



HAL
open science

An approach for improving Fault-Tolerance in Automotive Modular Embedded Software

Caroline Lu, Jean-Charles Fabre, Marc-Olivier Killijian

► **To cite this version:**

Caroline Lu, Jean-Charles Fabre, Marc-Olivier Killijian. An approach for improving Fault-Tolerance in Automotive Modular Embedded Software. 17th International Conference on Real-Time and Network Systems, Oct 2009, Paris, France. pp.132-147. inria-00442000

HAL Id: inria-00442000

<https://inria.hal.science/inria-00442000>

Submitted on 17 Dec 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

An approach for improving Fault-Tolerance in Automotive Modular Embedded Software*

Caroline Lu^{1,2,3}
¹RENAULT Technocentre
1, Avenue du Golf
78288 Guyancourt Cedex
caroline.lu@renault.com

Jean-Charles Fabre^{2,3}, Marc-Olivier Killijian^{2,3}
² CNRS ; LAAS ; 7 avenue du colonel Roche,
F-31077 Toulouse, France;
³ Université de Toulouse ; UPS, INSA, INP, ISAE ;
{jean-charles.fabre, marco.killijian}@laas.fr

Abstract

Error detection and error recovery mechanism must be carefully selected in automotive embedded applications mainly because of limited resources and economical reasons. However, major safety concerns, brought by new customer services (i.e. chassis control), motivate the automotive industry to search for new means for improving robustness in operation. The challenge is to study a "low-cost", portable and flexible dependability solution. The guiding principle is to rigorously control what/when information is essential to get, and what/when instrumentation is necessary, to perform fault-tolerance. The paper proposes an approach to develop a defense software, as an external customizable component, based on observation and control mechanisms provided by current standard in the automotive industry.

1. Introduction

Improving software fault-tolerance is a common interest for aeronautics, railway and automotive software-based systems. However, the automotive context meets more stringent economical constraints and resources limitations, due to higher volume of vehicle production and lower criticality of vehicle functions compared to avionics. A "lightweight" solution for fault tolerance is studied.

To optimize online verification means to avoid exhaustive systematic information storage and checks, thanks to preliminary safety analyses. Identifying, at first, major critical data and control flows of application software enables to perform selective verification. The drawback of such an application-specific approach would be a lack of adaptability and

portability for reuse of safety mechanisms, if they are not well organized and coordinated.

Our approach favors reuse by applying the "separation of concerns" principle [1] to realize customizable defense software. The defense software, implements a fault-tolerance strategy and is separated from functional software. Both software parts interact with each other only through an instrumentation interface. The error monitoring strategy is application-specific and derived from safety analysis, whereas the instrumentation interface between functional and defense software is as generic as possible. This way, if functional software evolves, the interface may evolve but the strategy and defense software remain unchanged. On the other hand, if defense software has to evolve due to arbitrary change (addition or removal of a strategy) the interface may be adapted, without any change on functional software. Feasibility of the presented framework and robustness improvements has been experimented with several prototypes. Efficiency of the defense software is evaluated by fault injection.

The aim of the work reported in this paper is to present the approach and the principle of the fault-tolerance framework. It addresses particularly interaction errors between application software and lower software layers. Section 2 precisely describes the context of the work, in terms of automotive software architecture, fault model and safety requirements. The fault-tolerance framework and a corresponding development cycle are proposed in Section 3. The architectural solution is discussed according to two main aspects: design of defense software (Section 4) and instrumentation interface (Section 5). Finally, Section 6 gives an idea of early implementation issues.

2. Automotive software context

New major standards are emerging in the automotive landscape and will probably influence

* This work has been partially supported by the SCARLET project financed by ANR (the French science foundation, ground transportation research programme PREDIT) focused on robustness of executive software in critical automotive applications.

dependability of tomorrow's embedded software. The first one, AUTOSAR [2], standardizes complex automotive software, structuring it in modules and abstraction levels. We focus particularly on the interaction between application and basic software modules. Another standard, ISO-26262 [3], aims at promoting functional safety measures, at each step of the development cycle of a product.

2.1. Fault model

An automotive embedded system may fail in operation due to either physical faults (hardware, EMC, etc.) or residual bugs from design or development phase of the software development process [4].

Physical faults are modeled as permanent and transient bit-flips and stuck-at in the code and data memory segments. This kind of fault is always possible due to the aggressive environment of automotive applications and the increasing complexity of the hardware components and system architecture.

Bugs during design may occur due to non respected rules for design (MISRA), bad temporal design (sizing, execution order, etc.), bad resource sizing, bad data usage (wrong choice of data for usage, wrong handling of a data, etc.) or non expected modes. Bugs at development phase are likely to happen during manual coding, because of misinterpretation of specifications, coding errors, compiler or linker's default.

A new growing trend is automated code generation. Then scaling or configuration of tools may be wrong (it is enhanced by software complexity). The adaptation to the generator's constraints may be uncomplete (e.g. multiplication of boolean values is not optimal and not supported by all generators). Generator's defaults (especially if the tool is not certified) may lead to errors on the generated code. The integration phase takes a major role in the context of component-based systems and the use of *Components-Of The-Shelf* (COTS) as black boxes most of the time. At the integrator level, there may be again misinterpretation of specifications, coding errors (of glue code), bad scaling of global data, use of bad module version or configuration, and compiler or linker's default.

Actually, the statistical distribution of fault and their diversity are not the major interests from the application software viewpoint. All these faults result in transient or permanent failures on the functional data and control flow. Application-level faults are easier to translate into customer effect, and can be evaluated depending on levels of potential threat or undesirable event to people.

2.2. Automotive safety constraints

A given automotive embedded system is described by a set of specification documents and/or models. They are derived from functional and mechatronics

levels to application software design requirements. Safety analysis identifies a list of "unwanted system events (USE)" at application software level. These USE can be potentially safety-critical or not. In the first case, the customer may be endangered, whereas in the second one leads at worst to a dissatisfaction of the customer. Safety barrier must be designed to avoid both these types of unwanted events.

The selection of safety properties from these specifications is defined case-by-case for a given project, taking into account economical, hardware and software sizing constraints.

About potentially critical unwanted system events, the ISO26262 standard defines four safety levels called ASIL (Automotive Safety Integrity Level). They are graded from A to D level with a respectively increasing criticality. Each level is given a set of requirements within which safety methods and mechanisms are listed with graduated recommendation. Therefore, the proposed protection framework enables focusing on highly critical (ASIL C-D) functions and/or information only.

3. Framework overview

The proposed fault-tolerant architecture relies on "computational reflection" [5]. Basic concepts and the overall methodology are given before describing the defense software design and implementation.

3.1. Reflective Principle

The reflection paradigm [6] for fault-tolerance purpose relies on the ability of a system to check and to correct itself in a separate abstraction level. In practical terms, the software architecture (Figure 1) is clearly divided into two parts (functional and defense software) that interact together via an interface [7, 8]. The defense software has enough knowledge of the structure and expected behavior of functional software, to control it. To apply this principle to a given functional software, the main activities concern the definition of:

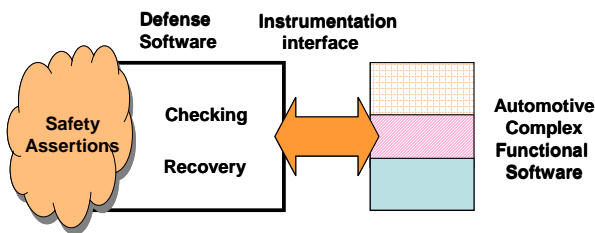
- Safety Assertions (Sections 3.5);
- Defense software (Section 4);
- Instrumentation interface (Section 5).

The fault-tolerant architecture corresponds to the defense software and the instrumentation interface. The defense software detects errors by checking safety properties and performs recovery using generic instrumentation and infrastructure functionalities.

The idea is similar to other industrial solutions to improve system robustness and safety in railways and aeronautics applications. In the electronic interlocking system Elektra [9] a two-channel-approach (notion of safety bag) performing specification diversity is used

for detecting software design faults. Airbus command and control systems rely on the notion of self-checking component composed of command and monitoring computers, in the series A320 to A380 [10]. However, such architectural solutions are not viable for the automotive industry for the time being, due to strong constraints on resources. A lightweight solution may be less robust than these systems.

Figure 1. Reflective System.



3.2. Framework

Following the reflective principle, fault-tolerance relies on the knowledge (a model) a system has of itself and safety properties. The accuracy of the knowledge determines the ability of the system to control its state and behavior. This is why a few refinement steps, using a top-down approach, may be necessary to improve fault tolerance.

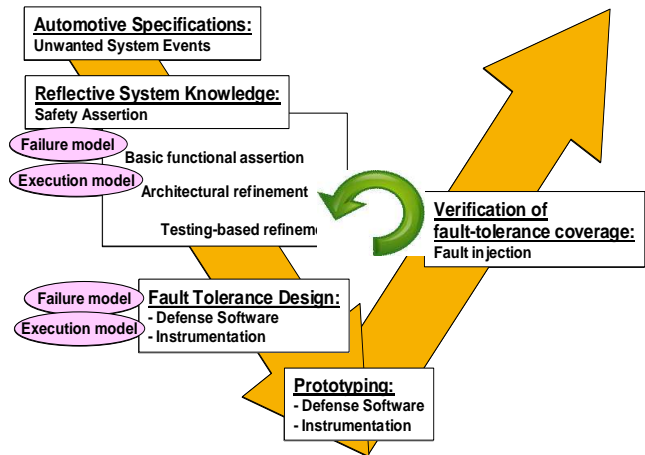
Figure 2 describes the main parts of the overall framework. Automotive unwanted system events are translated into basic safety assertions, according to a functional “failure model”, described in Section 3.3. At this stage, fault-tolerance is still designed at the application software level, ignoring the execution support. To deal with the real embedded system, a first refinement step of the safety assertions aims to take into account the software architecture and underlying infrastructure. It relies on a simple “execution model” (Section 3.4) of the system in operation.

Then, for each safety assertion, the corresponding defense software and instrumentation can be defined and implemented (Sections 4 and 5). In our approach, verification of fault-tolerance coverage is performed by fault injection. Depending on the results, a new refinement of safety assertions can be carried out, and fault-tolerance software design adapted accordingly. The process can be repeated iteratively until the expected fault-tolerance coverage is reached.

The next sections introduce briefly the two concepts of “failure” and “execution” models that are used, as a support to face the diversity of automotive applications. The “failure model” is an input of the definition of fault tolerance mechanisms. Regarding implementation, an “execution model” is essential to analyze the impact of faults and related errors on the system behavior. For this purpose, we do not need complex formalisms. We just need a simplified

representation of complex systems, highlighting specific concerns. From this model, we can factorize automotive safety needs into a limited number of categories, for which generic protection mechanisms can be defined and developed.

Figure 2. Overall framework.



3.3. Functional failure model

At application software level, we structure the failure model into two parts: data flow and control flow. Failures can impact both data and control: data often accompany the control or when data moves, control can be activated. The considered classification is not orthogonal. From the control flow viewpoint, the first question is the internal scheduling of computation steps within a software component, whereas data flow mainly means reasoning about data properties, availability, transformation and latency. The user can arbitrarily identify a data failure, a control failure or both, depending on his major concern.

Critical control flow failures.

They fall into 3 categories. The first one targets *control events*, which impact directly or indirectly, the activation or termination of execution of a treatment. Another type is a defect on the *sequence of execution*, either at the application level or at lower levels. The last category of control failure affects the *execution time* (deadline, timing evolution, periodicity, etc.).

Critical data flow failures.

Value and timing defects can be separated. Data include variables or exchanged messages, as inputs or outputs of software modules. A *value* may be faulty within a correct range or out of range. We may also have complex requirements on the values of a set of data (symbolic expression or equation). If functional *timing* constraints are explicitly given on data, we relate them to data communication instead of time of execution.

3.4. Execution model

The difficulty to combine a classical state-transition graph with the considered failure model, to highlight potential error sources, leads to introduce a dedicated simple representation. The objective is to describe the runtime behavior of a system as a sequence of “scheduled entity”. The “scheduled entities” are triggered by events and generate triggering events. They are also data consumers and producers.

The “scheduled entity” generic expression gathers two viewpoints. For the operating system, “scheduled entities” correspond to “tasks”. However, for the designer of the application, implementation issues, including tasks mapping, are generally unknown. For example, if applications are developed originally with Simulink tools, functional requirements specify the sequence of execution of “application level” connected boxes. Such “application level” boxes of the Simulink model become application-level functions after code generation. Consequently, these application level functions are also considered as “scheduled entities”. The mapping of a function within a task is the job of the integrator.

Control flow of a scheduled entity relates to the control events starting or stopping its execution. These events are produced by the environment or other entities. In parallel, *data flow* of a scheduled entity corresponds to the input data it consumes and the output data it produces during its execution. Interactions (and error propagation sources) through the software architecture at runtime are therefore based on data exchange and control events, and potential interweaving of scheduled entities.

The granularity level of the “failure” and “execution” models enables to deal with different automotive applications, from air-conditioning to torque control modules.

3.5. Design steps and refinement process

An example is given now, showing the way a real automotive unwanted system events is used in order to define the “reflective system knowledge” (Figure 2).

Basic functional assertion example.

Preliminary safety analysis of an automotive system identifies a list of “unwanted system events (USE)” (Section 2.3). These USE are the requirements, so to speak initial or basic functional assertions. For example, a realistic USE on an automotive transmission module could be the following:

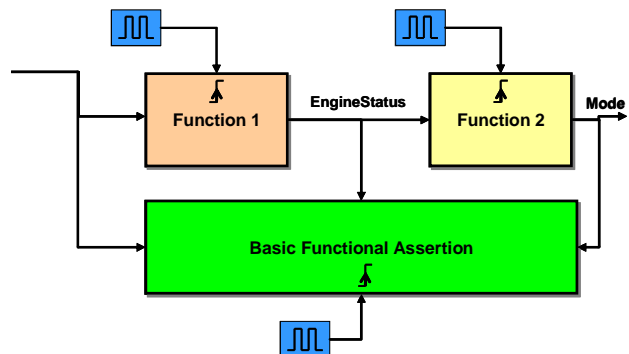
“The system is blocked (more than 1 second) in mode A, while the engine status is equal to 2, whereas it should switch to mode B”.

Figure 3 shows a Simulink-like model of the automotive application (“Function1”, “Function2”), which is targeted by the USE. Other functions, inputs and outputs of the real system have been hidden to keep the example simple. In the automotive context, “Function1” belongs to the static control module that interprets driver commands and environment measures. “Function2” is a part of the dynamic system control that computes the engine torque set-point. Figure 3 also shows the defense software that implements the “basic functional assertion” to be verified and that receives application critical data as inputs.

At a coarse abstraction level, our defense software is limited to a module that verifies the basic assertion, and is eventually able to switch the system in a predefined degraded mode. To perform the verification, it is mandatory to dive into low-level details to solve the following issues:

- 1) how/when to catch and store the information required to perform the verification (run the executable assertions with all parameters fixed).
- 2) how/when to perform the verification within the control flow of the system (thanks to the execution model) and when triggering the error recovery.

Figure 3. Example at functional level.



In this example, the considered type of fault is either the loss of a control event or a wrong data event. Both types of fault may lead to a miss of the change of mode. To derive the executable assertion from such high-level analysis, a refinement taking into account the underlying software architecture is necessary.

Architectural refinement example.

To perform the refinement, low-level implementation details are needed, either from underlying executive layers or from the communication services

For the given example, the input event of “Function1” (Figure 4) is implemented by the value recorded by a hardware sensor. It is transmitted to “Function1” every 10ms by a periodic task, reading the value of the sensor. The output of “Function2”, called “mode” must be consistent with the sensor value and

the engine status, according to the USE. An error may happen when a corrupted data is read by the sensor.

After a careful analysis of the way these functions have been implemented (i.e. mapped to OS or middleware objects, connected to each other and to the external world, the execution profile and which core parameters have been considered), a refined version of the assertion can be expressed, for instance:

“At the end of each sensor task, the mode is consistent with the value of the sensor, while the engine status is equal to 2”.

This refinement process enables the initial USE related assertion to be expressed in computing terms. It is worth noting that the final assertion depends on the implementation of the functions. It enables to identify:

- the required information to perform the check (e.g. sensor value, mode, engine status);
- the logical and/or numerical expression or the algorithm corresponding to perform the check ;
- where the check of the assertion has to be performed in the execution flow (e.g. end of sensor task);

Refinement evaluation.

After prototyping the defense software for the selected assertions, an evaluation phase can be started. We perform verifications by a fault injection technique, based on (i) the considered failures (cf. failure model) and (ii) the USE. Control flow failures are realized by the insertion of system calls in the program, whereas data flow is disturbed by selected communication service calls.

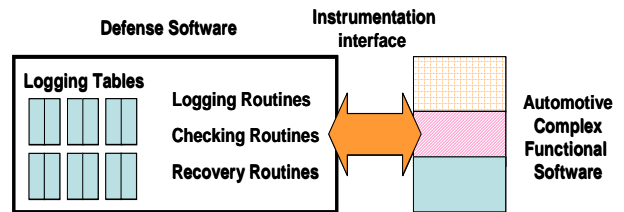
If the fault-tolerance coverage rate and residual failure modes do not match the expected results, then a refinement is needed. This involves diving into deeper analysis of the implementation, deeper knowledge of the software architecture or revising some steps in the assertion refinement process.

Finally, the refined assertions can be included into the defense software. This is done naturally since the defense software architecture is adaptable by construction thanks to the reflective approach. Then the “refinement-evaluation process” starts again until the expected fault-tolerance coverage rate is obtained.

4. Defense Software

The defense software (Figure 4) is organized around logging tables and three types of services that control (i) information logging (“*logging routines*”), (ii) error detection (“*checking routines*”), and (iii) error recovery (“*recovery routines*”). This application-related part of the fault-tolerant architecture is specified from a given set of selected safety properties.

Figure 4. Defense software organization.



4.1. Logging strategy and architecture

Logging or tracing are mechanisms often needed for debugging and diagnosis issues. Amount of tracing information depends on the objectives of the user: defect analysis with possibility to reproduce the failure scenario (extended tracing), defect analysis only to remove a bug (local tracing), performance profiling to determine where the system spends its execution time (selective tracing), etc.

The capture of software trace induces significant execution slowdown. Classical automotive applications (e.g. Body Control Module), within an ECU, may exchange several thousand of data, and may be controlled by several dozens of tasks that include both application and infrastructural tasks. Reducing overhead requires either sacrificing details or using hardware extensions.

As a result, the logging strategy has to select rigorously the *necessary and sufficient* critical information to get at runtime, according to fault-tolerance concerns. Then, the structure of storage is a major factor to reduce timing access to information for online error detection. Actually, software architecture for logging must be specified to favor reuse, adaptability with the diversity of automotive applications, and portability on different platforms.

The proposed logging strategy is derived from the model of execution from section 2. In order to supervise data and control flow, the system should record a history of task switches, application-level functions entries and exits, some system calls and data communication. Then, information logging consists in storing only events that belong to a critical flow. This preliminary selection (resulting in less than a hundred of critical data and a few critical tasks, for example) enables stored information to be redundant and diversified. It provides multiple viewpoints, like OS/application or data/control flow, that are of high interest for improving fault-tolerance. Instrumentation to catch information is detailed in section 5.

The logging architecture is organized into several “**bracket-tables**” that are updated and used at runtime. Any storage table can be considered as an opening or closing “bracket”. Each logging table has an associated table because information they store is symmetric. For

instance, when information regarding the start of a task is stored in a table (“opening bracket”), the information regarding the end of the same task is stored in an associated table (“closing bracket”). The number of tables depends on the number and the complexity of safety properties to be protected by the defense software. Tables should be kept small to reduce scanning of information by error detection routines. The “bracket-tables” are sorted into 4 categories:

- **The execution trace from OS viewpoint:** when a critical task starts execution, an “opening-bracket-table” entry is filled basically with the task identifier and a timestamp. The “closing-bracket-table” stores the same type of information, when the task ends (not when it is preempted). The length of the table depends on the complexity of the safety properties. For example, if a periodical sequence of execution must be verified, the length is that of the sequence. This category contains at most one couple of bracket-table (opening/closing).
- **The execution trace from application viewpoint:** when a critical application-level function starts, an “opening-bracket-table” entry is filled basically with the function identifier, the task identifier in which the function runs and a timestamp. The associated “closing-bracket-table” stores the same type of information, when the function ends (not when it is preempted). As above, the length of the table depends on safety properties. This category contains at most one couple of bracket-table that may replace (if functions are considered more meaningful than tasks) or be redundant with the preceding table providing the OS viewpoint.
- **The control event trace:** when an activation event (that impacts directly or indirectly the activation of a task) happens, an “opening-bracket-table” entry is filled basically with parameters that characterize the event, the current running task identifier, and a timestamp. The “closing-bracket-table” stores the same type of information, when the termination event occurs. This category may group several bracket-tables because there are several types of control events.
- **The data event trace:** when a critical data is written, an “opening-bracket-table” entry is filled basically with the data, the function identifier that produces the data, the task identifier in which it runs and a timestamp. The “closing-bracket-table” stores the same type of information, when the data is read. This category may group several bracket-tables because there are several types of data communication services.

Each table is associated with a dedicated routine (“logging routine”) that uses preferably existing infrastructure services to get information (“basic sensor services”, Section 5.2). When these services are not

available, data are collected through the routine parameters, or by additional instrumentation (“hooks”, Section 5.1).

4.2. Error detection strategy

Once application-specific safety properties are specified, the corresponding error detection routine is developed as an executable assertion. An assertion is verified at runtime within a corresponding “checking routine”. When an error is detected, the checking routine triggers a “recovery routine” (Section 4.3).

Safety properties may address critical data flow, control flow or both. The knowledge of defense software, about the behavior of functional software, and more particularly the critical flows, is gathered into the logging tables (that must be trusted). Checking routines rely on the information stored in these tables to verify assertions. The structure of logging tables is given in section 4.1. The content of tables is equal to the information that is needed to check safety assertions and recover from errors. As a result, this content (which event, when, how many) is determined from the safety assertions.

Assertions are pre- or post- condition at a verification point. The verification point depends on the safety property and if error detection “as soon as possible” is expected or not. For example, it can be the beginning of execution, a waiting point within a task, or the reception or emission of a control event, etc.

According to 3.3 that describe the main types of failures, the corresponding types of assertions address 1) control events, 2) sequences of execution, 3) timing constraints of execution, and 4) values or 5) timing constraints on data exchange.

Table 1. Basic reference to logging tables.

Assertion with:	Logging tables
Control event	Control event trace
Sequence of execution	Execution trace
Timing constraints of execution	Execution trace
Value constraints on data	Data event trace
Timing constraints on data	Data event trace

The analysis of an example gives an idea of how to derive the checking routine from an assertion such as:

“The acknowledgement of reception of Message 1, notified to Task 1, at latest 2ms after Message 1 has been sent, allows Task1 to activate Task 2, else Task 3 must be activated”.

“The acknowledgement of reception of Message 1” and the “activation of Task 2” are control events. “At

latest 2ms after Message 1 has been sent” is a timing constraint on data. The pseudo-code of the checking routine (called before the execution triggering of Task2) for this assertion is given in Figure 6.

Figure 6. Checking routine.

```
Check_P1 {
If {
/* Check in the control event trace of "ActivateTask" system call to find activation
of Task 2 by Task1 */
    LogTable_ActivateTask.ActivatedTaskID[i] == Task 2;
    LogTable_ActivateTask.RunningTaskID[i] == Task 1;
    LogTable_ActivateTask.Return[i] == ok;
/* Check in the control event trace of "DataAcknowledgement" service call to find
notification of reception of Message 1 to Task 1 */
    LogTable_DataAck.TaskID[j] == Task 1;
    LogTable_DataAck.MessageID [j] == Message 1;
    LogTable_DataAck.Return[k] == ok;
/* Check in the date event trace of "DataSent" service call to find time of emission
of Message 1 by Task 1 */
    LogTable_DataSent.TaskID[k] == Task 1;
    LogTable_DataSent.MessageID [k] == Message 1;
    T1= LogTable_DataSent.Time[k];
/* Check the timing constraint */
    T2 = LogTable_DataAck.Time[j];
    T1 - T2 < 2ms;
}
/* An error is detected */
Else      Recov_P1();
}
```

4.3. Error recovery strategy

Error recovery from application and from infrastructure viewpoints have complementary advantages and limitations. To reduce error latency and improve efficiency the core idea is to use infrastructure recovery controlled by application level consideration.

Degraded modes of automotive applications are generally very sophisticated at the system and application level. Once an error is detected the application is turned into a safe state. About data flow, degraded data is usually known to recover from invalid values. Missed timeout or acknowledgement of data exchange may lead to new communication requests or use of degraded values again. About control flow, apart from reset, recovery actions are limited, it consist in a change of working mode or application-level functions inhibition.

At the infrastructure level, recovery actions on control flow are also basic: reset, terminate and restart a task or a set of OS objects. Recovery actions are difficult to take without knowledge of the application. Killing and restarting an air conditioning, an airbag or

a torque control module has not same impact. From the application level, the support of execution is not supposed to take alone uncontrolled recovery decisions that could leave the system in an unexpected state. It is worth noting that infrastructure services represent a collection of software actuator (Section 5.3), which can improve fault-tolerance.

In the proposed fault-tolerant architecture, each “checking routine” is associated with one or more “recovery routines”. Recovery routines are calling available executive services (“basic actuator services”, Section 5.3) and update logging tables, if necessary.

The recovery action depends on the detected error. Going back to the example given in section 4.2, an example of pseudo-code of the recovery routine would be:

```
Recov_P1 { /* Error: Task 2 must not be activated but Task 3 */
ActivateTask (Task3); }
```

Control flow error recovery.

If a lost control event has been detected, the logging tables should have stored the event so that we can chain the correct treatment (activate a task, wake up a waiting task, etc). Another option is to duplicate the event and make sure it is received. On the contrary, if a wrong or untimely control event has been detected, it should be cleared.

If an error in the sequence of execution is detected, two types of recovery can be considered. Usually, we can terminate or chain a task, to restart another task within a degraded mode, or the same task. Otherwise, the whole application software component can be stopped, reinitialized and restarted.

Data flow error recovery.

In case of data flow error, a first option is to update data values, either with a good value if the error detection part managed to save the correct value, or with old/default value. If timeout data reception or acknowledgement data emission is missed, logging routines could have saved the exchanged value and the recovery restores communication, or it repeats the communication call.

5. Instrumentation

Two types of software instrumentation are considered (Figure 6): hooks and basic services. Hooks are the means to tie up defense software to functional software, and to insert code. The possibility to generate hooks automatically is an advantage regarding development cost. Basic services play the role of software sensors and actuators. The availability and authorization of their use enables to limit intrusiveness especially to get information. All fault-tolerance

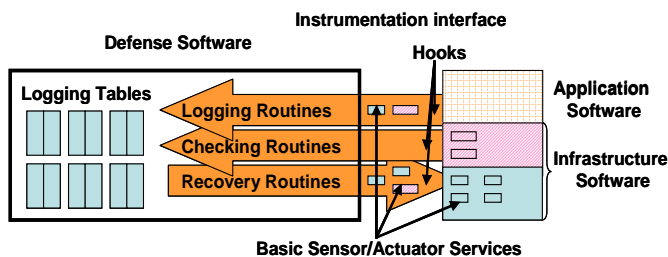
intelligence and control is contained in defense software described before. Now, instrumentation, as tools, intends to be generic, flexible and reusable.

5.1. Hooks

In C programming, hooks are entry points, with empty routines, located at selected places in the program. They are commonly used as debugging breaking points or exception treatments triggering. In the AUTOSAR OS specification [2], some hook functions are defined and implemented by the user. The operating system invokes them at specified times, such as tasks context switch, startup, shutdown, or detected errors.

These hooks are very convenient for “logging”, “checking” or “recovery” routines belonging to the defense software. The insertion of a hook, at a selected place in the source code, is related to both a place in the architecture of the software system and a moment of execution at runtime.

Figure 6. Instrumentation organization.



Hooks must be placed on critical data and control flow. Considering the 4 types of traces (section 4.1), recording the execution from OS and application viewpoints, can be done with hooks set at the beginning and at the end of execution of critical tasks and application-level functions. Hooks are also set at the corresponding critical services calls to capture the control and data event traces.

The instant of execution to trigger the “checking routine” and thus the location of the hook, is crucial. For the “logging routine”, the location of the hook is set where information is easier to capture.

Recovery routines are essentially triggered by checking routines, immediately after error detection. Nevertheless, an application can specify that after error detection the recovery has to be delayed to the end of task execution for example. In this case, a hook at the end of the considered task contains the recovery routine, which is activated only if quoted by the corresponding checking routine.

Several implementations of automatically generated hooks can be considered, especially “at” service calls. Hooks can be inserted just before or after the call instruction. Another solution is to add them within the service routine, at the beginning or the end. The difference is important with system calls, if the system

supports the separation between user and supervisor mode. In the first case, the hook routine runs in user mode, whereas in the second case, it runs in kernel mode and has access to more information if needed (task priority, etc.).

Another implementation issue is the use of parameters or not at the hook interface. It is an alternative to the use of sensor services. For example, after a write-service, the data value may be collect as an input parameter of a hook, instead of using a read-service to get the value.

5.2. Basic sensor services

Types of information that contribute to describe execution, control event and data event traces are:

- **Exchanged parameters (what):** return notification, activated task, set event, activated alarm, exchanged data, etc.
- **Current execution context (where):** application-level function identifier, task identifiers (task state, priority, etc. if needed) or interrupt routine identifier, current mode, etc.
- **Timestamp (when):** e.g. counter register value

Executive support should give the possibility to the user to get this information through an observation interface. For example, via OSEK-VDX operating system standard interface [11, 2, 12], some information is reachable: the running task identifier (“GetTaskID”), the task state (“GetTaskState”), the current state of event mask of a task (“GetEvent”), alarm characteristics (“GetAlarmBase”, “GetAlarm”), and the current mode (“GetActiveApplicationMode”). Autosar OS that can be considered as an extension of OSEK, has additional standardized interfaces: “GetISRID” to get the identifier of interrupt routines, “GetApplicationID” to get the identifier of a sort of partition (if the OS uses memory protection), and information about predefined scheduling tables (“GetScheduleTableStatus”, “GetCounterValue”, “GetElapsedCounterValue”).

The observation interface of the Autosar operating system is rich enough, if the user does not need to check the task priority. What is missing, at higher level, is essentially an identifier for application-level functions, which is added manually otherwise.

5.3. Basic actuator services

Basic actuator services can be defined independently from a particular implementation, even if the reference studied architecture is that of Autosar standard. Practically, functional infrastructural services are used (Table 2), even if they are not designed to perform recovery. Ideally, a specialized recovery interface should be provided by the infrastructure and should be well controlled by the user. Referring to the

model of execution in section 2, actuator services can be structured into control actions and data actions.

Control flow actuators.

At the operating system level, actions on control flow concern the life cycle of tasks and can be classified into 3 categories:

- **End of task execution:** the idea is to terminate the erroneous current treatment.
- **Start of task execution:** the objective is for example to launch a degraded task if switch to degraded mode is decided; or to launch the expected task after error detection on sequence of execution; or else to launch again the same task from the beginning to re-execute the same treatment with right entries. The activation of a task may be synchronous or asynchronous.
- **Suspension of task execution:** the idea is to temporarily stop the current execution, to allow the execution of another action/task.

Table 2. Recovery actions with AUTOSAR.

Recovery action	Useful Autosar services
End of task execution	TerminateApplication, TerminateTask, ChainTask, CancelAlarm
Start of task execution	ActivateTask, ChainTask, RestartTask (with TerminateApplication), SetEvent, SetRelAlarm, SetAbsAlarm
Hang of task execution	- (difficult with a static priority based scheduling)
Production of data	Rte_Write, Rte_IWrite, Rte_IrvWrite, Rte_IrvIwrite, Rte_Send
Consumption of data	Rte_Read, Rte_IRead, Rte_IrvRead, Rte_IrvRead, Rte_Receive
Renewal of data request	Rte_Send, Rte_Call
Inhibition of data	- (no direct means)

Data flow actuators.

At the communication level, actions on *data flow* relate to actions on data value and on data timing occurrence:

- **Production of correct or degraded data:** the recovery strategy overwrites the preceding erroneous data, by the right one.
- **Consumption of correct or degraded data:** data consumption instruction is called another time to get the correct value which is has been updated by the recovery strategy.
- **Renewal of data request:** data production or consumption instruction is called another time, when timeout reception or acknowledgement of emission is missed.
- **Inhibition or delay of data:** when invalid or untimely data is received, the recovery strategy acts

as a filter, to transmit only right data to the application.

The following section refines the description of defense software and instrumentation in the context of memory protection with kernel and user separation of modes and address space.

5.4. Protection of defense software

The protection of defense software is principally a matter of economical constraints. The more measures are taken to improve defense software, the more it is expensive. A “low-cost” solution is required, although all the proposed fault tolerance strategy relies on the robustness of defense software. The only design property of defense software that satisfies both opposite requirements is: the complexity of the defense software is considered much lower to that of the functional software by construction. Concerning enhanced validation process and hardware protection, it will depend case by case on available resources that are given to particular projects.

Enhanced validation process.

A rigorous development process, including verification methods, has to be performed. We use fault injection techniques (Section 3.5) to measure fault-tolerance coverage, and to detect remaining software errors of defense software.

When defense software is based on safety assertions that have a complex behavior (check of transitions that implies many data and control elements), the use of a formal language to implement these routines is to be considerate. Again, some automotive projects may not take this option for culture or economical reasons. In our work, we use the C language, respecting MISRA coding rules [12].

Hardware protection.

To strictly follow the principle of separation of functional and safety concerns promoted by the reflective approach, both software part should be spatially and timely separated. Taking the example of Elektra railway system [9], three processors operate functional services and three other processors supervise the functional part, in parallel. So many resources are still unaffordable in the automotive world. Instead, software redundant information logging is realistic, in the proposed architecture, if other resources and timing constraints are respected.

Simple separation of functional and defense software can be done by the use of hardware memory protection. Considering this particular context, hooks can be implemented in user space, for convenience of existing automatic code generation of hooks. The logging tables are the most critical data, so they must be stored in protected address space, separated from

functional part. Logging, checking and recovery routines, with the software sensor and actuator they contain, have to access the logging tables by reading or writing, so they also must be trustable. As these routines are called within hooks in user mode, that requires a switch from user mode to kernel mode.

6. Early implementation issues

We have developed several AUTOSAR software platforms, both on a virtual processor running on an UNIX machine and on a real embedded evaluation board. We use a Freescale evaluation board S12XEP100TM 16 bit microcontroller, with memory protection unit, and another S12XDP512TM, without memory protection. Our development environment is CodeWarriorTM from Freescale.

The AUTOSAR RTE is automatically generated by a software tool from Vector (Microsar RTE, DaVinci DeveloperTM 2.2). We worked both on several application components we synthesized, and on serial automotive software products we adapted to the AUTOSAR context. The safety properties we take as inputs are derived from real automotive requirements.

We use Trampoline [13], an open source operating system from IRCCYN, compliant to AUTOSAR OS.

Our current experiments show the feasibility of the approach to improve robustness on prototypes. We have compared protected and non-protected applications with similar hardware, by carrying out verification testing, using controlled fault injections that cause USE (Unwanted System Events). Protected applications perform fault tolerance of their failure model. However, the evaluation of robustness should be completed by comparison with other fault-tolerance solutions.

7. Conclusion

The automotive industry is facing increasing complexity of embedded software, error propagation and the need to meet robustness challenges, in spite of stringent economical constraints. As a representative context of tomorrow's automotive software, we chose to deal with the two emergent standards: AUTOSAR for modular multilayered software architecture, and ISO26262 about safety concerns.

The work reported in this paper shows an approach to develop customizable defense software, externally to the target system. The proposed fault-tolerant architecture is based on the classical separation of the functional implementation and that of the safety functions, using the interfaces (entry points) defined by AUTOSAR.

This approach is very attractive for the automotive industry since it enables to tailor defense mechanisms according to the needs on a case-by-case basis.

Feasibility study has been carried out on early implementations of synthetic AUTOSAR applications. Current work exemplifies in deep error detection and recovery mechanisms and focus on fault injection to evaluate the efficiency of the approach.

References

- [1] C. Lopes, W. Hursch. "Separation of Concerns". Technical Report, College of Computer Science, Northeastern University, Boston, USA, Feb 1995.
- [2] AUTomotive Open Standard ARchitecture, <http://www.autosar.org>
- [3] ISO/CD 26262-6, "Road vehicles, Functional safety, Part 6: Product development: software level", 2008.
- [4] R. Chillarege, IS. Bhandari, JK. Chaar, MJ. Halliday, DS. Moebus, BK. Ray, and MY. Wong, "Orthogonal defect classification-a concept for in-process measurements", *IEEE Trans. Softw. Eng.*, 18(11):943-956, 1992.
- [5] P. Maes, "Concepts and Experiments in Computational Reflection". *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Orlando, Florida, pp. 147-155, 1987.
- [6] J. Voas, "A Defensive Approach to Certifying COTS Software", *Reliable Software Technologies Corporation*, Technical Report: RSTR-002-97-002.01, 1997.
- [7] M. Rodriguez, J.C. Fabre, J. Arlat, "Wrapping real-time systems from temporal logic specifications". *European Dependable Computing Conference (EDCC-4, 2002)*, Toulouse (F), pp. 253-270, 2002.
- [8] F. Taiani, J.C. Fabre, M.O. Killijian, "Towards Implementing Multi-Layer Reflection for Fault-Tolerance". *IEEE International Conference on Dependable Systems and Networks (DSN'2003)*, San Francisco (CA, USA), pp. 435-444, 2003.
- [9] H. Kantz, C. Koza, "The ELEKTRA railway Signaling-System: Field Experience with an Actively Replicated System with Diversity". Alcatel Austria AG., Vienna, Austria, 1995.
- [10] P. Traverse, I. Lacaze, J. Souyris, "Airbus fly-by-wire: A total approach to dependability", 2004.
- [11] "OSEK/VDX Operating system". *Technical report*, 2005.
- [12] The Motor Industry Software Reliability Association, <http://www.misra.org.uk>
- [13] J.L. Béchenec, M. Briday, S. Faucou, Y. Trinquet, "Trampoline : An Open Source Implementation of the OSEK/VDX RTOS Specification", *IEEE Int. Conf. on Emerging Technologies & Factory Automation (ETFA'2006)*, Prague, Czech Republic. pp. 62--69 (2006) – see : <http://trampoline.rts-software.org> –