An Architectural Approach to Autonomics and Self-management of Automotive Embedded Electronic Systems

D J Chen, R Anthony, M Persson, D Scholle, V Friesen, G Deboer, A Rettberg, C Ekelin

An Architectural Approach to Autonomics and Self-management of Automotive Embedded Electronic Systems

D.J. Chen[1], R. Anthony[2], M. Persson[1], D. Scholle[3], V. Friesen[4], G. deBoer[5], A. Rettberg[6], C. Ekelin[7]

1: Royal Institute of Technology - KTH, SE-100 44 Stockholm, Sweden
2: University of Greenwich, SE10 9LS London, United Kingdom
3: Enea AB, SE-164 21 Kista, Sweden
4: Daimler AG, 89081 Ulm, Germany
5: Robert Bosch GmbH, 31132 Hildesheim, Germany
6: The University of Paderborn/C-LAB, 33102 Paderborn, Germany
7: Volvo Technology AB, SE-412 88 Göteborg, Sweden

Abstract: Embedded electronic systems in vehicles are of rapidly increasing commercial importance for the automotive industry. While current vehicular embedded systems are extremely limited and static, a more dynamic configurable system would greatly simplify the integration work and increase quality of vehicular systems. This brings in features like separation of concerns, customised software configuration for individual vehicles, seamless connectivity, and plug-and-play capability. Furthermore, such a system can also contribute to increased dependability and resource optimization due to its inherent ability to adjust itself dynamically to changes in software, hardware resources, and environment condition. This paper describes the architectural approach to achieving the goals of dynamically self-configuring automotive embedded electronic systems by the EU research project DySCAS. The architecture solution outlined in this paper captures the application and operational contexts, expected features, middleware services, functions and behaviours, as well as the basic mechanisms and technologies. The paper also covers the architecture conceptualization by presenting the rationale, concerning the architecture structuring, control principles, and deployment concept. In this paper, we also present the adopted architecture V&V strategy and discuss some open issues in regards to the industrial acceptance.

Keywords: embedded middleware, dynamic configuration, autonomic computing

1. Introduction

DySCAS (Dynamically Self-configuring Automotive Systems) is a project funded by the European Commission that aims to advance basic technologies and introduce context-aware and self-managing behaviours into automotive control systems [1]. This will allow an automotive embedded electronic system to dynamically configure itself according to the environmental and internal conditions, to cope with unexpected events, and to handle emerging use cases and external devices not known at the deployment time.

In modern automotive vehicles, embedded electronic systems have been widely employed for bringing in advanced features in regards to driver assistance, fuel efficiency, dynamics control and active safety, accounting for a very large portion of all innovations in the automotive industry. However, due to increasing expectations on flexibility, dependability, time-to-market and cost-efficiency, current state-of-the-art vehicle electronic systems, for which a static configuration is defined during the development process and remains stable over the complete lifetime of the vehicle, will not be sufficient. Future scenarios are typically concerned with: 1) building ad-hoc networks with a number of mobile devices to share functionality, 2) enhancing support for resource optimization and QoS (quality of service), maintainability, and dependability, 3) cost efficient and reliable field-upgrades of software to enable the latest innovations and personalization. These needs are further explained in Section 1.1 of this paper in terms of the DySCAS use cases.

Current and emerging automotive standards address the integration and configuration challenges of automotive embedded systems; AUTOSAR (AUTomotive Open System ARchitecture) is the most prominent of these [2]. The AUTOSAR standard allows (re-)use of “off-the-shelf” components and services across different manufacturers by providing an open standard for the componentization of automotive software and platform services. While the introduction of such standards in the automotive industry constitutes a very important progress, their delimitations on static configuration implies that the challenges relating to the above mentioned future scenarios remain open.

DySCAS intends to complement such automotive standards in regards to dynamic self-adaptive configuration, while interoperating or at least co-

existing with certain deployed static technologies, and integrating related state-of-the-art approaches that traditionally target other application domains. These approaches include in particular autonomic computing [3, 4], control-theoretic approach to computer control [5], and middleware [6, 7, 8]. The autonomic computing paradigm advocates self-managing behaviours that allow software programs to modify their own behaviours according to the contextual conditions, and is widely acknowledged to be a solution to the software complexity crisis. The control-theoretic approach has been used to support load-balancing and quality of service. State-of-the-art middleware technology provides basic mechanisms for communication transparency in distributed systems and configurational adaptability.

This paper provides an overview of the approach taken by the DySCAS project to dynamically self-configuring automotive embedded electronic systems. It presents the targeted application area and the architectural implications in Section 2, the architecture solution in Section 3, the mechanism and basic technologies adopted for embedded reasoning and decision-making in Section 4, and the adopted architecture V&V strategy in Section 5. The architectural modelling is currently being performed in UML2 with an upcoming alignment with the EAST-ADL2 architecture description language [9, 10]. We conclude by highlight the open issues in Section 6.

2. DySCAS use cases and their architectural implications

In DySCAS, the envisioned future needs of vehicle embedded electronic systems have been investigated by the automotive partners and captured in terms of four generic classes of use-case (GUC) which each comprise several specific use-cases (SUC) with related functionality [11]. Such use cases then forms the basis for deriving the functional and nonfunctional requirements as well as for the verification and validation through simulation and experimental systems. See also Figure 1 for an illustration of the mapping from problem domain to the solution domain according to this strategy.

GUC1. A new device is attached to the vehicle – relating to the dynamic discovery and incorporation of devices that a vehicle comes into contact with. The 'device' can be a mobile phone, PDA, MP3 player etc, but can also be a wireless network hotspot. For the intended feature, the system must perform a series of actions which, depending on the SUCs of concern, can include: discovery of the device, establishing connectivity, identification of the device or its owner (for security), negotiation of service provision (level of service, direction of service), providing or accepting the actual service, service termination and device disconnection.

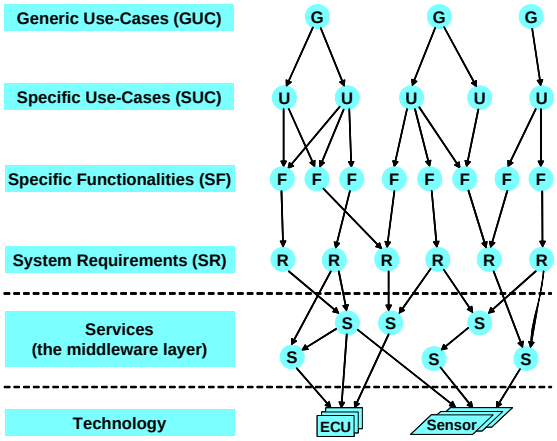

Figure 1. Mapping from problem domain to solution domain in DySCAS.

The use cases represented by GUC1 necessitate embedded reasoning and decision support in respect to access control, integration of functionality, and interoperation. In particular, it is concerned with the determination of which devices can be connected based on the type of device, its ownership, the services it offers or requests, vehicle owner/driver preferences for security and service prioritization, and the context such as the amount of available processing and storage resources available. This last point is very important because the use cases are generally related to infotainment services which involves media processing and streaming and in addition to requiring substantial resources, also has a real-time aspect. Specific architectural impacts of this class of use case include the need for open and standardized communication interfaces, an API for device drivers, and a means of learning and storing dynamically created decision details reflecting driver preferences with respect to various devices and configurations.

GUC2. Integrating new software functionality – relating to dynamically maintain the functionality by operating or application software. This can involve adding totally new components or applications, as well as upgrading and downgrading existing components and applications. It targets the situations such as when a new functionality has become available, a problem has been detected with current configuration, a time-limited licence has expired for a specific add-on service or a certain personalised services needs to be removed due to the change of car ownership. The use case can be internally triggered, for example to resolve an incompatibility or dependency problem; or can be externally triggered, as when a user requests a specific feature update.

The use cases represented by GUC2 imply embedded reasoning and decision support with respect to configuration management, impact

assessment of changes, and error handling. This in particular include the resolutions of whether an upgrade is necessary, whether an upgrade is allowed (given the current set of component dependencies), which components to upgrade, and, subsequently, whether to rollback. Specific architectural impacts of this class of use case also include the need for a modular binary image, and storage of current configuration details (component versions, dependencies), and of allowed configuration details, in a repository.

GUC3. Closed reconfiguration – relating to providing support for advanced system-wide error handing and fault treatment in a transparent way. For example, should an application service fails, due to software bugs, hardware problems, or power shortages, the DySCAS middleware will synchronize related services and then relocate the service according to the vehicle state. This may also involve shutting down a less important service to free up resources.

The architectural implications of GUC3 include support for detection of unexpected events as well as support for self-diagnosis, impact analysis and confinement of errors, online fault localization and treatment. Moreover, there are also needs of support for state transfer, process migration, checkpoints, and logging of historical data and state information.

GUC4. Resource optimization – relating to dynamically reorganise and balance workloads to maximise the efficiency with which resources are used. This includes for example dynamic selection among redundant sensors or ECUs for reliability and QoS, shutting down unused devices for power saving.

The implied architecture support will include the online resolution of which services should be active at each ECU, which services/ECUs can be shutdown. Specific architectural impacts of this class of use case include the need for instrumentation (power availability etc.) and the ability to shutdown ECUs.

To satisfy these use cases, the inherent conflict between flexibility and robustness in distributed embedded systems need to be carefully resolved by the architecture. One issue is concerned with the provision of component and architectural information when assessing the impacts of potential changes (e.g., adding a new software component). For dynamic configurations, the information usually relates not only to the system functionality but also to the performance, safety, resource constraints, and many other aspects. To this end, it is important that the prerequisites and constraints of components in regards to interoperability, portability, and performance, and other qualities of concern can be maintained at runtime along with a consistent global view of configurational states and resource deployment. The complexity of embedded systems also implies that multi-criteria trade-offs need to be supported for resolving configurational alternatives.

Besides the reasoning and decision support, dynamic configuration in general also implies advanced communication and execution support. This includes location transparency and access transparency (see e.g., [12]) that are considered necessary to migrate and reallocate functionality, to move a (mobile) resource, to allow efficient interactions with replicas. One performance related challenge is the middleware overheads, both in time and in resource utilization. The platforms on which DySCAS will operate (i.e. ECUs in vehicles) have several domain specific restrictions, including limited processing and memory. For example (many ECUs have only 8-bit processors which run at low clock speeds and have limited amount of memory, although vehicles are likely to have some 32-bit ECUs as well). Moreover, the conventional communication buses used in automotive systems have restricted data rates. For example, the CAN bus operates at up to 1Mbit/ second (whilst the MOST bus runs at up to 24.8Mbits/second but is only supported on the most-powerful ECUs). The fact that the systems are embedded also makes it difficult to change the run-time code (operating system and application code). To minimize the impacts of such overheads, the choice of control algorithms, the instantiation, mapping, and allocation of middleware services, as well as the planning and controlling of the configuration and management tasks, are all of particular concern. For dynamic operations, determinism can be achieved through mechanisms that reserve and initialize necessary resources in advance. It is also possible to facilitate dynamic changes by allowing different operation modes of applications.

3. An overview of the DySCAS architecture

The architecture constitutes an overall design for the intended DySCAS middleware system where various policy-driven self-management mechanisms are defined, integrated, and realized in an automotive context. It specifies the external concerns (e.g., available vehicle information and technology constraints), the system functionality and building-blocks in terms of middleware services and components, the data and interfaces, the expected behaviours (e.g., the execution states and control flow), and the implementation design for software configuration and deployment. For complexity control and design effectiveness, the architecture design work adopts an incremental refinement process, running from conceptual design, to function design,

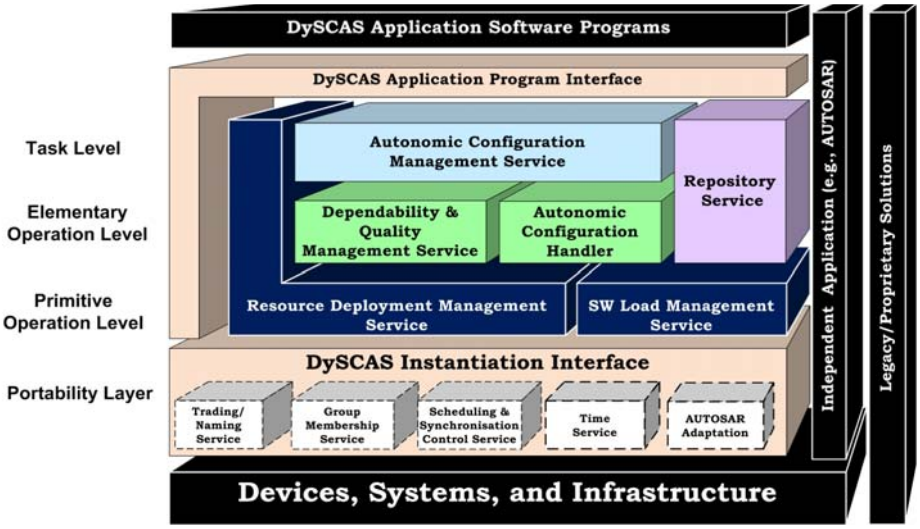

Figure 1. A schematic overview of the DySCAS architecture.

then to software design, and finally to detailed implementation design.

DySCAS middleware services

Figure 1 provides an outline of the DySCAS middleware architecture, including the major middleware services and the external interfaces towards the application software and target platform. For technical and practical reasons, DySCAS provides the freedom to run other standalone and legacy software applications along with DySCAS applications. Such external applications are not ported to the DySCAS middleware, but directly to the underlying target platform and hardware devices. The functional dependencies, in the form of communication between the middleware services, are summarized in Table 1.

Table 1. An overview of the interdependencies of DySCAS middleware services.

uses services of	Application Programs	DySCAS Application Program Interface	Autonomic Configuration Management Service	Repository Service	Dependability & Quality Management Service	Autonomic Configuration Handler	Resource Deployment Management Service	SW Load Management Service	DySCAS Instantiation Interface	Target Platforms, Infrastructure, and Devices
Application Programs	✓	✓	×	×	×	×	×	×	✓	✓
DySCAS Application Program Interface	✓	✓	✓	✓	×	×	×	×	✓	×
Autonomic Configuration Management Service	×	✓	✓	✓	✓	✓	✓	×	✓	×
Repository Service	×	✓	☑	✓	✓	×	×	×	✓	×
Dependability & Quality Management Service	×	×	✓	✓	✓	☑	✓	×	✓	×
Autonomic Configuration Handler	×	×	✓	✓	✓	✓	✓	✓	✓	×
Resource Deployment Management Service	×	✓	☑	✓	✓	☑	✓	×	✓	×
SW Load Management Service	×	×	×	✓	×	☑	☑	×	✓	×
DySCAS Instantiation Interface	☑	☑	×	✓	×	×	☑	☑	✓	✓
Target Platforms, Infrastructure, and Devices	☑	×	×	×	×	×	×	×	☑	✓

✓ is allowed to use × is not allowed to use
☑ only restricted use (for notification of events)

On top of the portability layer, the middleware system is structured into three levels of control. This layering strategy allows a hierarchical decomposition of control tasks through which a larger reconfiguration problem is reduced to more elementary operations. This pattern is widely adopted in many complex control systems [13].

The DySCAS middleware services are further divided in two groups: (1) *optional services*, providing basic support for network and platform transparencies, and (2) *core services*, providing embedded reasoning and decision support through the contained policies and other control functions. The optional services are placed in the *DySCAS Instantiation Interface* (shown as dashed blocks in Figure 1). These services interact directly with the underlying system platforms and provide support in respect of portability and interoperability, transparent communication, concurrency control, membership management, much as the support offered by other traditional middleware systems. Such services are optional as the same services can be obtained through systems, network, or other external middleware. Under these circumstances, the components implementing such services act as wrappers/containers for the corresponding external services. The core middleware services provide the reasoning and decision support through the contained policies and other control functions. See Table 2 for an overview of these services.

While performing the policy-based reasoning and decisions, each middleware service is built to be context-aware, of which the general concept is illustrated in figure 2. An example of a middleware service is the device discovery support by the DySCAS *Resource Deployment Management Service*. The decision is based on information like the type of a newly presented device, the offered

TABLE 2. An overview of DySCAS Core Services and some of their properties

DySCAS Core Service	Overall System Roles	Related Autonomic Features [3, 4]
Autonomic Configuration Management Service	*Analyzer* for the overall impacts of requested dynamic changes; *Planner* of configuration tasks.	*Self-configuring* (on the online configuration reasoning support); *Self-healing* (on error-handling and fault removal).
Repository Service	*Repository* of files for decision policies, component images, and information relating to knowledge of configuration and diagnostics.	N/A
Dependability & Quality Management Service	*Dependability and security controller*; *Performance Optimizer.*	*Self-healing*; *Self-optimizing*; *Self-protecting* (for external accesses and service requests).
Autonomic Configuration Handler	*Coordinator* of distributed dynamic configuration operations.	*Self-configuring* (through the execution synchronization support).
Resource Deployment Management Service	*Monitor* of target and external resources that also brings together distributed information and generates normalized figures of quality feedback; *Executor* of dynamic configuration operations.	*Self-defining*; *Context awareness* (in respect to external devices/systems); *Self-configuring* (mainly through sensing and execution support).
SW Load Management Service	*Executor* of dynamic load operations.	*Self-configuring*.

services and/or resources, the context in terms of current vehicle state and presence of other devices. Other middleware services will be brought into play once a device has been recognised. For example the security support by the DySCAS *Dependability & Quality Management Service* must decide whether the device and the application services it offers are to be accepted into the system.

DySCAS middleware control paths

Across the architecture layers, there are three paths of control:

- *Path_1: context monitoring and event detection path;*
- *Path_2: reasoning and decision path;*
- *Path_3: actuation and synchronisation path.*

The context monitoring and detection path is the most data intensive, while the other two paths are mainly event driven.

The context monitoring and event detection path performs the role of monitoring the context given by the current status of vehicle applications as well as the current deployment of target and external resources. It detects the events/states of interest according to a set of predefined policies. In addition, this path is also responsible for consolidating the information from distributed sources and for providing normalized quality figures for the reasoning and decision functions. In DySCAS, this path runs from underlying system platform to the *Resource Deployment Management Service* via the *DySCAS Instantiation Interface*. Multiple context monitoring and event detection paths can exist in a networked system, targeting individual nodes, network realms, and the entire network system separately.

The reasoning and decision path is triggered when an event or state of interest is detected (e.g., discovering an external device) by the monitoring and detection path. It performs the roles of assessing such events/states and planning for the configurational adaptations. The contained policies capture the configurational rules, including the allowed variability and constraints. This provides a system with the ability to reason about the correctness and efficiency of its current configuration, and to resolve potential configurational changes without eroding the architecture or violating the functionality and dependability (e.g., safety, security, and availability). In DySCAS, this path is subdivided into:

- a *dynamic configuration control path,* supported by the *Autonomic Configuration Management Service*, and

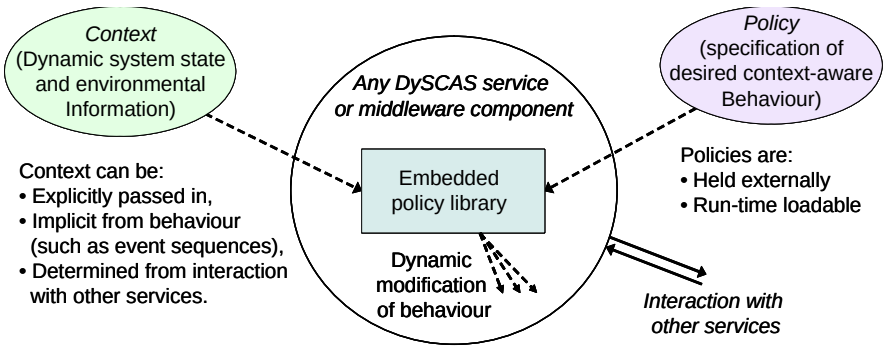

Figure 2. A configurable and context-aware DySCAS service

- a *dependability and QoS path,* supported by the *Dependability&Quality Management Service.*

Of great importance to the DySCAS middleware system is the actuations and synchronizations of dynamic configurational actions on a distributed system. This is supported by the actuation and synchronisation path, invoked by configurational decisions in the reasoning and decision path. For each dynamic configurational decision (e.g., updating a software component), there is normally a sequence of primitive operations (e.g., invoking transferring states, unloading, loading, initializing, and executing a software component). For primitive operations, the actuation and synchronisation path also provides the scheduling and triggering support across a distributed platform. During the executions of primitive operations, the status is frequently monitored. A failed execution may cause rescheduling of the primitive operations or revising of configurational decisions.

DySCAS middleware deployment concept

In DySCAS, each individual resource domain, ranging from an individual ECU node at the lowest level, to a network domain, and to an aggregation of networks, is allowed to have its own complete set of core services that together form a global monitoring and decision hierarchy in a cascade way. For example, in a networked system, there can be multiple middleware control paths, targeting individual resources separately (e.g., a node or a network realm). Normally, each of these services is deployed for an individual resource domain, such for each node and for an entire network domain (e.g., a group of ECUs sharing a specific communication bus). Global decisions in a networked system are then derived by consolidating local decisions. Each DySCAS middleware service can act as a proxy for consolidating a global system view or for obtaining system-wide decisions. See figure 3.

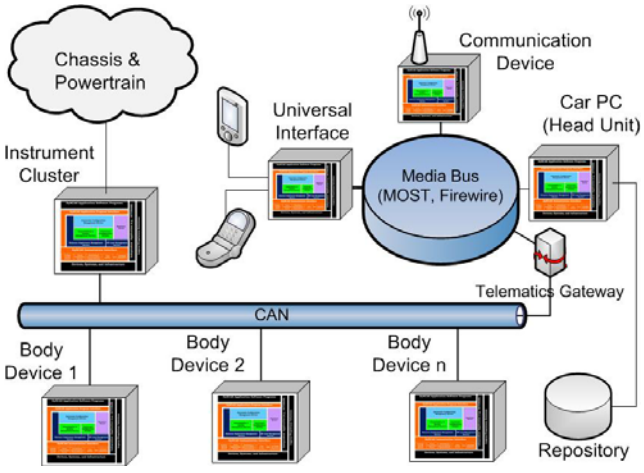

Figure 3. An example deployment of DySCAS middleware in vehicles.

For performance reasons, the DySCAS middleware also allows the context information suppliers and the decision makers to be allocated at different positions within a network according to the computational resources available. For dependability reasons, the services can be implemented with redundant components or distributed. When dealing with aperiodic tasks when their behaviour is not known, share-driven or server-based scheduling [14] is considered.

4. Policy as the basic mechanism for embedded configurational reasoning and decisions

In the DySCAS project, different technologies have been investigated for intended middleware services. As concluded in [15], different autonomics control approaches are more or less suitable for usage in a vehicular environment.

The main constraint in the automotive context is the limited resources typically available in an ECU. Computationally or memory intensive approaches hence are less relevant, and hence policies was chosen. The particular characteristics of policy-based computing that influenced this decision include: the policy logic is used to specify high-level behaviour, in a standard and platform-independent way; the policy logic is stored separately from the mechanism and is loaded at run-time initiation (or even during run-time), effectively as data rather than as application code; policy evaluation has low run-time resource requirements; and policy configuration is sufficiently flexible that it is generally applicable to all of the identified use cases.

Special considerations for DySCAS

A detailed review of the state of practice in policy-based computing has been provided in [16]. Basically, there are two ways of implementing policy through software. The first of these is based on statically embedding the actual policy logic (the rules) into the application code. During system start-up or run-time, the context information and configuration parameters (such as counts, thresholds and flags) are passed in. This form of policy configuration is also referred to as *template-configuration*. It is considered ideal for situations where a system needs to take into account context information such as user preferences or security flags for example, but the rules applied to these values are static. Another way of implementing policy through software is to store the configuration values and the policy rules externally to the software program, referred to as *policy-configuration*. This approach is much more flexible as a dynamic configuration of the policy itself is allowed.

When multiple policies are applied in a system, one challenge is concerned with unexpected feature

interaction and conflicts between the decisions. The DySCAS middleware system will employ hierarchical policies so that the complexity of individual policies stays low. A previous work on multi-policy will also form a basis for the resolution [17]. Several of the identified use cases require dynamic decisions to be made which are simultaneously influenced by several contextual factors. Thus there is an identifiable need for integrating engineering decision technique, such as Utility Functions (UF), to be supported the online architectural decisions. UFs have the particular strength of enabling the various contextual factors to be weighted (possibly dynamically) to reflect their relative importance when choosing amongst several alternative paths.

In DySCAS, dynamic updates of policies and dynamic creation of new templates could be used to support automatic detection and resolution of rule conflicts (such as in [18, 19]), to facilitate policy maintenance, and to expedite urgent processes. However, the challenges in regards to design and V&V complexity need to be resolved. The issues of concern include: whether policies can be changed during run time (on-line) or whether policies can only be loaded at software initialisation (in which case updates must occur off-line); whether policy implementation mechanisms support automatic self-adaptation of policies (for example to improve the performance of the system by dynamically tuning rules or parameters); and the extent to which policy mechanisms provide feedback to users concerning the effectiveness of policies, identified inefficiencies, or conflicts between rules. Upgrading a policy can be achieved using the same context information but perhaps utilizing more sophisticated logic. However, if the context information provided to the policy is somehow statically decided (for example it is provided in a hard coded communication relationship with other components) then the scope for logic upgrade is limited.

Usage of AGILE-Lite in DySCAS

The policy implementation that the DySCAS reference implementation is to be based upon the AGILE policy expression language and integration framework [20, 21]. AGILE has been developed in C++/.NET. The implementation library, documentation (including grammar specification and API usage), sample applications and policies are available at. On the other hand, AGILE is not optimised for resource-constrained environments.

To support the implementation in the targeted automotive environment as the DySCAS middleware, a lightweight version, AGILE-Lite has been developed. This library is using the same flexible grammar as AGILE, but has been designed specifically for deployment in embedded applications. The AGILE-Lite implementation library will be internally optimised for performance and resource efficiency, and has been written in C, as this has both high portability and code efficiency. This approach allows policies to be loaded dynamically, and to be replaced at run time. This facilitates highly flexible dynamic configuration, for example a new policy version could be delivered to the vehicle via a wireless hotspot and at some subsequent appropriate moment (non critical activity) the policy can be loaded into the relevant component without having to re-start it.

5. DySCAS V&V strategy

The V&V support for the development of DySCAS middleware is being conducted through analysis, simulation, and experimental platforms. For the purpose of complexity control and earlier feedback, an incremental approach regards to implementation detail is also applied.

The analysis work focuses on some important aspect of the middleware system, such as in respect to the compositional behaviours and the overall system safety in an automotive context. The simulation provides a faster evaluation support and allows better insight into the processes. A simulation can be run in slow-motion, and far more data can be made available to the developers if needed. To explore issues closer related to the hardware, the simulation work is conducted in part through the TrueTime [21] toolbox in Matlab/Simulink, which provides simulation models of common network types (e.g. CAN, Ethernet) and typical real-time operating systems (RTOS). In the DySCAS project, several experimental systems are currently being built up by the partners. These experimental systems will be used as the validators for the architecture in actual real systems. Many of the experimental systems will also be used as demonstrators, to show new features available in future automotive vehicles.

6. Conclusion and open issues

This paper has described the approach taken by the DySCAS project to a middleware system for autonomics and Self-management of Automotive Embedded Electronic Systems. It covers the motivations and reasoning behind the middleware architecture as well as the architecture solutions. Further, the usage of policy-based computing as the basic mechanisms for embedded reasoning and decision making in the DySCAS middleware has been outlined. Finally, the adopted V&V strategy has also been introduced.

One open issue is concerned with the industrial acceptance of dynamic configurable embedded systems. As the automotive applications are often safety relevant, it is important to introduce dynamic

behaviour in such a way that the system remains predictable and robust despite its autonomy. This requires that significant effort be devoted to verification of adaptive behaviours and validation of overall system concepts. Furthermore, new concepts must be developed for testing and legislation approval. Moreover, since the DySCAS middleware is more powerful and complex than static middleware solutions, it's implementation is related to additional costs in terms of extra bus capacities, CPU power, and memory consumption. It has to be proven within the project that these costs are fully justified by the benefits coming from DySCAS, in terms of reduced costs due to redundancy, increased flexibility and cost efficiency, and reduced time-to-market.

A migration path from existing static systems to dynamic DySCAS systems is given by the specification approach followed in DySCAS: Since the system specification is independent from the underlying self-configuration mechanisms, it is possible to restrict the self-configuration behaviour of the system very strictly. This may be reached by degrading the algorithms for self-configuration to a predefined set of tested system configurations which are activated depending on the system environments/situations. In the extreme case, a DySCAS system may be degraded to a static system. With this, a vehicle manufacturer is free to decide to which extent he would like to introduce reconfigurability to the DySCAS system, paving the way to a migration from existing static systems to DySCAS systems.

It is intended that DySCAS will form the basis of standards describing self-adaptive configuration and behaviour in automotive systems. It is mutually important for both AUTOSAR and DySCAS and highly advantageous for the beneficiaries (primarily the vehicle manufacturers, and thus the vehicle owners, drivers), if the results of DySCAS can be merged into the AUTOSAR roadmap.

8. Acknowledgement

The DySCAS project is funded within the 6th framework program "Information Society Technologies" of the European Commission. Project number: FP6-IST-2006-034904.

10. References

[1] DySCAS project website: www.DySCAS.org.
[2] AUTOSAR initiative: www.autosar.org
[3] P. Horn. "Autonomic computing: IBM perspective on the state of the information technology". In AGENDA'01, Scottsdale, AR, 2001. (www.research.ibm.com/autonomic)
[4] IBM, "An architectural blueprint for autonomic computing". IBM and Autonomic Computing, April 2003. (www-306.ibm.comm/autonomics/pdfs/ACwpFinal.pdf)
[5] K-E Årzén, et.al., "A Robertsson, Roadmap on Control of RealTime Computing System", EU/IST FP6 ARTIST2 NoW, Control for Embedded Cluster.
[6] P. Cointe, editor: "Meta-Level Architectures and Reflection". 2nd international conference, Reflection '99, St. Malo, France, volume 1616 of LNCS. Springer, 1999.
[7] D. C. Schmidt. "Adaptive middleware: Middleware for real-time and embedded systems". Communications of the ACM, Vol 45, Issue 6, June 2002.
[8] W. Emmerich, Engineering Distributed Objects, John Wiley & Sons, April 2000.
[9] P. Cuenot, D.J. Chen, S. Gérard, H. Lönn, O. Reiser, D. Servat, R. T. Kolagari, M. Törngren, M. Weber. "Improving Dependability by Using an Architecture Description Language". The 4th LNCS book on Architecting Dependable Systems (ADS), Springer, 2007.
[10] P. Cuenot, P. Frey, R. Johansson3, H. Lönn, M.-O. Reiser, D. Servat, R. Tavakoli Kolagari, D.J. Chen. "Developing Automotive Products Using the EAST-ADL2, an AUTOSAR Compliant Architecture Description Language". ERTS 2008 (AUTOSAR methodology).
[11] DySCAS Consortium, D1.2 Scenario and System Requirements, 2007. www.DySCAS.org.
[12] M. Kohli and J. Lobo, "Realizing Network Control Policies Using Distributed Action Plans, Journal of Network and Systems Management", 11 (3), Plenum Publishing Corporation, September 2003..
[13] J.S. Albus, F.G. Proctor, "A Reference Model Architecture for Intelligent Hybrid Control Systems", Proc. Intl. Federation of Automatic Control, USA, 1996.
[14] T Nolte. "Share-Driven Scheduling of Embedded Networks", PhD thesis, Department of Computer and Science and Electronics, Mälardalen Univ, Sweden, 2006.
[15] DySCAS Consortium, D1.1B Evaluation of Existing Technologies, 2007. www.DySCAS.org.
[16] R. Anthony "A Policy-Definition Language and Prototype Implementation Library for Policy-based Autonomic Systems", 3rd Intl Conf on Autonomic Computing, Dublin, June 2006, IEEE.
[17] R. Anthony, "Policy-based techniques for self-managing parallel applications", Knowledge Engineering Review, 21 (3), 2006, Cambridge University Press.
[18] R. Ananthanarayanan, M. Mohania and A. Gupta, "Management of Conflicting Obligations in Self-Protecting Policy-Based Systems", proc. of the 2nd Intl. Conf. on Autonomic Computing, IEEE, Seattle, 2005.
[19] J. Chomicki and J. Lobo, "Monitors for history-based policies", Policies for Distributed Systems and Networks, Springer, Berlin, 2001.
[20] AGILE support website: www.PolicyAutonomics.net.
[21] R. Anthony, "Policy-centric integration and dynamic composition of autonomic computing techniques", 4th Intl Conf on Autonomic Computing, USA, June 2007, IEEE.
[21] TrueTime: Simulation of Networked and Embedded Control Systems. Website: www.control.lth.se/truetime/

12. Glossary

ECU: Electronic Control Unit
GUC: Generic Use Case
SUC: Specific Use Case
QoS: Quality of Service
UF: Utility Function