Robustness of Automotive Applications Using Reflective Computing: Lessons learnt

Jean-Charles Fabre, Marc-Olivier Killijian
CNRS ; LAAS ; 7 avenue du colonel Roche
F-31077 Toulouse, France
Université de Toulouse ; UPS, INSA, INP, ISAE ; LAAS
F-31077 Toulouse, France
{Jean-Charles.Fabre, Marco.Killijian}@laas.fr

François Taiani
School of Computing and Communications, Lancaster University, Infolab21, Lancaster, UK
f.taiani@lancaster.ac.uk

ABSTRACT
In this paper, we present our experience and lessons learnt in applying a multi-level reflective approach to the design and implementation of an industrial embedded dependable system. We reflect in particular on the process by which ideal academic results and assumptions may be mapped to a concrete industrial context. More precisely, our reflection is based on our experience in building an adaptive defense software for a multilayer embedded platform in the automotive industry. This defense software provides a safety bag and is based on computational reflection, an advanced architectural mechanism to separate cross-cutting concerns. Our implementation uses the AUTOSAR middleware, the automotive standard for modular embedded software, and relies on software sensors to observe the behavior of the system, executable assertions to check on-line properties, and software actuators to perform recovery actions. This leads to defense software that is uncoupled from the real functional system and can be adjusted and specialized according to the needs of the system integrator.

Categories and Subject Descriptors
C.3 [**Special-Purpose And Application-Based Systems**]: Real-time and embedded systems

General Terms
Reliability.

Keywords
fault-tolerance, adaptation, reflection, robust software, automotive applications.

1. INTRODUCTION AND OBJECTIVES
As computing is increasingly used in a large range of everyday products, software dependability is emerging as a key issue in industries (e.g. automotive, home automation) in which it until recently only played a minor role. This evolution calls for new approaches to realise dependable software that take into account the specific standards, organizational needs, perspectives, and economic constrains of these industries.

For instance, recent efforts in the automotive industry have aimed to consolidate hardware infrastructures, proposing to execute code from multiple vendors on a single network of Electronic Control Units (ECUs). These industry-wide efforts are supported by a modular approach to software development, embedded in an industry wide standard, AUTOSAR. In this environment, software fault-tolerance measures are required to protect cars from malfunctions, in particular to detect errors and enact recovery actions. Unfortunately current approaches to these problems are often ad-hoc, and developed on a case-by-case basis. They are hence particularly costly to implement, integrate, and maintain. Software engineering methods are thus called for to offer a clear, systemic approach to the design, development, and integration of dependable software systems, while taking into account the organizational needs, perspectives, and economic constrains of the automotive industry. Solutions should in particular be highly generic (to allow the same or similar mechanisms to be applied across multiple applications, provided by different vendors), with clear design rules and processes, in order to track needs and impact and ease integration efforts.

Software engineering approaches to fault-tolerance have been proposed to tackle issues of reuse, genericity and integration. Computational reflection is one such approach that proposes to tackle elegantly these problems, and offer generic, compositional approaches to implement fault-tolerance mechanisms and harden software systems. Unfortunately, it is unclear to which extent reflection can be transposed to an industrial setting such as that of AUTOSAR, because of the usual reliance of reflective approaches on specific tools, and the high control they assume of the underlying runtime. To shed light on this question, we discuss in this paper our experience in applying multi-level reflection, an approach developed in an ideal-world academic setting, to an industrial context, the automotive industry. Key to our experience was the absence of any dedicated tool to implement our approach, and the need to respect a standard not originally developed to allow the kind of cross cutting composition that reflection allows. AUTOSAR also presents a number of challenges in that its programming model differs substantially from that of “traditional” computing runtime, with its own terminology (runnables, task bodies) and concepts, which we had to map onto.

In the rest of the paper, we first present some background information on multi-level reflection (Section 2). Section 3 explains the context of application and the overall reflective framework in automotive applications. Section 4 gives the major development steps of the defense software and some key elements of the implementation. We observe with satisfaction that the current standardized software architecture does provide basic mean to implement a reflective approach and take advantage of the separation of concerns between application and dependability mechanisms. Section 5 summarizes the lessons learnt. These lessons are inputs to the standards and the technologies used today to develop automotive embedded systems. Their integration into the standards should be seamless and will help automotive systems integrators better control the robustness and evolution of the systems they develop.

2. REFLECTIVE COMPUTING TECHNOLOGIES

Computational reflection refers to a computing system's ability to reason about and act upon itself. Originally proposed in the context of programming languages [1], reflection can be applied to solve elegantly a range of cross-cutting computational problems, from tracing, and encryption, through to hardening in real-time OSs [2], and replication [3,4,5,6]. It can also be credited to have strongly influenced novel programming practices such as aspect orientations. In architectural terms, a reflective system typically distinguishes between a *base level*, where the system's primary functions (guiding a rocket, computing a braking profile, processing a payroll) are implemented, and a *meta-level*, where computation about the base level takes place (Figure 1).

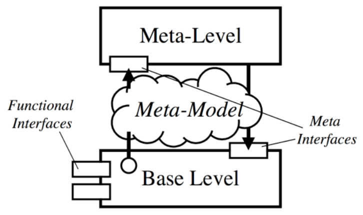


Figure 1. The key elements of a reflective system

The structure and behavior of the base level is usually exposed to the meta-level through a *meta-model*, which captures the key concepts that are made available to the meta-level to control and observe the base level. For instance a reflective object oriented language might expose the classes, methods, and attributes of the base level program, and offer interfaces (called *meta-interfaces*) to observe and modify these elements[1]. The elements of the meta-model might be structural (e.g. which methods does an object contain?) or behavioral (e.g. when is method *m* invoked?). The final set of elements included in a meta-model usually depends on i) the nature of the base level, and ii) the purpose of the reflective architecture. For instance, a reflective Real Time OS in which reflection is used to harden synchronization mechanisms might capture locking events (e.g. lock creation, activation, deletion), scheduling events (e.g. thread creation, blocking, waking, preemption termination), etc.

[1] Many popular programming languages such as Java are in this respect *partially* reflective in that they offer interfaces (in java.lang.reflect for instance) to observe a program's structure and execution, but no direct possibility to modify this program at run-time.

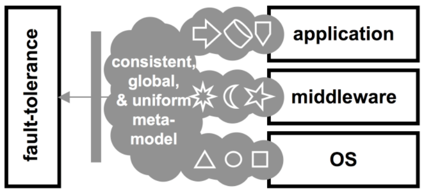


Figure 2. Multi-level reflection in layered systems

Multi-level reflection (Figure 2) extends the above principles to multi-layered systems [7]. It is based on the observation that most computer systems comprise multiple layers of interacting software, and that to efficiently implement fault-tolerance mechanisms, one often needs to combine control and observation information obtained from different layers. For instance, multi-threaded servers are typically not deterministic, which limits their use in active replication schemes. One approach to remove the non-determinism caused by multi-threading consists in intercepting and instrumenting all mutex operations to force a consistent scheduling across all replicas [8]. Doing so however can be extremely expensive [9]. In this situation, multi-level reflection is able to determine how mutexes used at the OS level relate to requests processed at the middleware level. This in turns allows the meta-level to intercept only those mutexes that do have an impact of request processing, and considerably lowers the overhead of this approach [10].

Opening-up all stacks of a system is however fraught with dangers, as it introduces new error propagation channels, and might render a platform brittle to change. To address this issue, multi-level reflection prescribes a demand-driven meta-model design, where developers first identify a family of algorithms they wish to support at the meta-level (e.g. a family of FT replication algorithms), and derive a reflective footprint, i.e. the set of reflective capabilities the system needs to support across its layers to implement this family. For instance to control non-determinism in RPC–style middleware like CORBA, a multi-level meta-model needs to include mutex, sockets, threads, request, request life cycle events, and notions of application boundaries, and should be able to integrate these together to detect which mutexes have an impact on request processing.

3. APPLICATION TO AUTOMOTIVE EMBEDDED SYSTEMS

3.1 Industrial Context and Problem statement

Early embedded software found in cars was mainly custom-made, limited in scope, and tightly linked to its underlying execution hardware. By contrast, today's vehicles contain an increasing number of functions that are controlled by software (up to several gigabytes of code in a car). To cope with this evolution, modern automotive systems follow some classical software architecting principles such as componentization and software layering to master complexity. By providing more structure to the code, these techniques aim at improving maintainability, preventing obsolescence, and promoting the use of software components including

COTS (*Components Off-The-Shelf*). To support these techniques, the automotive industry has developed a standardized software framework, AUTOSAR, (*AUTomotive Open System ARchitecture* [11]), that provides a standardized architecture for complex automotive systems. In particular, and quite importantly from a dependability viewpoint, AUTOSAR allows several applications with different criticality levels (notion of *Automotive Safety Integrity Level – ASIL*) to execute side by side.

Besides the use of architectural frameworks such as AUTOSAR, the production of robust systems must rely on principled software engineering techniques and development processes (ISOS26262 [12] for automotive applications) to produce correct components. These processes must in particular include fault-prevention techniques, and encompass the design of appropriate fault-tolerance mechanisms to deliver systems that are resilient at runtime. Finally, they must encourage designs that are **lightweight** for performance reasons, **easy to adapt** for the system integrator, and **evolvable** to take into account new requirements and evolving technologies.

In this context, our approach consisted in developing a defense software to improve the robustness of automotive embedded systems, while keeping this defense software clearly separated from the target system.

3.2 The AUTOSAR Software Architecture

AUTOSAR defines a common automotive software development environment, which promotes reuse and portability across different hardware platforms. The *AUTOSAR* methodology relies on description, static configuration, and automatic code generation tools, and favors tool interoperability.

The *AUTOSAR* reference model is composed of three principal software layers:

The Application Layer is divided into basic functions (called *runnables*) organized in software components. *Runnables* are the schedulable entities that the system integrator maps onto tasks managed by the operating system. During the integration phase, runnables may be grouped within a task, according to criteria such as workflow, periodicity, data consistency, etc…

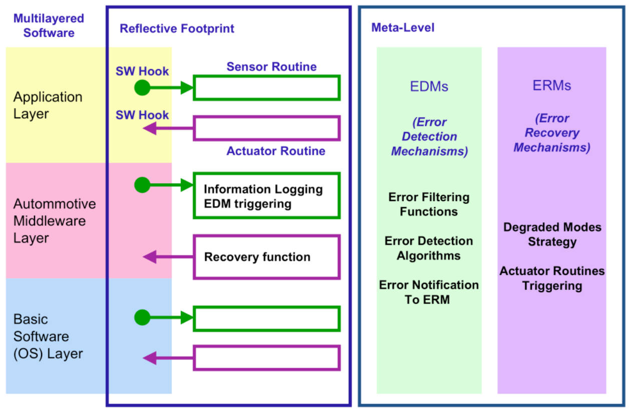
The AUTOSAR RunTime Environment (*RTE*) is a sort of middleware for automotive applications. In the *AUTOSAR* methodology, the *RTE* is automatically generated from the configuration of software components and basic software. The *RTE* plays the role of a glue code, using OS objects such as tasks, resources, and events to provide its own functionalities to the application layer.

Basic Software Layer provides two main components: *AUTOSAR OS* that manages task processing, alarms, memory and interprocess communication; and *AUTOSAR COM*, which deals with message exchange management.

3.3 Framework for the defense software

We have applied multi-level reflection to the AUTOSAR architecture to develop a defense software that is external to the target system. The resulting reflective framework enables a defense software to be defined, specialized and attached to a given operational component system. The target system and its corresponding defense software thus form a *self-checking component*. This idea is not new and can be found in other safety critical domains, like the *COM/MON* approach in avionics [13] and the notion of *Safety Bag* for railways signaling systems [14]. This type of wrapping approach is particularly attractive for automotive applications for obvious economic reasons.

Our defense software is organized in two parts: *Error Detection* (EDMs) and *Error Recovery Mechanisms* (ERMs). The EDMs are composed of several executable assertions, 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 of operation). The interface between functional and non-functional software, or *meta-interface* in the reflective terminology, 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.



Multilayered Software
Reflective Footprint
Meta-Level
Application Layer
SW Hook
Sensor Routine
SW Hook
Actuator Routine
Autommotive Middleware Layer
Information Logging EDM triggering
Recovery function
Basic Software (OS) Layer
EDMs
(Error Detection Mechanisms)
Error Filtering Functions
Error Detection Algorithms
Error Notification To ERM
ERMs
(Error Recovery Mechanisms)
Degraded Modes Strategy
Actuator Routines Triggering

Figure 3. Overall framework

The set of software sensors and actuators that is necessary and sufficient to check a given multilevel property at runtime constitute the reflective footprint (cf. Figure 3) of this property [7]. This reflective footprint is implemented using AUTOSAR hooks, this notion being essential for us. Initially devoted to debugging purposes in AUTOSAR, the hooks have been used as a control/command interface of the target system in our work, together with basic OS services.

4. DEVELOPMENT OF THE DEFENSE SOFTWARE

4.1 Development Process

The development process of the defense software follows the steps briefly described below:

1) Analysis of the target system and identification of application level faults, also called *Undesirable Customer Events (UCE)*. Such UCE correspond to safety requirements at the application level (cf. ISO26262 standard);
2) Selection of the faults impacting both the *control flow* and the *data flow* at various levels of abstraction (within various software layers) that could lead to the previously identified UCE. This step demands a clear understanding of the system design and to some extent of its implementation (e.g. how runnables are mapped into tasks);

3) Definition of the dependability assertions that must be verified on-line from a history of some application state information. This requires the precise identification and capture of the state items required for the on-line evaluation of these assertions;
4) Definition of the required error detection and recovery mechanisms. This implies defining the way error events are signaled and processed, but also what kind of recovery action must be performed (re-execution, reset, move to a safety state, or user-defined degraded mode of operation);
5) Definition and implementation of the corresponding observation and control mechanisms, using existing hooks, and to some extent user-defined hooks that do not belong the standard;
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.

In our case study three synthetic applications have been developed: air conditioning, airbag, and torque transmission control system.

4.2 Implementation

The three applications run on top of the RTE, which provides communication variables for the runnables that are mapped onto OS tasks. In our approach, safety constraints can be associated to individual applications, even when they all share the same embedded runtime. Just to give some examples extracted from our case study, a safety requirement for a given application (UCE) might be related to either a data flow or a control flow error, as illustrated below.

> The operational mode of the air conditioning system can only be computed when all inputs are available (dashboard command, filters and sensors values)

Figure 4. Example of Data flow error / Bad input values

> The torque control system is blocked (more than 1s) in mode 1, while the engine status mode is 2 (whereas it should switch to mode 2 as well)

Figure 5. Example of Control flow error / Incorrect transition

The violation of these properties can lead to customers' discomfort, at least, or even to engine damages in extreme cases.

The key challenge is then to derive assertions from the above UCEs. An assertion is defined with respect to the implementation of the applications it is associated with, and implemented using hooks and OS services (or system calls). These means are used to i) trace and log the input, output, and behavior relevant to the assertion, ii) trigger the assertion verification, and finally iii) activate recovery actions.

For detection, an assertion can be a simple logical expression or a small program using logged data (runnables inputs, RTE variables, OS data items like tasks id, priorities, etc.). In practice we have used standard hooks like `PreTaskHook()` and OS services `GetTaskID()` of the OS, but also several user-defined hooks. The latter were necessary to capture runnables execution within tasks, i.e. what is the runnable running in a given task. The hooks available in the standard are mostly located at the level of the RTE and the OS interfaces and were not sufficient to implement some of our assertions. Some hooks have been modified, some other added by hand in the generated source code.

Regarding recovery mechanisms, we used RTE kooks and services (like `Rte_IrvWrite`) to update communication variables but also *AUTOSAR OS* services like `ChainTask()` to terminate a task and trigger another one, `TerminateApplication()` to terminate and application, `TerminateTask()` to suspend a task.

From a dependability viewpoint, the evaluation of the defense software was carried out by fault injection, more precisely by mutation testing. Mutants were developed from the task bodies description, provoking the corruption of the control flow, the data flow or both, leading thus to an application level failure of the application (UCE). Most of the errors were detected, the rest being not captured by our assertions due to a lack of observability, or for which our recovery mechanisms were ineffective (lack of error detection or tolerance coverage).

A full account of the work carried out can be found in [15, 16], but also in more details in Lu's PhD thesis [17].

5. Lessons Learnt

First, we report on the problems, constraints, the limitations we faced in applying a generic reflective framework to an industrial case study. Second, we analyze the benefits that could be obtained by relaxing some of the constraints implied by the ideal world described in Section 2.

Although we could not rely on any advanced reflective technology, but had to work with AUTOSAR instead, the separation of concerns provided by reflection proved to be of high interest to our industrial partners, both from an integrator's point of view (the car manufacturer Renault in our case) and from a system provider's perspective (Valeo in our case). The ability to separate the design and implementation of the error detection and recovery mechanisms from the functional component architecture provided a clear advantage in terms of maintainability, understandability, and code organization. In contrast to more ad-hoc practices, our partners perceived reflection as providing a clear, disciplined approach to organize the development of robustness mechanisms in a systematic way.

Beyond code, concrete tools, and mechanisms, this type of feedback shows the importance of conceptual models to drive, communicate, and reason about detection and recovery mechanisms in embedded systems; and that such concepts can be useful even in the absence of dedicated technology, provided some minimal capabilities (hooks in our case) are available. This also illustrates that advanced software engineering techniques, such as multi-level reflection, can be particularly interesting to practitioners, and that path-ways exist to transpose existing academic results into concrete pre-production environments.

As a second clear benefit, our experience shows that reflective concepts can be applied without relying on any specific language, tools, or ADL approach. We used reflection at an architectural level, which made it inherently compatible with the AUTOSAR software infrastructure and the ISO26262 development process prescribed by our case study. A key enabler in this strategy was the possibility to realize software sensors and actuators within AUTOSAR as means to control the system. Interestingly, the AUTOSAR hooks we ended up using had not been originally designed as sensors and actuators, but had been primarily included for debugging and tracing purposes.

This pragmatic approach came, however, with a number of downsides and limitations if compared to an ideal solution. Ideally, in order to maximize the range of detection and recovery mechanisms that can be implemented, all aspects of the state and behavior of a system should be observable and controllable. The actual levels of observability and controllability effectively realized should then be determined by the reflective footprint of the targeted mechanisms (reification of events, introspection of data structures, intercession features, i.e. activation of actions). Industrial reality is quite different from this ideal situation. Although there is a clear link between the *Undesirable Customer Events* and the reflective footprint of the corresponding assertions, any implementation is constrained by the capabilities of existing hooks and OS calls to implement the verification of the property. This is a particularly limiting constraint, but one that is unavoidable to remain consistent with the AUTOSAR standard and the production tools used during the development process. For instance, user-defined hooks at any level of the system require an invasive form of instrumentation. Although this is not particularly difficult to realize since the source code of the RTE is available, it remains unacceptable for an industrial software production line. The hooks and other system calls must belong to the standard to be considered by the tools used to generate the RTE and the final system.

In other words, the available hooks and system calls provide a language to implement safety assertions. The capabilities of this language determine how efficiently the required assertions can be implement, or even if they can be implemented. The richer this language is, the easier it becomes to realize powerful and efficient assertions, but the more demands it sets on the capabilities prescribed by the standard.

In the light of our experience, one can think then of proposing some extensions to the standard, such as additional hooks (e.g. `Runnable_Start`), extended hooks signatures (e.g. extra parameters), and extra system calls (e.g. `Kill_Task`). A key capability to be added, for instance, would be to explicitly expose the notion of `Application_ID`/`Runnable_id` at all layers of the system architecture. This extension is motivated as follows. In AUTOSAR, multiple applications can run on the same platform, while possessing different criticality levels (ASIL). The verification of a given property related to a safety requirement of a highly critical application typically implies logging data and performing appropriate checks at well-defined execution points. Capturing the required data for all applications is possible but particularly counter-efficient, an acute limitation in an embedded environment. The availability of `Application_ID`/ `Runnable_id` would allow the data capture to be limited to the highly critical applicationd, and more generally enables the verification of fine-grained properties with limited performance penalty.

Interestingly, we only encountered a few multilevel properties in our prototype. The major reason for this is that most of the system is included in the RTE, with few system functionalities implemented elsewhere. The AUTOSAR OS for instance is a simple scheduler. The runnables are treated as macroinstructions. Multilevel reflection did play a key role, however, in the situation discussed in the previous paragraph, which consisted in exposing through manual instrumentation the `Application_ID`/ `Runnable_id` at all layers to be able to target the behavior of one particular critical application among several others sharing the same *Electronic Control Unit (ECU)*.

The metamodel that resulted from our experiment turned out to be quite simple and tightly linked to both the AUTOSAR programming model (runnables mapped to tasks and communicating through dedicated variables), and the observation and control capabilities of the AUTOSAR platform. In this context, our focus on control flow and data flow errors was a natural choice, as those may arise from any type of faults (residual software faults within runnables, integration faults possibly dues to incorrect tools, hardware faults impacting both data and control flows). Interestingly, our industrial partners were more convinced by the possible occurrence of faults impacting data than control flow. They naturally thought of data, typically obtained from external physical captors, as suspicious, but they struggled to envisaged that control, being internal to the software system, could become corrupted, because they implicitly assumed that runnables, the design process and the tools were zero default. Although fault tolerant computing appears as a necessity to these practitioners for such a complex software infrastructure, they tend to consider software as correct by construction, and find it hard to admit that software faults may occur. This perception goes beyond performance overheads that could prevent some solution to be used (replication for instance). Rather, it is a matter of limited background with complex software architectures running on more sophisticated *Electronic Control Units* (e.g. Freescale S12XE or even multi-core processors) for practitioners.

From an academic viewpoint, the use of reflective computing capabilities to develop external defense software has been limited. The difficulty for us was first to analyze the target software platform, its accompanying development process and tools. The second difficulty was the limitation of the observation and control features, namely the hooks. Our "manual" approach, i.e. developing new hooks by hand, was not accepted by our industrial colleagues for very sound reasons. However, they did accept that hooks should be promoted for dependability reasons in the AUTOSAR standard and, since they contribute to the standard as company representatives, were ready to argue which additional hooks and intercession system calls are necessary and why. We see this as a particularly beneficial result of this work, both from an industrial and academic viewpoint.

Looking forward, a more detailed metamodel of automotive applications would ideally be needed, that would more directly take into account real-time aspects, RTE features and mapping to OS objects, and the distribution of applications among several ECUs. We think that multilevel reflection is likely to provide the most benefit on finer-grained meta-models that more precisely expose how low-level services are shared between application components. These low-level services are usually critical to a system's stability, as their misbehavior may impact a large range of applications, or even the whole system they support. It is important to ensure in this case that a non-critical application may not be allowed to impair the behavior of a critical one. Considering the increasing complexity of embedded software systems in the automotive industry, this is certainly a situation for which researchers and practitioners alike should start preparing themselves.

Another limiting factor is the notion of standard that should be obeyed by all players. Considering the code generation strategy of AUTOSAR, the use of compiler-based reflective capabilities during the code generation process could in particular be of high interest to improve observability and controllability. In practice, the tools should be reflective or, to be more concrete, "aspectizable". Aspects could then be used on a case-by-case basis to adjust the hooks to the needs.

Finally, we would like to stress that the most challenging part of our experiment consisted in identifying the assertions from the UCEs, and then in determining the recovery actions to be taken. These issues are largely independent from the chosen implementation strategy, whether conventional (i.e. based on ad-hoc instrumentation) or reflective. However, even for these problems, we found the reflective approach to be useful, because it made visible the problems and the possible solutions. A good knowledge of the implementation is however mandatory in both cases.

6. CONCLUSION

Adaptation is a crucial issue in today's computer systems. Although automotive systems remain largely static, their execution model relatively simple, and their observability limited by the AUTOSAR standard, we can say from our experience that the reflective approach was both useful and well received by our industrial partners. In particular, the separation of concerns provided by reflection between functional aspects and dependability mechanisms appealed to our industrial partners. This might seem natural from an academic point of view, but is far from trivial considering the many constraints faced by automotive engineers, and we think is quite encouraging considering the difficulties of knowledge transfer between academia and industry. Multi-level reflection was perceived as a new disciplined way of thinking about dependability mechanisms, that could be applied in practice, and offered much better levels of maintainability over ad-hoc instrumentation, which all agreed was unsustainable for complex component-based systems. The key lesson we learnt is that the approach should comply with the standards and related tools to be widely accepted in practice, and we are confident that our industrial colleagues will do their best to promote this work in the relevant standard bodies.

7. ACKNOLEDGMENTS

The authors wish to thank very much Caroline LU for her PHD work on the subject. She was able to make a bridge between our academic views and the real automotive world. We would also like to thank Philippe Quéré (Renault TechnoCentre in Paris), Luc Fougerousse and Yannick Moreaux (Valeo) for their support and contributions in this work. We also acknowledge the SCARLET project and in particular IRCCYN colleagues for the help regarding AUTOSAR OS.

8. REFERENCES

[1] 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).

[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] G. Agha, et al. "A Linguistic Framework for Dynamic Composition of Dependability Protocols.", in the IFIP Conference on Dependable Computing for Critical Applications (DCCA-3). 1992. Palermo (Sicily), Italy: Elsevier. p. 197-207.

[4] B. Garbinato, R. Guerraoui, and K.R. Mazouni, "Implementation of the GARF Replicated Objects Platform.", Distributed Systems Engineering Journal, 1995. 2(1): p. 14-27.

[5] T. Pérennou and J.-C. Fabre, "A Metaobject Architecture for Fault-Tolerant Distributed Systems : the FRIENDS Approach.", IEEE Trans. on Computer, Special Issue on Dependability of Computing Systems, 1998. 47: p. 78-95.

[6] J. Salas, R. Jimenez-Peris, M. Patino-Martinez, and B. Kemme. 2006. Lightweight Reflection for Middleware-based Database Replication. In *Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems* (SRDS '06). 377-390.

[7] 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).

[8] Basile C., Kalbarczyk Z., Iyer R. K.: A Preemptive Deterministic Scheduling Algorithm for Multithreaded Replicas. IEEE Int. Conf on Dependable Systems and Networks (DSN'2003), San Francisco (CA, USA), pp. 149-158 (2003)

[9] Napper J., Alvisi L., Vin H. M.: A Fault-Tolerant Java Virtual Machine. IEEE Int. Conf on Dependable Systems and Networks (DSN'2003), San Francisco (CA, USA), pp. 425-434

[10] Taiani, F., Fabre, J.C., Killijian, M.O.: A Multi-Level Meta-Object Protocol for Fault-Tolerance in Complex Architectures. IEEE Int. Conf on Dependable Systems and Networks (DSN'2005), Yokohama, pp.270-279 (2005).

[11] AUTomotive Open System ARchitecture, http://www.autosar.org

[12] ISO/WD 26262-6: Road vehicles, Functional safety, Part 6: Product development: software level (2010) : http://www.iso.org/iso/catalogue_detail.htm?csnumber=51362

[13] 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.

[14] 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), pp 463-471(1995).

[15] C. Lu, J.-C. Fabre, M.O. Killijian, "Robustness of modular multilayered software in the automotive domain: a wrapping-based approach", in Proc. of the 14th Int. IEEE Conf. on Emergent Technology and Factory Automation (ETFA'09), Palma-de-Mallorca, Spain, Sept. 2009.

[16] C. Lu, J.-C. Fabre, M.O. Killijian, "An approach for improving Fault-Tolerance in Automotive Modular Embedded Software, in Proc. of the 17th Int. IEEE Conf. on Real-Time and Network Systems (RTNS'09), Paris, France, Oct. 2009.

[17] C. Lu, "Robustesse du logiciel embarqué multicouche par une approche réflexive : application à l'automobile (in French), Dec 2009, National Polytechnic Institute of Toulouse (http://ethesis.inp-toulouse.fr/archive/00001060/)