [17,14].

4.2.2 **Implementation aspects.** A weakness of the software based EDMs described in chapter 4.1 is that the System Services assumed to execute the detection mechanisms are partly unguarded. For example nothing prevents a transient fault on the processor address register or decoder to cause an illegal branch into the output section of a BPE, which could cause uncontrolled output. A countermeasure for this is to use the inherent redundancy of a processor, for example, by adhering Hamming Distance separation (the number of bits that differ between two binary codes or words) between on one hand, the addresses of the processing part of a BPE including the error detection before any input/output operation, and on the other hand the input/output operation. This prevents any FS violation caused by a single bit address fault. As indicated by physical fault injection experiments [6] [2] single bit errors can be assumed to account for more than 90% of all errors.

For a distributed implementation using this model the efficiency can be increased by using a suitable communication protocol, like the TTP protocol [10]. This way many of the fault tolerance tasks can be done on the

communication layer of the system, for example: the time tagging of messages used in chapter 4.1.3 can be omitted in a time-triggered communication system since all involved timing is known pre run-time.

5. A sample implementation: Steer-by-wire

As a part of the X-By-Wire project a prototype steer-by-wire demonstrator will be developed to evaluate the architecture proposed within the project, and its appropriateness for future X-By-Wire applications in vehicles. Other parts that will be evaluated are the process, methodologies and detailed solutions but this is out of the scope of this paper.

The demonstrator is only briefly described here. A more detailed description can be found in [3].

5.1 Overview

The main function of the prototype is to implement a vehicle steering function without mechanical link between the steering wheel and the driving wheels.

This will be done by computing the vehicle steering angle, starting from the driver's input, taking into account vehicle handling and vehicle situation, and consequently move the road wheels. To improve the driver's feeling of the vehicle's behavior, a steering wheel feedback actuator will be used.

The control system of this demonstrator is described in the XBW model. An overview of the system is presented in Figure 8.

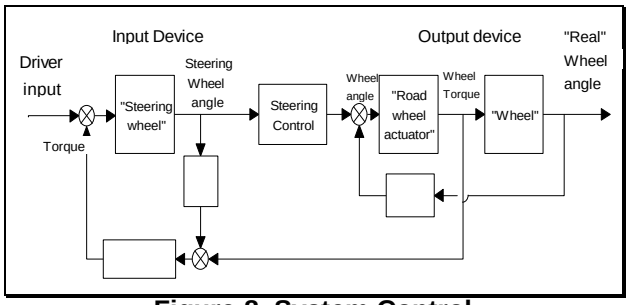

Figure 8. System Control.

The control system is implemented using commercial off-the-shelf controller nodes based on Motorola MC68360 communicating and synchronized via the dual channel bus using the fault tolerant time-triggered TTP protocol [10].

The controller nodes work as Fail-Silent Units (FSUs) using the principles presented in this paper. Each FSU is replicated in two or more instances forming distributed Fault Tolerant Units (FTUs). Each FSU communicates on the bus via a TTP bus controller implementing the TTP protocol.

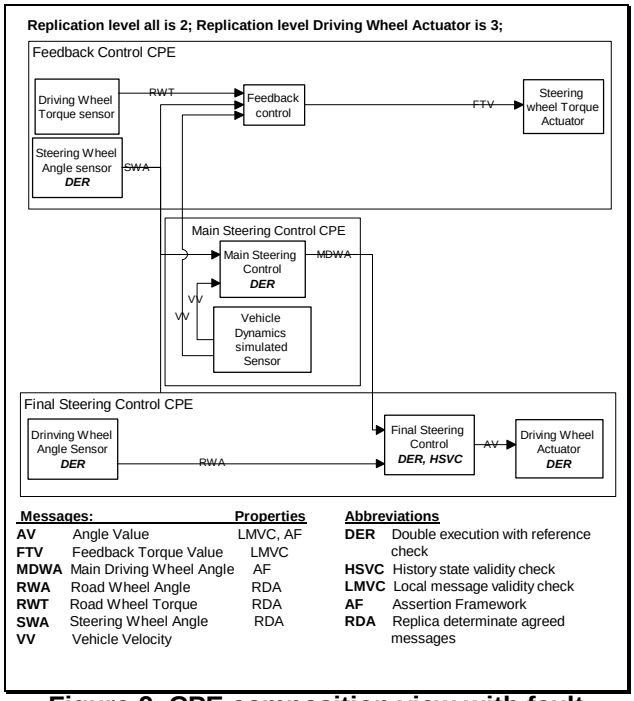

Figure 9. CPE composition view with fault tolerance mechanisms.

The system services for the execution and fault tolerance is supplied by the ERCOS [13] real-time operating system and a prototype xOLT off-line tool according to the DFR [16] concept for the systematic fault tolerant services.

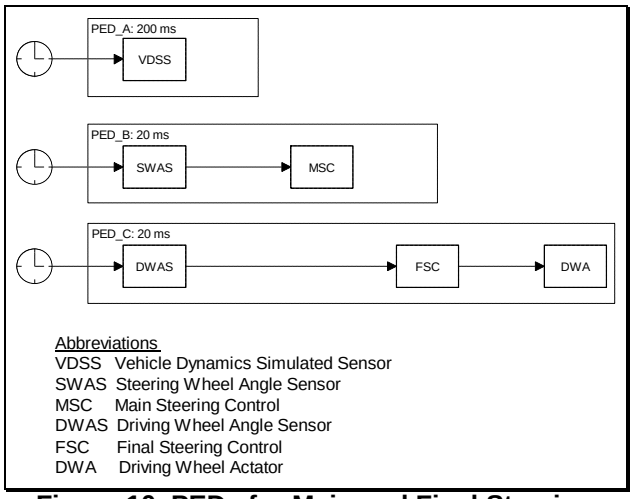

Figure 10. PEDs for Main and Final Steering control.

The required FT property of the prototype is FO/FS (Fault Operational/Fail Safe). This means that any one permanent fault must be tolerated and still providing specified functions. Any additional transient fault must be managed in a safe way. For a steering system that means that steering function must still be functioning.

Three FTUs are used, the Steering Wheel FTU, the Main Steering Control FTU and the Driving wheel FTU. The sensors for the angle and torque measurements are also replicated, one sensor connected to each FSU. The actuators are also replicated and connected to an FSU.

Figure 9 describes the control system using the XBW model and the real-time processing is described in the PED view in Figure 10.

6. Summary

This paper presents the XBW model, a conceptual model that uses easy to understand graphical syntax, combined with attributes to get a description suitable for use with systematic fault tolerance. It further includes a fault model that enables analysis of fault tolerance on the system or application level. Further it proposes a systematic application-independent framework for fault-tolerance. It is shown how the XBW-model together with the fault model supports the use of error detection mechanisms such as *double execution, double execution with reference check, validity checks for messages, Block level Signature Check* and *Watchdog Timer*. These error detection mechanisms are well known and combined they give a high error detection coverage. They are necessary to fulfill the fail-silent assumption in the model. These mechanisms are independent of the semantics of a specific application and can thus be applied systematically. Each of the mechanisms can be implemented without coupling the application software to a specific fault-tolerance strategy.

7. Acknowledgments

The XBW model is based on the contribution of all the partners in the X-By-Wire project, including Daimler-Benz (coordinator), Centro Ricerche Fiat, Chalmers University of Technology, Ford Europe, Magneti Marelli, Mecel, Robert Bosch, Vienna University of Technology, and Volvo.

The Brite-EuRam III Project *Safety Related Fault Tolerant Systems in Vehicles* - "X-By-Wire" is partly funded by the European Commission.

8. References

[1] T.P. Baker. "Stack-Based Scheduling of Realtime Processes." *The Journal of Real-Time Systems*. 3, 1991, pages 67–99.

[2] G. Choi, R. Iyer, and D. Saab, "Fault Behaviour Dictionary for Simulation of Device-level Transients", in *Proceedings of the IEEE Int. Conf. Comp. -Aided Design*, Nov. 1993, pages. 6-9.

[3] V. Claesson "Prototype Implementation using the XBW Software Model" *National Swedish Conference on Real-Time Systems*, SNART 1997. Also as Technical Report no. 98-7, Department of Computer Engineering, Chalmers Univ. of Tech., Göteborg, Sweden, 1998.

[4] A. Damm, "The effectiveness of software error-detection mechanisms in real-time operating systems", *Proceedings of the 16th Int. Symposium on Fault-Tolerant Computing (FTCS-16)*, IEEE, Vienna, Austria, June 1986, pages 171-176

[5] H.A. Hansson; H.W. Lawson; O. Bridal, C. Eriksson; S. Larsson, H. Lönn, M. Strömberg "Basement: an Architecture and Methodology for Distributed Automotive Real-Time Systems" *IEEE Transaction on Computers*, Sept 1997.

[6] R. Johansson, "On Singe Event Phenomena in Microprocessors", *Technical Report no. 162L*, Department of Computer Engineering, Chalmers Univ. of Tech., Göteborg, Sweden, 1993.

[7] J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, and G. Leber. *Integration and Comparison of Three Physical Fault Injection Techniques*. In Predictably Dependable Computing Systems. B. Randell, J.-C. Laprie, H. Kopetz, and B. Littlewood (eds). Springer, 1995, pages 309–327.

[8] R.M. Kieckhafer, P.M. Thambidurai, C.J. Walter, and A.M. Finn. "The MAFT Architecture for Distributed Fault-Tolerance". *IEEE Transactions on Computers*. Vol. 37, Nr. 4, 1988, pages 394–405.

[9] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, C. Senft and R. Zainlinger. "The MARS approach". *IEEE Micro*. Vol. 9, Nr. 1, Feb. 1989, pages 25–40.

[10] H. Kopetz and G. Grünsteidl. "TTP – A Protocol for Fault-Tolerant Real-Time Systems." *IEEE Computer*. Vol. 27, No. 1, Jan. 1994, pages 14–23.

[11] J.C. Laprie (Ed). "Dependability: Basic Concepts and Terminology." *Volume 5 of Dependable Computing and Fault-Tolerant Systems*, chapter 5.2, Fault-Tolerance. Springer Verlag. Wien, New York. 1992, pages 23–28.

[12] G. Miremadi, J. Karlsson, U. Gunneflo, and J. Torin. "Two Software Techniques for On-Line Error Detection". *22th*

International Symposium on Fault-Tolerant Computing. 1992, pages 328–335.

[13] S. Poledna, T. Mocken, J. Schiemann and T. Beck. "ERCOS – An Operating System for Automotive Applications." *In SAE International Congress and Exposition*. Detroit, MI, USA. 1996, pages 55–65.

[14] S. Poledna. *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism*. Kluwer Academic Publishers. 1996.

[15] S. Poledna,. "Optimizing Interprocess Communication for Embedded Real-Time Systems." *Proceedings of the Real-Time Systems Symposium.*, Washington. 1996.

[16] S. Poledna, C. Tanzer. "DFR Objects: A Meta Object Model for Distributed Fault-Tolerant Hard Real-Time Systems". *TTTech Technical Report*. 1997.

[17] S. Poledna. "Deterministic Operation of Dissimilar Replicated Task Sets in Fault-Tolerant Distributed Real-Time Systems". In *Proccedings of the Sixth IFIP International Working Conference on Dependable Computing for Critical Applications (DCCA-6)*. Springer Lecture Notes Series. Grainau, Germany. Mar. 1997.

[18] D. Powell. "Failure Mode Assumptions and Assumption Coverage". In Proceedings of the *22nd International Symposium on Fault-Tolerant Computing*. Computer Society Press of the IEEE. Boston, Massachusetts. Jul. 1992, pages 386–395.

[19] F. B. Schneider, (1990). "Implementing Fault-Tolerant Services Using the State Machine Approach A Tutorial." *ACM Computing Surveys* Vol 22(nr 4): 299-319.

[20] N.R. Saxena and E.J. McCluskey. Control Flow Checking Using Watchdog Assists and Extended-Precision Checksums. *IEEE Transactions on Computers*. Vol 39, Nr. 4, April 1990, pages 554–559.

[21] M. Schuette, J. Shen, D. Siewiorek and Y. Zhu, "Experimental Evaluation of Two Concurrent Error Detection Schemes", *Proceedings of the 16th Int. Symposium on Fault-Tolerant Computing (FTCS-16)*, IEEE, Vienna, Austria, June 1986, pages 138-143.

[22] D. Siewiorek and R. Swarz, *The Theory and Practice of Reliable System Design*, Digital Press, 1982, pages 18-19.

[23] J.H. Wensley, L. Lamport, J. Goldberg, M.W. Green, K.N. Levitt, P.M. Melliar-Smith, R.E. Shostack and C.B. Weinstock. "SIFT: The Design and Analysis of a Fault-Tolerant Computer for Aircraft Control". *In Proceedings of the IEEE*. Vol. 66, Nr. 10, Oct. 1978, pages 1240–1255.