Regular paper submitted to **ETFA 2009** **Approx. word count:** 5500
The material contained in this paper has been cleared through the authors' affiliations

Robustness of modular multi-layered software in the automotive domain: a wrapping-based approach

Caroline Lu [1, 2, 3]
[1] Renault Technocentre,
1 Avenue du Golf
78288 Guyancourt Cedex,
France
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 ; LAAS ;
F-31077 Toulouse, France
{jean-charles.fabre, marco.killijian}@laas.fr

Contact author: Jean-Charles Fabre {Jean-Charles.Fabre@laas.fr}

Keywords: Wrappers, Reflective computing, On-line checking, Defense software, Autosar software architecture

Abstract

New automotive modular multi-layered software organization particularly favors use and interoperability of Components-Off-The-Shelf. However, the integration of software components is error-prone, if their coordination is not rigorously controlled. The risk of failure is increased with the possibility to multiplex software components with heterogeneous levels of criticality, observability. Most of dependability mechanisms, today, address locally errors within each component or report them to further diagnosis services. Instead, we consider a global wrapping-based approach to deal with multilevel properties to be checked on the complete multilayered system at runtime. In this paper, we introduce a framework to design robust software, from analysis to implementation issues, and we illustrate the methodology on simple case study.

1. Introduction

Initially, in automotive embedded systems, software was mainly standalone, controlling a single functionality. Today, as increasing number of vehicle functions is controlled by software, we can find from hundreds of megabytes up to several gigabytes of code in a car. To cope with this evolution, automotive software architecture tends to follow principles of componentization and superposition of software layers. Better structuring aims to improve maintainability, obsolescence management, and the use of COTS (*Components-off-the-Shelf*).

Complexity, multiple sources of faults, and cost reduction stand in the way of software robustness. Actually, dependability requirements become higher with the introduction of some safety critical applications, such as X-by-Wire. Another challenge, but classical problem in terms of reliability, is the robust integration in a whole system, of software components having different levels of criticality, observability and controllability. To answer these dependability constraints, most of automotive protection mechanisms address errors locally, within each software component, or report them to further diagnostic services.

However, we consider that addressing some real-time problems requires a global approach able to manage information through the fences of modularity and layers of abstraction at the same time, having the complete system under supervision. In this paper, we discuss how to apply software wrapping to improve robustness of automotive modular multilayered software. But, more than an additional technique dealing with robustness, we intend to build a methodology to design robust software. In fact, classical mechanisms are used and combined but a new way of thinking is introduced.

The contribution of this paper is to show how "principled defense software" can be developed, as wrapping software devoted to fault tolerance. The new trend of system design in application automotive relies on COTS components and extensive use of glue code generators and tools provided by external software houses. This new context may introduce multiple software fault sources (e.g. component integration faults, bugs in code generation tools), not forgetting classical physical faults (hardware defects, ECM, etc.). From a system integrator viewpoint, i.e. a car manufacturer, this is a real problem for the next future.

The paper is organized as follow. Section 2 describes the problem statement and the objectives of our work. Section 3 introduces the proposed approach principles. In section 4, we briefly present AUTOSAR [1] that is our reference architecture. Section 5 defines how to express application-specific safety properties, which represent selective knowledge of the system to perform fault-tolerance. Section 6 describes the core issues of the defense software implementation. We illustrate the methodology by an early case study in Section 7. Related works is discussed in Section 8 before concluding in Section 9.

2. Problem statement and objectives

The Autosar standard [1] (see. Section 4) promotes a multi-layered software architecture for automotive applications, basically composed of an executive software layer and a thin middleware layer, supporting the execution of applications. Although this architecture is slightly different from conventional middleware-based systems, it introduces similar problems from a dependability viewpoint. Application takes advantage but also depends on generic or specific services that belong to the underlying layers (middleware and OS services). This way, the architecture introduces potentially error propagation channels. In addition, the observed failure modes of the system are more severe when the initial error propagates [2]. In a multi-layered architecture, the correct behavior of an application not only depends on the correctness of underlying services. It has been shown in [3] that the correct behavior of an application also relies on *multilevel properties*. As an example, determinism depends on various decisions taken at various levels of the software architecture.

The objective of the work is to develop a *lightweight* solution to the above problem due to resource constraints in automotive applications. To achieve this, our robustness approach consists in developing a *defense software* as a set of *wrappers* checking multilevel properties at runtime [4]. The defense software corresponds to error detection and recovery mechanisms and can be customized according to the need. The checks may only address a subset of the applications running on the same executive support. We can thus choose to target only selected critical applications.

The wrappers check the correctness of the application that depends on the behavior of the middleware (communication channels between application components) and OS functions (task management and scheduling) despite accidental and design faults that could impair both the control flow and the data flow of the application. They also trigger recovery actions.

3. Technical issues & framework principles

Software wrappers sit around a software component and limit its behavior in desirable ways. As far as dependability is concerned, the wrappers are devoted to error confinement and recovery actions, for instance to put the system in a safe state or to trigger some degraded mode of operation. To be efficient the wrappers must intercept component interaction but also observe and control internal objects. This implies that *software sensors and actuators* must be added to software components. Software *hooks* are implemented, in order to enable fault detection and recovery actions, to be added to the system, as external software with limited intrusiveness. Our design method is based on behavioral reflection [5] providing separation of concerns between applications and dependability monitoring services. The software development, the maintenance and the evolution of each part are then simplified.

The defense software is thus organized in two parts: *Error Detection* (EDMs) and *Error Recovery Mechanisms* (ERMs). The EDMs are composed of several algorithms, each of them corresponding to a given multilevel property to be checked. When errors are detected, EDMs send reports to ERMs (error filtering or recovery strategies that depend on specified degraded modes). The interface between functional and non-functional software is made of software sensors and actuators. Their location in the software architecture depends on the design and the algorithms implemented in EDMs and ERMs. Sensors log information at runtime and trigger EDMs whenever necessary. Actuators are recovery functions that are monitored by ERMs.

Our proposed framework (cf. Figure 1) based on behavioral reflection [5] generalizes the notion of hooks that can be found in the automotive basic software (namely the real-time kernel, AUTOSAR OS in this context). A meta-interface corresponds to the set of software sensors and actuators that is necessary and sufficient to check a given multilevel property at runtime. This specific set of sensors and actuators corresponds to the notion of reflective footprint associated to a given system property [3].

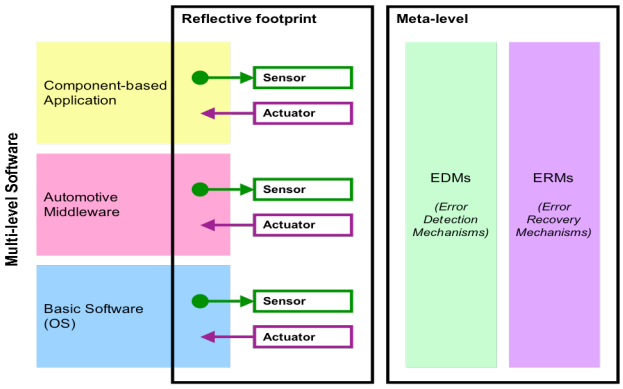

Fig. 1. Wrapping framework principle

To reach this aim, our approach follows the steps given below:

1) Analysis of the target system and its provided services;
2) Selection of the faults that are considered at various levels of abstraction;
3) Definition of the dependability properties that must be verified on-line;
4) Definition of the error detection and recovery mechanisms;
5) Definition and implementation of the corresponding observation and control mechanisms;
6) Implementation of the fault tolerance mechanisms based on software sensors and actuators;
7) Evaluation by fault injection of the coverage of such fault tolerance mechanisms.

The work reported in this paper does not dive into the details of each step, but motivates and illustrates to some extent the global approach that we propose step-by-step.

4. AUTOSAR software architecture

AUTomotive *Open Standard ARchitecture* [1] is a development partnership started in 2002 supported by major automotive manufacturers, suppliers and tool developers. Its goal is to define a common automotive software development environment, which enhances reuse on different hardware platforms. The Autosar methodology relies on description, static configuration, and automatic code generation tools, and it favors interoperability of tools.

The Autosar reference model is composed of three principal software layers:

Application Layer. At the application layer, software programs are divided into *"runnables"*, similar to functions (in C programming), or methods (in object-oriented programming). *Runnables* are the schedulable application entities that the system integrator maps into tasks managed by the operating system. During the integration phase, runnables can be grouped within a task, depending on criterions like workflow, periodicity, data consistency, etc… For example, in order to avoid concurrent writing or reading access on a given data, the integrator can force an execution sequence of the concurrent runnables in the same task.

Runtime environment. The Autosar Runtime environment is a software part, which provides interfaces enabling the exchange of signals between runnables, and it is responsible for data consistency. In the Autosar methodology, the RTE is automatically generated from the configuration of software components and basic software. Due to its intermediate position, the RTE also plays the role of a glue code: it uses OS objects such as tasks, resources, events to provide its own functionalities to the application layer.

Basic Software Layer. The basic software layer provides the services that are necessary to the execution of the application. Inside this layer, there are two main components from the RTE point of view: AUTOSAR OS that manages task processing, memory and inter-process communication; AUTOSAR COM, which deals with message exchange management. This layer runs on top of one ECU.

The analysis of the target architecture and services correspond to ***step 1*** in the design process of the defense software.

5. Dependability properties definition

5.1. Fault assumptions – application level

We assume a system may fail in operation due to either physical faults or residual bugs. From the user viewpoint, these faults all result in transient or permanent failures on the functional data and control flow. The effect, then, can be evaluated depending on levels of potential threat to people or undesirable event to the customer.

We structure the failure model on application software into two parts: data and control flow (Table 1). Better or more detailed modeling is out of the scope of this paper. The user can arbitrarily identify a concrete failure regarding data flow, control flow failure or both, depending on his major concern, and the target application requirements.

We define three main types of critical *Control Flow* failures:

- The first one, in table 1, targets *"triggering events"* of execution. These events may be ruled by timing constraints, data reception, emission or other mixed pre- or post-conditions. At executive level, for instance, an asynchronous functional treatment can be implemented with an explicit *"task activation"* call within another treatment. The *"task activation"* call is a *"triggering event". A failure may occur if an event is not expected or if it is missed*.
- Another control failure type is a *"wrong sequence of execution"*. We make a slight difference between unexpected sequences (that makes sense only if reference functional sequences are defined) and forbidden sequence (that has been specified). For instance, *a wrong sequence failure may be due to a wrong mapping of runnables to tasks.*
- A third category of control failure type affect the *"time of execution"* of a treatment unit. It may be a consequence of a wrong computation but also all other failure types of Table 1. Complex timing (timing evolution, periodicity, etc.) requirements may be missed. Non expected early delivery may also corrupt both data treatments and dependencies of execution. *A deadline miss is conventional example of a timing failure.*

Concerning critical *Data Flow* failures, we distinguish value and timing defects. Data include global variables, data queues, or exchanged messages, as inputs or outputs of software modules. A value may be faulty for two reasons, either it is a valid value but wrong in the computing context or the value is out of range. We may also have more complex requirements on the values of a set of data. A Byzantine failure is an example of inconsistency between several data.

Critical Flow	Type of failure		Basic Safety Assertion to be verified at appropriate Execution Control Points
Control Flow	Triggering Event	Non expected	Only critical triggering events with predefined execution context constraints or timing constraints must be processed.
		Invalid/ forbidden/ missed	Critical sent triggering events with predefined execution context constraints or timing constraints must not be lost.
	Functional sequence of execution	Non expected	Critical sequences of execution are referenced and other are non-expected.
		Invalid/ forbidden	Critical invalid or forbidden sequences of execution are referenced and must not happen.
	Time of execution	Too late/ missed deadline	Critical scheduled units must meet their timeout constraints.
		Too early	Critical sent triggering events and data with predefined execution context constraints or timing constraints must not be lost.
		Bad evolution /missed periodicity	Critical scheduled units must meet their timing constraints.
Data Flow	Exchanged data value(s)	Invalid/ out of range / forbidden	Critical data must meet their value constraints (range, forbidden values, produced value equal to consumed value).
		Wrong value (within range)	Critical data must meet their value constraints (evolution, dependencies).
		Byzantine (inconsistent)	Critical input data for several consumers must be the same for all consumers.
	Time of communication exchange	Too late	Critical data must meet their timeout constraints.
		Too early	Critical data must be produced and logged to prevent late consumption.
		Missed synchronization	Several critical data must be all produced before being consumed.

Table 1. Failure Model and related Basic Safety Assertion

If functional timing constraints are explicitly given on data, we classify such constraints as a requirement on time of communication exchange of data, instead of time of execution. We have in the same way, the notion of too late and too early delivery of data. When we deal with data items produced by several entities and consumed by only one entity, we can be faced to synchronization problems (e.g. all data items must be available before consumer starts processing).

The application-specific failure model is ***step 2*** in the design process of the defense software.

5.2. Multilevel architecture related properties

The generic failure model presented above is concerned only with faults affecting the application software level. Once a sub-model is identified and extracted for a given case study, we need to derive, from these high-level concerns, implementation issues on the complete software architecture down to low-level layers. Therefore, we consider that the global dependability of the application is a "multi-level" problem, making the bridge between applications and software infrastructure.

To better understand how multilevel properties are verified at runtime, we need to introduce two notions:

1) *Execution control point (i.e. a verification point)*: this notion indicates where/when the verification of the property must be performed in the execution flow. An analogy can be made with the notion of *joinpoint* in AOP, the *advice* being the implementation of the assertion verification. The control point can be the first instruction of a runnable, or the termination of a task (similarly to *before* and *after* notions in AOP);
2) *Execution Context*: this notion relates to the context of execution in which the verification must be performed, that includes the identification of runnable in progress, the task where the runnable is activated or an interrupt routine (an ISR, Interrupt Service Routine, that is scheduled as a task in practice). Some timing information can be attached to an entity execution, such as a deadline.

For example, in table 1, a *non-expected triggering event* means concretely that when an additional untimely event happened at runtime, the system must to detect the case and inhibit it. The execution context or other characteristics of events have to be identified in order to discriminate nominal events from others, possibly unexpected. The safety assertion *(or "multilevel property")* we can draw from this failure example is:

*At **Execution Control Point**, only specified critical triggering events with specified **Execution Context** including timing constraints must be processed".*

Using AUTOSAR OS implementation, a triggering event may be the activation of a task (through the *"ActivateTask"* service), the triggering of an Interrupt Service Routine, or an OS event emission (through the *"SetEvent"* service).

- The *Execution Control Point* can be the beginning of execution or a waiting point within a task, or the instant of reception of the triggering event.
- The *Execution Context* of the triggering event means the identifiers of the current task or an Interrupt Service Routine.

These notions are of interest to trace, through the system layers, the semantic information that is required to perform the verification of a multilevel property.

In the same way, we can express other basic assertions of table 1. The *"scheduled units*", mean either tasks at OS level, or *runnables* that are schedulable functions used by application designers at high-level. Functional specifications address first *Runnables* that can be mapped to separate *Tasks* or not. Expressing the safety assertion with *Runnables* or *Tasks* is arbitrary; it depends on the use case and optimization issues. This way, *"sequence of execution"* means a sequence of one or the other type of scheduled entities, either runnable or tasks. The complexity of the mapping of runnables to tasks, the use of external tools to generate the RTE, are possible sources of faults during the development process that can impair the expected *"sequence of execution"*.

The definition of multilevel properties is ***step 3*** in the design process of the defense software.

6. Defense software implementation

6.1. Error detection and recovery strategies

For a given failure, the error detection algorithm is the implementation of the multilevel property, in other words the detection assertion. The detection algorithm uses information captured at various layers of the systems; the notion of hook gives a practical mean to implement the reflective footprint required to verify some assertion. The invocation of the error detection algorithm depends on the multilevel property and can be done within a hook routine. The algorithm should also check if the detected error is sporadic or permanent.

Error recovery depends on degraded modes of the application. However, degraded modes of automotive applications are generally very sophisticated at the system/application level. On the contrary, at the infrastructure level, recovery actions are usually basic: reset, inhibition of applicative functions, termination and restart of tasks, recovery of messages timeout reception or non-acknowledgement emission. The idea is to use infrastructure recovery controlled by application level consideration, to take advantage of both approaches. Practically, recovery is implemented by low-level service calls that are performed either within hooks, or within safety tasks. For example, when an error is detected on the sequence of execution, the software actuators, which are used to modify the flow of execution, are the system call to terminate or to activate tasks.

The use of standard hooks or additional hooks is necessary to implement the defense software. In summary the hooks are used (i) to capture information, (ii) to trigger the verification of the assertion and (iii) to trigger the recovery actions.

The definition of error detection and recovery strategies is ***step 4*** in the design process of the defense software.

6.2. Software instrumentation

In C programming, hooks are entry points, located at selected places in the program. They are commonly used as debugging breaking points or exception treatments triggering. The idea is to use hooks as software instrumentation, in order to collect and store the information that is required to compute error detection algorithms and to control recovery actions. The hooks are used to (i) observe the computation in progress, (ii) verify the assertions at *Execution Control Points* according to a pre-defined *Execution Context*, and (iii) finally execute recovery actions.

Using hook routines, the minimum information that we log is:

- Time, which is needed to make timestamps;
- OS object identifiers that can be obtained by using OS services;
- *Runnables* identifiers (which do not exit in the AUTOSAR specification);
- Exchanged data, between runnables for instance.

The precise location of the hook depends the information to get and on implementation optimization. Actually, we need hooks around service calls that perform critical control or data flow actions. Putting hooks *before* or *after* an interface can be easily realized by automatic code generation.

In the AUTOSAR context, the *RTE* [1] implements a series of hook functions with empty routines that are invoked automatically by the generated *RTE* when selected events occur (e.g. several OS service calls, *runnables* invocation, etc.). We use these hooks when they exist in the AUTOSAR RTE specification, and add the missing ones depending on our use case. For instance, the OS primitive *ChainTask* (triggering a task from another task) has no hook attached in the standard. For detection purposes first, we introduced such hook that is invoked when the primitive *ChainTask* is called. Actually, this hook can be used to notify the termination and activation of tasks. On the other hand, this hook may be also be of high interest for recovery, e.g the *ChainTask* call can be inhibited when an error is detected, or a safety task can be activated instead.

The definition of software instrumentation is ***step 5*** in the design process of the defense software.

7. Simple case study

We have developed several AUTOSAR software platforms, both on a virtual processor running on UNIX and on a real embedded evaluation board. We experienced the adaptation of serial automotive software products to the AUTOSAR context, adding our multilevel reflective mechanisms. We showed the feasibility of our approach and obtained promising robustness improvement on the studied prototypes by testing and early fault injection campaigns.

In this paper, we briefly illustrate the steps of our approach with a simple synthetic case study, from step 2 to 6 (step 1 correspond to Section 4). Due to space limitation, step 7 is out of the scope of this paper that only intends to highlight the methodology.

7.1. Platform description

We have developed an embedded system, implemented on a Freescale evaluation board Star12XE™ 16 bit microcontroller, with memory protection unit. We use Trampoline [6] an open source operating system from IRCCYN compliant to AUTOSAR OS (see also http://trampoline.rts-software.org). Our development environment is CodeWarrior™ from Freescale. The RTE is automatically generated by an AUTOSAR tool from Vector (DaVinci Developer™ 2.2). We have synthesized several application software components and multiplexed them on the AUTOSAR infrastructure. We develop our "multilevel reflective framework" on selected critical control flow that realizes simplified part of torque control arbitration.

Only the critical control and data flow is described here (Figure 2). The case study application is composed of three tasks that manage 6 runnables. (Real names of functions and data have been omitted).

- The task *Task_Driver*, including runnables *Runnable1* and *Runnable2* is activated by an alarm every 16 ms, with a priority equal to 3.
- The task *Task_ESP*, including runnables *Runnable3* and *Runnable4* is activated by an alarm every 4 ms, with a priority equal to 5.
- The task *Task_Torque*, including runnables *Runnable5* and *Runnable6*, is activated by an alarm every 16 ms, with a priority equal to 1.

Runnables communicate through global variables (produce and consume data), as shown in Figure 2.

7.2. Failure model & safety assertions

We illustrate here 2 types of failures to be avoided: the first one about control flow, and the second one about data flow. Extracts from table 1 are given to show the corresponding generic failures and assertions. Then they are specialized to the case study.

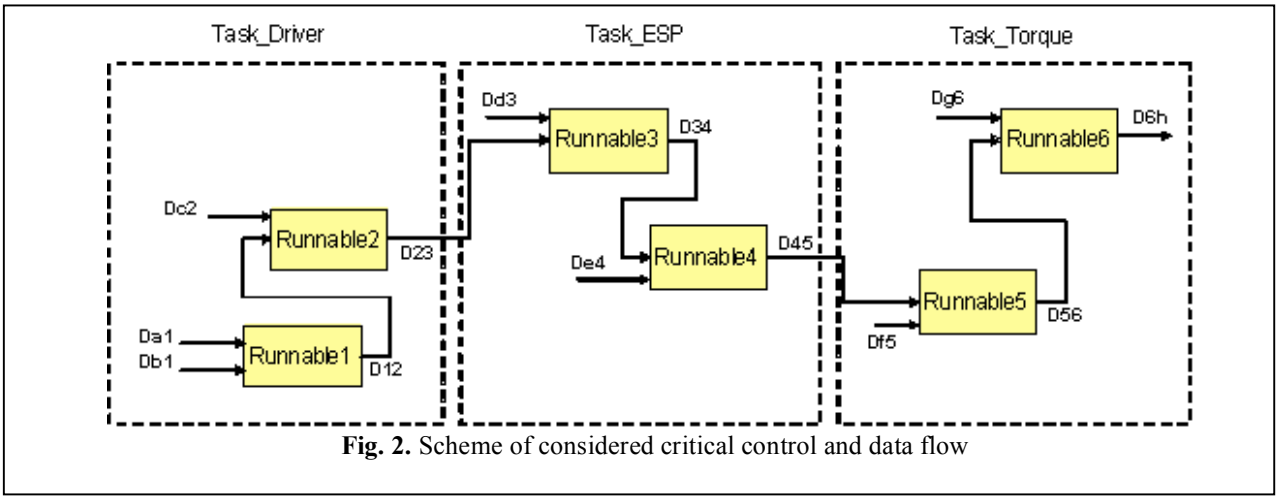

Fig. 2. Scheme of considered critical control and data flow

Failure & assertion n°1.

Functional sequence of Execution	Non expected event
*At **Execution Control Point,*** critical sequences of execution are referenced and other are non-expected.	

Table 2. Extract of Table 1 (Control Flow)

The expected periodic atomic sequence pattern are S1= {Runnable1, Runnable2}, S2= {Runnable3, Runnable4}, S3= {Runnable5, Runnable6}. If S1 or S3 atomic sequence is broken, it has to be re-executed. If it is S2, degraded values are taken. Before S3, if S1 is older than 4 executions of S2, S1 must be re-executed before S3.

Failure & assertion n°2.

Time of communication exchange	Missed synchronization
*At **Execution Control Point***, several critical data must be all produced before being consumed*".*	

Table 3. Extract of Table 1 (Data Flow)

Each runnable must have its inputs available before it can run. If the produced data is inconsistent with the consumed data, the logged data is taken as an input. If the produced data is invalid, a default value is taken.

7.3. Defense software implementation

Error Detection. The algorithms are simply the implementation of assertions 1 and 2. For the first assertion, the atomicity of sequences S1, S2 and S3 are checked at the beginning of each critical runnables (*Execution Control Point)*. The verification of precedence of S3 is done before the execution of *Task_Torque*. For the second assertion about data flow, data consistency is checked before each critical runnables.

Error Recovery. When error is detected, recovery actions are immediately taken. The re-execution of runnables is implemented by the "*ChainTask*" OS service, that terminates the current task and activates the requested one. Concerning data flow, the decision to take default values, is realized by a "*Rte_IWrite*" service of AUTOSAR RTE. Default values are in our case the last correct value.

Hooks. Information needs to be caught at runtime before and after each critical runnable and task runs, which gives the location where some hooks must be added. Hooks around runnables log the runnable ID, task ID and timestamp, which are updated at runtime. These hooks type is defined in the RTE specification, for debugging issues. Hooks around tasks log task ID and timestamp. For this purpose, we use OS hooks routines: "*PreTaskHook*" and "*PostTaskHook*", that occur at each context task switch.

About data flow, we need to use hooks around "*Rte_IWrite*" and "*Rte_IRead*" service calls of the RTE. They already exist in the standard. Within these hook routines, we store the exchanged data and the execution context (*Runnable ID, task ID, timestamp*).

Figure 3 illustrates, for assertion n°1, in pseudo-code, the structure of functional software (*RTE.c*), defense software (*DefenseSW.c*) and instrumentation (*Hooks*). Bold instructions in *RTE.c* are original application. Those in *DefenseSW.c* are the existing services of the software infrastructure we use to perform fault tolerance.

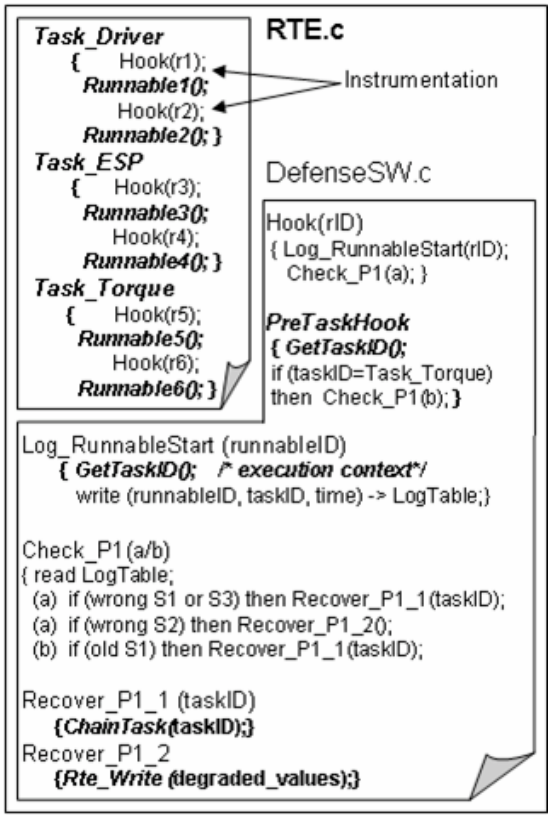

Fig. 3. Defense software for assertion n°1

8. Related works and comments

Reflective computing provides separation of concerns that is of interest for managing non-functional mechanisms, e.g. implementing defense software as error detection and recovery wrappers. Our work is based on previous works: (i) wrapping for improving robustness was carried out on real-time microkernels [2], wrapping multilayered software using reflection for fault-tolerance was described in [3].

The proposed approach is a lightweight alternative wrt other industrial solutions to improve system robustness and safety in railways and aeronautics applications. In the electronic interlocking system Elektra [7] a two-channel-approach with specification diversity is used for detecting software design faults and providing safety (*Safety bag*). 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 [8], as a corner stone for heavy redundant architectures. Clearly such heavy architectural solutions are not economically viable for the automotive industry for the time being, due to strong constraints in terms of resources.

The approach cannot compete with such redundant architecture with respect to crash/hang faults, but can be efficient to deal with residual software faults, integration faults, transient hardware faults. The underlying assumption is that the defense software is more reliable than the target system software (possibly including COTS) because a strong validation process can used during its development by the system integrator. Ideally, the defense software should be isolated from the target system software in a separate address space (memory protection). This decision also depends on performance tradeoffs.

The proposed work is consistent with the forthcoming ISO26262 [9] standard that defines graded safety levels called ASIL (*Automotive Safety Integrity Level*). They are graded from A level to D level with a respectively increasing criticality. We believe that our approach and framework provide a suitable technical solution to improve the robustness of applications at various safety levels running on the same platform.

9. Conclusion

Robustness of embedded software is a crucial issue due to the increasing complexity software functions and their supporting runtime environment. The multiplexing of software components on a given hardware platform, the reuse of existing software components, the flexibility of system configurations and the adaptation to user needs are essential. To this aim, complex multilayer software architectures have been defined. Improving the robustness of application in such software architecture is a real challenge. The automotive industry is very active to meet this challenge through standardization activities (AUTOSAR, ISO26262). Our approach relies on multilevel reflection concepts. The work reported in this paper shows that this approach is promising and that the proposed framework can be of high interest to check runtime properties of complex applications based on a multilayered system.

The initial case study showed the feasibility of the approach and its consistency with the automotive executive layers concepts (runnables, RTE/OS and hooks). This simple implementation already shows interesting characteristics of the multilevel RTE (this is where most of hooks can be placed even for application or operating system purposes). Additional hooks that we define and do not exist in existing specifications may be proposed to AUTOSAR consortium. The implementation of the defense software is thus consistent with the standards and flexible.

The main benefit of this approach is clearly that the defense software can be developed as an external software customized on a case-by-case basis for an embedded application and its critical safety properties.

Current work focuses on fault injection to evaluate the approach efficiency on more complex case studies.

10. References

[1] AUTomotive Open Standard ARchitecture, http://www.autosar.org

[2] Rodriguez, M., Fabre, J.C., Arlat, J.: Wrapping real-time systems from temporal logic specifications. *European Dependable Computing Conference* (EDCC-4, 2002), Toulouse (F), pp. 253--270 (2002).

[3] Taiani, F., Fabre, J.C., Killijian, M.O.: Towards Implementing Multi-Layer Reflection for Fault-Tolerance. *IEEE Int. Conf on Dependable Systems and Networks* (DSN'2003), San Francisco (CA, USA), pp. 435--444 (2003).

[4] Voas, J.: A Defensive Approach to Certifying COTS Software. Reliable Software Technologies Corporation, *Technical Report: RSTR-002-97-002.01* (1997).

[5] Maes, P.: Concepts and Experiments in Computational Reflection. In *Proc of Int. Conf. on Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA), Orlando, Florida. pp. 147--155 (1987).

[6] Béchennec, J.L., Briday, M., Faucou, S., Trinquet, Y.: 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).

[7] Kantz, H., Koza, C.: The ELEKTRA railway Signaling-System: Field Experience with an Actively Replicated System with Diversity. In *Proc of the Int. Conf. on Fault Tolerant Systems* (FTCS 1995), Pasadena, USA, pp 463--471(1995).

[8] P. Traverse, I. Lacaze, J. Souyris, "Airbus Fly-by-Wire: A Total Approach to Dependability", in *Proc. 18h IFIP World Computer Congress,* Toulouse (F), pp.191--212, Kluwer Academic Publishers, 2004.

[9] ISO/WD 26262-6: Road vehicles, Functional safety, Part 6: Product development: software level (2007).