A Middleware Approach to Dynamically Configurable Automotive Embedded Systems

Richard Anthony[1]
44 (0)20 8331 8482
R.J.Anthony@gre.ac.uk

Paul Ward[1]
44 (0)20 8331 8588
P.A.Ward@gre.ac.uk

DeJiu Chen[2]
46-(0)8-790 6428
chen@md.kth.se

Achim Rettberg[3]
49 441 9722-247
Achim.Rettberg@iess.org

James Hawthorne[1]
44 (0)20 8331 8588
J.Hawthorne@gre.ac.uk

Mariusz Pelc[1]
44 (0)20 8331 8588
M.Pelc@gre.ac.uk

Martin Törngren[2]
46-(0)8-790 6307
martin@md.kth.se

ABSTRACT

This paper presents an advanced dynamically configurable middleware for automotive embedded systems. The layered architecture of the middleware, and the way in which core and optional services provide transparency and flexible platform independent support for portability, is described. The design of the middleware is positioned with respect to the way it overcomes the specific technical, environmental, performance and safety challenges of the automotive domain. The use of policies to achieve flexible run-time configuration is explained with reference to the core policy technology which has been extended and adapted specifically for this project. The component model is described, focussing on how the configuration logic is distributed throughout the middleware and application components, by inserting 'decision points' wherever deferred logic or run-time context-sensitive configuration is required. Included in this discussion are the way in which context information is automatically provided to policies to inform context-aware behaviour; the dynamic wrapper mechanism which isolates policies, provides transparency to software developers and silently handles run-time errors arising during dynamic configuration operations.

Keywords

Automotive embedded systems, dynamic configuration, policy-based computing.

1. INTRODUCTION

Modern automotive control systems operate in complex environments in which many software and hardware components interact to provide a wide range of functionalities, and in which a diverse array of potential problems and inefficiencies can arise. The embedded nature of the systems brings additional problems such as restricted computational resources and updating behaviour is difficult. However, future use-scenarios imply frequent configuration changes to update versions, support field upgrades, and allow owner customisation for example for infotainment preference settings, fleet-specific configuration and driver profiles that can be taken from vehicle to vehicle. To meet these challenges, the traditional fixed behaviour systems must give way to more flexible dynamic systems. The trend is the same for many other types of embedded systems, [1]. The next generation of control systems need to have several potentially conflicting qualities which include high performance in real-time, high robustness, efficiency, extensibility and support for flexible, simple and fast (re)configuration.

The Dynamically Self-Configuring Automotive Systems project (DySCAS) is developing a middleware (MW) that meets the above mentioned challenges. The key motivational themes driving the design are: flexibility - to permit timely upgrades and reconfiguration at low cost; reliability - to ensure safety and predictability, and to support self-diagnosis and repair where possible; and transparency - so that application developers and end users are provided with appropriately abstract interfaces to the underlying control system and are thus relieved of the complexity burden.

To achieve the desired qualities, the DySCAS MW comprises a number of software components each of which can embed dynamically replaceable logic at pre-defined 'Decision Points' (DPs). The MW has been designed such

Conference name: ISVCS 2008, July 22 - 24, 2008, Dublin, Ireland.

[1] The University of Greenwich, Greenwich, London, UK.

[2] Royal Institute of Technology (KTH), S-100 44 Stockholm, Sweden.

[3] Carl von Ossietzky Universität Oldenburg, Offis e.V., Escherweg 2, 26121 Oldenburg, Germany.

that the application software sees a single system image and is not directly aware of physical hardware resources such as Electronic Control Units (ECUs) and sensors. This enables the MW to perform reconfiguration 'silently', for example to achieve load balancing for performance optimisation, or task relocation to overcome an ECU failure. A high-level policy grammar permits behaviour within each DP to be specified very flexibly, and to be changed when required without redeploying code. An innovative 'Dynamic Wrapper' (DW) mechanism enables the dynamic configuration behaviour to automatically collapse down to pre-defined static behaviour on a per-DP basis should any run-time problems arise which would affect the correctness of the dynamic adaptation.

2. Automotive challenges and DySCAS

Embedded electronic systems have been widely employed in modern automotive vehicles. By means of computer software and hardware, automotive embedded systems offer unique opportunities for advanced functionalities, high performance, and flexibilities in system development and maintenance. Today vehicles in series production already contain the same amount of electronics as aircraft did two decades ago. It is predicted that the share of automotive embedded systems in respect to a vehicle's total value will reach 40% by 2015, bringing in innovations and new features in driver assistance, fuel efficiency, vehicle integration and traffic safety [2]. The increased use of embedded electronic systems in vehicles, however, also implies growth and change in product and development complexity. For many advanced applications, there are emerging needs on the integration of distributed data and functionalities and the incorporation of behaviours with different criticalities and types (e.g., time- and event-triggered tasks), further characterized by real-time, resource and dependability constraints. In system development, such product complexity is augmented by the involvement of multiple stakeholders and organisations, heterogeneous technologies and components, and lifecycle concerns in regards to maintenance, upgrade, variability and reuse. To cope with the technical and managerial challenges, new technologies, tools, and methodologies are necessary [3].

Current automotive embedded electronic systems adopt a static configuration scheme, in which the environment assumptions, system functionalities and behaviours, component compositions and resource deployment are defined during the development process and kept stable over the complete lifetime. This is, however, insufficient for many future scenarios of automotive vehicles. Technological improvements have lead to more powerful, yet cheaper and smaller ECUs and a greater variety of sensor types; facilitating new and exciting use cases. User-expectations and competition between manufacturers have led to an increased importance being attached to the ability to support automatic dynamic configuration to achieve fault tolerance, optimised performance and user customisation.

Like embedded systems in other domains (e.g., avionics and med-tech), automotive embedded systems are safety critical because of their effects on the vehicle, the environment and humans. Meanwhile, automotive embedded systems also constitute consumer products and are thus sensitive to usability, cost-efficiency, and reliability. For example, it is expected that future automotive vehicles will provide the ability of building ad-hoc networks between vehicles and with external mobile devices to share information and functionality. The ability of allowing cost efficient and reliable field-upgrades of software is also considered important for vehicle customization, personalization, and incorporation of technology innovations. For the reasons of dependability, time-to-market and lifecycle efficiency, future scenarios of automotive embedded systems also call for enhanced QoS (quality of service) support. This will permit post-development time optimization according to the actual resource utilizations and operation conditions. The support for load-balancing, on-line V&V, and error-handling also makes it possible to reduce the number of ECUs, wiring, and power consumption and to potentially migrate part of the costly development-time testing effort to run-time.

DySCAS aims to advance the basic technologies and introduces context-aware and self-managing behaviours into automotive embedded electronic systems [4]. Targeting the above mentioned future scenarios, the DySCAS approach explicitly addresses the automotive needs in regards to configuration flexibility, quality assurance, and complexity control in particular in the infotainment and telematics domains. DySCAS develops and proposes a MW system that allows automotive embedded systems to dynamically reconfigure themselves according to the environmental conditions, application states and resource deployment, to cope with unexpected events, emerging use cases and optimization needs, and external devices not known at the deployment time. To promote industrial acceptance, the DySCAS approach explicitly addresses a number of challenges facing automotive products, including the MW overheads and predictability, the provision and management of design and operation information at run-time, the synchronisation of distributed behaviours, and the concerns of usability, security, and robustness. These are discussed in the remainder of this section.

Many automotive applications have real-time requirements, ranging over closed loop periodic controllers to multimedia and communication functions. The systems are often highly resource constrained because of the large series being produced. To introduce MW solutions in automotive embedded systems, the performance overheads in time and in resource utilization (e.g., bus, CPU, and memory) need to be properly handled. While overheads are unavoidable because of the MW mechanisms, the DySCAS approach aims to keep the MW overhead as small as possible while

making the behaviours predictable. To this end, the choice of algorithms, the instantiation, mapping, and allocation of MW services, as well as the planning and controlling of the MW tasks, are all of importance. For the dynamic operations, predictability will be achieved through mechanisms that negotiate and reserve necessary resources in advance and provide synchronisation with application conditions. One particular concern is the choice of target platforms. The DySCAS project does not restrict the MW solutions but stipulates a strategy by explicitly specifying the assumptions and dependencies of the MW services on the system software and hardware platforms. The AUTOSAR RTE is here used as a reference [5] complemented by some typical MW configuration and instantiation alternatives, allowing the engineers to further optimize the needed capacity in engineering practices.

One challenge related to the QoS and dynamic configuration is dealing with the complexity of embedded systems and the existence of multiple quality concerns. Instead of a single optimization need, there can be multiple and sometimes conflicting adaptation goals and behaviours, varying according to the vehicle states and service modes. Moreover, depending on the functional interdependencies of application programs and the sharing of system resources and devices in a system, a change on the system configuration or component behaviour can affect the overall functionality and performance negatively. To avoid unexpected impacts on system dependability, the decisions also need to take the vehicle state, such as in terms of vehicle speed and direction into consideration. For example, it is controversial to allow software download and update when the car is moving at 100km/h. The DySCAS approach aims to provide necessary run-time support for enabling a systematic and efficient implementation of QoS and dynamic configuration behaviours in automotive systems. The core is a set of MW services that facilitates the sampling of system configuration and operation states, the computation for QoS and dynamic configuration decisions, and the actuations of such decisions.

For a networked system, it is important that the decisions are made based on a consistent global view and actuated in a synchronized way. This in turn necessitates the MW support for consolidating distributed information in regards to vehicle conditions, application states, operation events, and resource availability, as well as the support for disseminating the consolidation results. Another important aspect underlying the embedded decisions is the incorporation of offline design decisions, specifying the set-points of QoS control, the variability of functional and operational dependencies, and the design and technology constraints on reconfiguration and software upgrade. The assumption in DySCAS is that the system developers will derive such information based on the system architecture specification. The MW provides support for tracking and maintaining such information at system runtime (e.g., as component meta-data). To validate the MW support and promote dynamic configuration in automotive systems, the DySCAS project is currently investigating algorithms for resolving the architectural dependencies and change impacts at system run-time.

The move from static system to dynamic configurations is a large step for the automotive industry. The introduction of adaptive aspects of configuration and behaviours, and the ability to defer part of the configuration decisions and V&V efforts beyond the point of systems deployment, call for enhanced support for error detection, error handling, and error repair. The intended support is related to the control and coordination of related operations such as: monitoring the system execution, planning for rollback and producing checkpoints, transferring and recovering component states, re-flashing a node and controlling the execution states of application software and devices.

3. Background and related work

3.1 Middleware and Control

Over the years, a wide range of MW technologies has been developed for distributed computer systems, targeting various application areas. MW technologies for distributed software focus on the deployment and integration of independently developed software components, while emphasizing the support for scalable and dynamic configuration. MW in this direction includes for example EJB [6], DOT NET [7], CORBA and CCM [8]. Many of such MW products are not suitable for embedded systems due to their memory and performance overheads or the lack of support for real-time, data consistency, and fault-tolerance. To overcome these shortcomings, many MW solutions for real-time and embedded systems have been developed, including HADES [9], ARMADA [10], and the RT-CORBA implementations TAO [11] and ZEN [12]. In the domains of sensor networks, ubiquitous and networked embedded systems, there are also MW technologies developed to support advanced dynamic configuration and automated software maintenance. MW in this direction includes 2K [13] and RUNES [14], applying meta-object protocols and reflection for run-time inspection and adaptation of configuration and behaviours [15], QoS control for optimal performance and reliability when the availability of resources changes [16]. There are also many dedicated MW efforts targeting particular aspects or application domains of dynamic configuration, such as the Jini network technology [17] for the plug-and-play of non-real-time services and devices, the HAVi [18] software architecture for the configuration and interoperation of home networks, the OSGi architecture [19] for coordinated development, deployment and management of network services, and the Simplex architecture [20] for the online upgrade of automatic control software. One formal approach is given by the component framework Lusceta

[21], providing not only a QoS-aware reflective MW, but also formalisms for specifying, simulating, analyzing, and run-time synthesizing QoS management.

Due to the lack of compatibility in regards to the automotive specific standards and technologies in respect to system specification and implementation, existing MW solutions will not be suitable as base technologies on which the DySCAS MW system can be built. Nevertheless, these generic solutions together provide a reference source for the design of the DySCAS architecture in regards to MW structuring, fault-tolerance, and execution control. One important basis for the DySCAS architecture is the AUTOSAR (AUTomotive Open System ARchitecture) [5] standard. It provides a domain specific approach to the specification and management of application software components, system services and run-time environment, and the overall system configuration integrating application and system resources. While AUTOSAR is limited to static configuration, the standardized architecture framework and modelling support constitutes a very important basis for DySCAS to understand and specify the application and platform characteristics.

3.2 Policy-based configuration

In DySCAS, policy-based computing is used to achieve the dynamic configuration of the automotive MW. Policies provide a powerful means of representing the logic required to make decisions which is decoupled from the underlying deployed code. Policies are flexible and can be formalised by using a closed grammar described in a formal notation such as EBNF or a schema definition language. A suitably expressive language enables a wide range of behaviour to be represented at high level by a relatively simple policy description. Policies can also be used at a lower, more-detailed level if required.

Policy technologies usually provide general guiding strategies or can only be changed between executions [22, 23]. Ponder [24] permits run-time changeable security policies and has a very feature rich and extensible grammar. It is suitable for use in self-adapting policy-based security software but may not be as well suited for applications with other requirements. Ponder has been used in [25] for the management of differentiated services networks.

AGILE [26] is the policy language used in DySCAS. A policy script can be loaded into an application at run-time to change the behaviour of the application at the point where the script is inserted. The scripts are loaded and processed by an AGILE library instance. The DySCAS MW supports run-time adaptability though the use of AGILE-Lite; a lightweight, embedded version of AGILE [27, 28].

The points at which decision logic can be changed, called Decision Points (DPs), are specified at design time; [29] provides a detailed explanation of DPs and the supporting mechanisms. Policy scripts can be loaded into these points (usually when the component containing the DP is initialised) and can replaced with other policy scripts during run-time, yielding different or more advanced decisions.

The AGILE language has a level of flexibility more normally associated with a lower-level programming language. For example, indirect addressing is supported at the policy script level, so that all constructs can be dynamically configured by changing the parameter variables supplied. Consider ‘rule’ constructs which can be used to implement Boolean logic as well as simple conditional tests. It is possible to use the outcome of one rule to contextually change the behaviour of another rule by changing the actual parameters (not simply the values) compared in the second rule. Other dynamically configurable constructs include:

1. Utility Functions. A simple tool for choosing a path based on which of the provided options has highest utility value in a given context. The utility of an option is obtained as the sum of the products of each *term* and *weight* pair (*terms* represent sensed environmental conditions, *weights* represent the relative importance of each *term*). The action associated with the option that has the highest utility is performed.
2. Tolerance Range Checks. A TRC checks a variable against upper and lower bound conditions (specified as a centre value and an acceptable range value; both dynamically configurable). A three-way fork is implemented, with separate actions performed in each case of above upper bound, within bounds, and below lower bound. This is a simple and effective way to restrict certain actions from being performed unless a condition is violated by more than a configurable margin, and is thus particularly useful in hysteresis control.

Although powerful, AGILE policies at the same time can define functionality at a high level so that developers can focus on the intended business logic and need not be experts in concepts such as autonomics and policy-based configuration. A policy can be a statement of intent, without having to describe exactly how the behaviour is achieved, (since the lower-level mechanisms are pre-built). An AGILE policy editing tool further simplifies the task of preparing policy scripts; making script editing less problematic and less error-prone, see also [26, 30, 31].

Policy languages currently do not support ‘learning’ in the AI sense, which in any case would be too big a leap for automotive embedded systems which represent a highly safety-critical domain. However, the AGILE language does provide a means of persisting adapted policy state (representing user preferences, contextually-informed decisions etc.) in the form of a ‘template’ which can be used to initiate other instances of the policy (i.e. in other DPs, or in the same DP at a later time). This mechanism allows long term adaptation, spanning many evaluations of a policy.

4. The DySCAS Middleware

The architecture constitutes an overall design for the

intended DySCAS MW system where various policy-driven self-management mechanisms are defined, integrated, and realized in an automotive context. Figure 1 provides an outline of the DySCAS MW architecture, including the major MW services and the external interfaces towards the application software and target platform. Above the portability layer, the MW 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 [32].

DySCAS MW 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 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, interoperability, transparent communication, concurrency control, membership management, much as the support offered by other traditional MW systems. These services are optional as the support can be obtained through systems, network, or other external MW. Under these circumstances, the components implementing such services act as wrappers/containers for the corresponding external services.

Across the architecture layers, there are three paths of control: 1. context monitoring and event detection path; 2. reasoning and decision path; and 3. actuation and synchronisation path. 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 monitors the events/states of interest and consolidates the information into a consistent context definition. This path is data intensive and runs from the system platform to the Resource Deployment Management Service (RDMS) 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 dissemination of the context information is supported based on the publish/subscribe paradigm. In each MW service component, there is a context management proxy responsible for subscribing necessary context information published by local or remote RDMS, preprocessing the obtained context information such as into normalized quality figures, and triggering the computation or decision modules in the case of context change.

The reasoning and decision path starts when a context change 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 configuration rules, including the allowed variability and constraints. This provides a system with the ability to reason about the correctness and efficiency of its current state, and to plan for changes without eroding the architecture or violating the functionality and dependability (e.g., safety, security, and availability). In DySCAS, this path is subdivided into: 1. a dynamic configuration control path, supported by the Autonomic Configuration Management Service (ACMS), and 2. a dependability and QoS path, supported by the Dependability & Quality Management Service (DQMS).

Of great importance to the DySCAS MW 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. Major MW services in this path include the Autonomic Configuration Handler (ACH), the Resource Deployment Management Service (RDMS), the SW Load Management Service (SLMS), as well as other services in the Instantiation Interface providing support for the portability and system interaction. Through this path, each dynamic configuration decision (e.g., updating a software component) is refined first into a set of elementary operations (e.g., invoking transferring states, unloading, loading, initializing, and executing a software component) and then into more primitive operations in terms of device and system service invocations. The actuation and synchronisation path also provides the scheduling and triggering support for the actuations across a distributed platform. During the executions, the status is frequently monitored. A failed execution may cause rescheduling of the operations or revising of configurational decisions.

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 MW 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 as 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 MW service can act as a proxy for consolidating a global system view or for obtaining system-wide decisions. For performance reasons, the DySCAS MW 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.

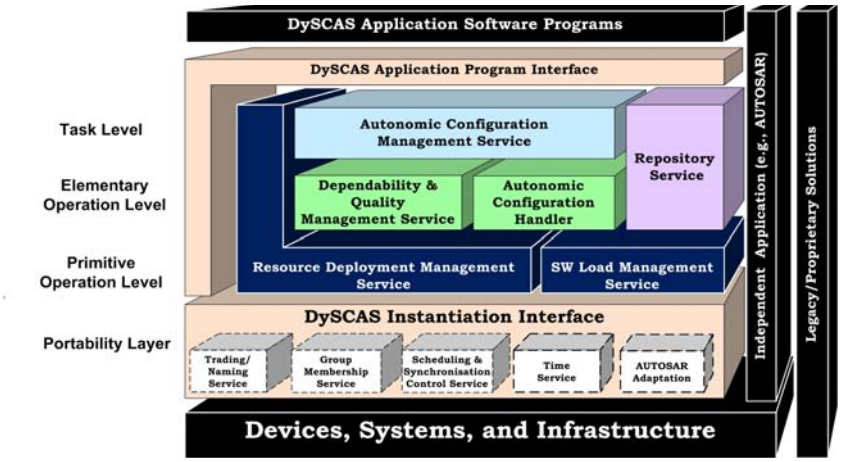

Figure 1. A schematic overview of the DySCAS architecture.

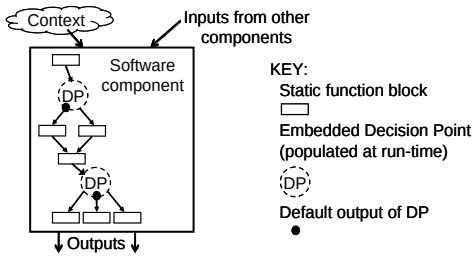

Figure 2. A software component with multiple DPs

5. The DySCAS Dynamic Component Model

The application and resource flexibility in a DySCAS system is realized using techniques inspired by the policy-based computing paradigm. In contrast to those approaches that only provide change from one entire system configuration to another, this method allows small changes to occur independently at various points throughout the system. We describe a methodology for very flexible configuration of software post-deployment and without the need to compile new code or even restart the system.

As explained in section 4, the configuration 'intelligence' needs to be distributed across components. DySCAS achieves this in three key ways: A flexible, designed-for-purpose MW; a versatile component model which supports dynamic mapping of components' context requirements and dynamic mapping of the executional behaviours of components (e.g. start-up initialization and ordering, operation modes); and policy-based configuration, in which each component (in the MW as well as application software) can include a number of policies which can be easily upgraded without changes to the deployed code.

The architecture has been designed with dynamic self-configuration as a core-feature and not a bolt-on. The fundamental concept is that each software component that makes up the MW itself, or applications that operate at the next level, can embed zero or more DPs. Each DP can be dynamically configured by loading a policy at run time. If a component has multiple DPs each operates independently having its own policy and context requirements.

The DySCAS dynamic component model describes several design, operational and interaction aspects of software components in the DySCAS system, to support the required flexibility with respect to deferred behaviour and dynamic configuration. The model specifies:

- A method for design-time embedding of DPs into software components;
- Run-time support for the operation of DPs;
- Support for multiple independent DPs per software component;
- The ability to specify default behaviour per DP, for example if a policy is not loaded;
- A mechanism to dynamically load the appropriate policy into a DP from an on-system repository;
- A mechanism to dynamically replace a policy with a new version;
- A mechanism to automatically map the required context information to each DP;
- The Repository Service (RS), which provides the appropriate policy for a DP on demand;
- The Context Management Service (CS), which provides the required context information to policies within DPs;
- The Dynamic Wrapper mechanism (DW) which facilitates automatic handling of faults arising from the operation of dynamic configuration.

The basic approach is to identify, at design time, places in the software where dynamically changeable behaviour is appropriate. At each of such points a DP is embedded into the compiled code, marking out the possibilities for reconfiguration after deployment. The way which a decision is made (the logic) is not statically compiled into the DP; it is specified in a policy which is loaded (in the form of a data file, via an API method) into the DP at run-time. This separation of decision logic from the compiled code is an advanced method of policy based computing and is the key to post-deployment reconfiguration in DySCAS. Figure 2 shows the design-time view of a software component with a combination of multiple DPs and statically compiled functional blocks.

A key characteristic of computing systems in the automotive domain is the difficulty in making changes post-deployment during a long vehicle lifecycle. Under current practice this can only be achieved by directly servicing each vehicle by suitability qualified personnel with specific equipment. DPs allow flexible run-time configuration of software components, at any time that policies can be loaded into the system. This can be physically, a user

transfers new policies to the vehicle using a storage medium, or directly to a wireless enabled vehicle. This capability not only allows the changes to be made to correct possible problems but also to make use of new devices and information.

The DP concept also future proofs applications. There are circumstances where a developer is aware that future enhancements to behaviour will be necessary, but is not aware of the details at the time of application deployment. In such cases DPs can be embedded at the appropriate places in the logic and very simple policies can be provided initially which can be replaced with more-sophisticated logic when necessary.

5.1 Dynamic context management

The fact that post-deployment configuration changes are supported implies that in general the context requirements of a DP's policy are not known at the time of software deployment (because the policy logic itself is deferred). It would be possible to design a system in which the context information available to a particular DP were fixed when the DP was inserted. For example a DP which controls cabin temperature can be known to need '*current temperature*' and '*current desired temperature*' as input context even before the actual policy logic which controls the temperature is provided. However, such a fixed approach to context provision limits any future policy logic to reasoning based on the same context information types. A new use-scenario in which the cabin temperature control is to take into account the state of alertness of the driver (automatically cooling the cabin when the driver's control inputs are sensed to be less precise – indicating drowsiness) can only be supported if the additional context information can be provided to the new policy. Dynamic context mapping also helps to facilitate independent upgrade of functionally dependent components. Consider a scenario where component A generates context information consumed by component B (for simplicity assume that each component has a single DP, i.e. a single policy operates in each component). An upgrade of component A may result in context information being produced to a different precision, in a different format, or in the generation of new context information not previously available. In a fixed context-mapped system such changes cause component incompatibilities and thus component upgrades ripple through the system. However, if context mapping is dynamic, only the policy for component B need be changed (without changing the code) thus keeping the number of component changes low and thus simplifying the change management process which in turn enhances reliability whilst reducing change and testing costs.

However, dynamic context mapping introduces several new challenges which include: how to dynamically identify and match together the appropriate context producers and consumers; whether to directly couple, or decouple the consumer-producer component pairings, how to maintain low interaction intensity between components (and thus to keep communications overheads and complexity low), and how to handle a 'Context Not Available' situation experienced at a consumer.

To achieve dynamic context mapping, and to automatically handle the challenges identified above, the DySCAS component model includes a pair of mechanisms: the Context Manager Service (CS); and the Dynamic Wrapper (DW) - see section 5.3.2.

The CS is itself policy-configurable so that it can be made to operate in different modes as required by an implementer, and its behaviour can be dynamically changed if required. The default operation model is publish-subscribe-consume. A component that requires a particular context item issues a subscribe request to the CS, which updates a *subscriber table* of current context demands. A component that produces a particular context item publishes it to the CS (which updates a *context information table*). The CS checks to see if the value has changed and if so pushes out the new value to all subscribers. In this way the CS decouples the producer and consumer and also reduces communication and processing at the consumer since context is only pushed to them on a state change basis, and not at the rate that individual samples are produced. The CS also serves as a cache of most recent values which are made available when a new consumer is initiated, thus avoiding the new consumer having to either contact several producers directly, or waiting until all required context values have been pushed to it. Changing the CS policy would enable for example broadcast operation or a hybrid of broadcast and the subscribe approach.

A further key benefit of the CS is that provides transparency to the developers of software components in that a context producing component does not need any design time consideration of consumer components, and a consumer component does not need any design time consideration of what context will be used or where it is generated.

Managing the context information in this way also supports reconfiguration of the location of running software. For instance, an important software component may be shifted from one node to another due to resource availability. The context information required by this component's DPs can be routed to the new location dynamically.

5.2 Behavioural boundaries

The critical nature of the application domain requires careful consideration when allowing configuration changes to be made. During the normal software development procedure, the compiled code (the static function blocks shown in figure 2) can be extensively tested and proven using formal methods. Robustness and fault-tolerance of the methodology is guaranteed by limiting reconfiguration within a software component using two design-time specified constructs. Firstly, the number and location of the

decision points within the component, define the possible changes to the behaviour. Secondly, the dynamic wrapper associated with each DP specifies the behavioural boundaries for that DP, so that any policy-based reconfiguration cannot result in actions beyond these limits (this is explained in more detail in section 5.3.2).

5.3 Middleware integration

The DySCAS MW supports dynamic policy-based reconfiguration by providing a policy evaluation library, AGILE-Lite, and supporting DP and DW mechanisms. Dynamic context management (described above) is also provided to support policy-dependant context requirements.

5.3.1 Decision Point Support

AGILE-Lite provides DP organisation, the functionality to parse and evaluate a policy at run-time and robustness features using the Dynamic Wrapper mechanism. The library is a lightweight implementation for embedded systems, requiring minimal resources to achieve dynamic and robust behaviour (the 'lightest' version to date has a memory footprint of just 34kbytes). It supports an XML grammar based on the AGILE policy language [26]. When requested, AGILE-Lite will evaluate a specified DP and produce a decision result based on the currently loaded policy and current context information.

Internally the library architecture contains three functional layers shown in figure 3:

- Decision Evaluation Module (DEM); parses and evaluates policies, producing a result based on the policy logic and current context.
- Dynamic Wrapper (DW); responsible for silent handling of errors and ensuring a legal result is returned for a evaluation request.
- Decision Point API; provides developer access, including loading and evaluation of policies for specified decision point.

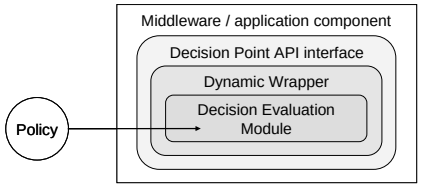

Figure 3. 3-layer architecture of the AGILE-Lite library

5.3.2 The Dynamic Wrapper

As shown in figure 3, the DW forms the interface between the software component (via the API) and the DEM.

The DW is responsible for making context subscription requests to the CS on behalf of its specific DP. A policy's required context inputs (termed 'environment variables') are identified when the policy is parsed by the DEM.

The DW is also responsible for providing mechanisms for basic validation and robustness support. During the policy load operation the DW determines whether:

- An appropriate policy was located and provided by RS,
- The policy script was loaded from file correctly,
- The policy was parsed correctly, and found to be referentially self-consistent,
- The policy meets the DP specification (in particular this requires that the possible logic outputs of the policy match the DP configuration).

In case of any of the above problems, the policy load processes is deemed to be unsuccessful and the wrapper sets up error flags for diagnostic purposes. Where a previous policy had been successfully loaded, automatic rollback to that policy is initiated.

When the DP is invoked (and thus the policy must be evaluated), the DW monitors whether a policy has been loaded in the DP, and that the various policy objects that constitute the policy logic have each been instantiated correctly (for example there was sufficient memory allocated). The policy objects include the types: Policy, Template, Variable, Action, Rule, UtilityFunction, ToleranceRangeCheck, and ReturnValue. Additionally, through the context subscription mechanism, the DW determines if a context-not-available situation arises – which would prevent the policy logic reaching the correct outcome.

If any of the above DP-evaluation tests fail, the DW sets up corresponding error flags and returns the DP-specific 'default' value, which is pre-defined by the software developer. Thus the DW ensures that, should any run-time problem occur related to the dynamic policy configuration aspect, the affected DP collapses down to statically defined and thus totally predictable and verifiable behaviour.

Through its silent error handling (i.e. by trapping errors generated in the DEM, and returning a predefined 'legal' return value to the component), the DW makes a significant contribution to system robustness and safety. The implementation of dynamically configurable components has been achieved in a manner which can be only advantageous in comparison to static components and will never decrease the system stability and integrity. From a component developer point of view, the DW is entirely transparent, because it works silently and cannot cause component failure. The only way to detect its intervention is to check whether error flags were set or not. In general, a developer can simply use the decision result produced by the DP in their code.

5.3.3 The DP API

During the design of a policy-configurable component, the developer identifies those points at which dynamic behaviour is required. For each of these points one DP should be created. At DP instantiation it is required to specify default outcome (in case of dynamic evaluation failure) and all other valid outputs. The developer should then decide how the component will behave in response to the policy decision. To complete each DP, an AGILE policy

script is written and stored in the RS, to be loaded at run-time. When the component is executed and a DP is evaluated, a decision is made using this policy and the current context information. The decision then becomes a data item within the component and processed as coded by the developer. The DW ensures that this decision is either the default value or one of the other supplied valid outputs. This transparency removes the burden of writing any special error handling code for the DP.

The API functions make the three layer library structure transparent to the component developer. If the component developer is not interested in errors arising during policy loading or decision evaluation and chooses not to test or handle any error flag set up silently by dynamic wrapper, it is actually indistinguishable whether the component evaluation returned and actual policy decision or the default value predefined at DP instantiation. This transparency simplifies the use of the library and guarantees that the worst possible outcome of a failure in dynamic decision evaluation is safe, static and non context-aware behaviour of a component. A DP can be embedded and operational as a deferred logic place holder, prior to providing a policy.

6. Automatic internal reconfiguration

DySCAS configuration is implicitly dynamic, but it can be triggered in a variety of ways. Policy-based configuration of components is usually performed each time the component is initialized. New DP-specific policies can also be loaded into the system between, or during run-times. A new policy may be loaded *'pulled'* because a user requests a specific upgrade, or *'pushed'* during an annual service or may be automatically loaded 'in the field' perhaps delivered from the vehicle manufacturer's back-end systems to the vehicle via a wireless network hotspot.

However, additional forms of automatic dynamic configuration are directly supported directly by the MW. In particular it is necessary to support load balancing to ensure efficient use of resources within the vehicle, and in some cases the resources of temporarily attached user devices. To run applications or tasks for example of the vehicle infotainment system more efficiently, a migration to the additional mobile device makes sense to use its unused resources. Thus it is possible to migrate for example tasks of the navigation system to a connected PDA for faster and more detailed map rendering and more optimal calculation of routing information.

After the connection of the PDA to the vehicle infotainment network with the aid of standardized interfaces like Bluetooth or WLAN, the device is discovered and the appropriate device information, locally running processes, and device and network resources are registered by a dedicated service.

In consideration of all running processes and the resources situation within the vehicle infotainment system appropriate services decide on a possible load balancing according to strategies; based on characteristics and parameters of the system and tasks, and initiate the task migration where required. Thus the appropriate navigation system tasks migrate from the navigation system to the PDA. After the calculation the results of the tasks are sent back to the navigation system, where they are used.

To realize the use case scenario described above the MW architecture is required to fulfill several requirements:

- Event management – Added devices and device removal have to be discovered by the vehicle infrastructure.
- Device registration - Detailed information and capabilities of the newly added devices have to be registered.
- Resource management - Status information and resource load of each device (ECU) have to be known.
- Load balancing - Potential task migrations have to be initiated based on strategies which take into consideration characteristics and parameters of the system and tasks.

A detailed description of these four tasks within the automotive MW architecture is provided in [33] and [34]. For the load balancing we use a cost based strategy. The Load Balancer evaluates possible migration of tasks to the additional device. Migration is only a useful option if the cost of migrating is lower than the cost of keeping tasks with their original device. The cost benefit ratio for tasks of busy devices is computed which helps the Load Balancer to form the decision of whether to migrate or not.

The calculation of migration costs of tasks is realized according to the priority list of the *most loaded* strategy (i.e., tasks with a high load have a higher priority). This enhancement enables dynamic load balancing and is therefore a basis for self-optimisation in a vehicle network.

The Load Balancing mechanism is automatically invoked, and it can also be internally policy-configured for additional flexibility – for example several different load balancing algorithms can be simultaneously deployed within the same component each optimal under different circumstances. A policy is used to contextually select which of the fixed algorithms is the most appropriate at a given instant.

7. Conclusion

The research issues addressed by the DySCAS project are quite challenging. The automotive domain traditionally uses static functionality because of issues which include strict real-time performance and the safety critical nature of the systems. Thus whilst the potential advantages of introducing dynamic configuration and context-aware behaviour are very high, opening up whole realms of future use-cases, the accompanying technical requirements are very demanding. The project is still ongoing and presented here are partial solutions towards the desired dynamically configurable systems. This paper has described some of the novel and technically advanced aspects of the automotive MW, focusing on the architecture and the software component model. In combination these two aspects permit very flexible dynamic configuration and context-awareness by

distributing the configurational intelligence across the various components as required. A dynamic context mapping service decouples components so that component upgrades can be performed in isolation. The strong obligations of robustness, validation and verification are met by wrapping the dynamic configuration mechanism with an automatic fault-handling mechanism which silently downgrades a problem component to static default behaviour.

Acknowledgements

The DySCAS project is funded within the 6th framework program "Information Society Technologies" of the European Commission. Project number: FP6-IST-2006-034904. The project website provides further details [4].

References

[1] K-E. Årzén, A. Cervin, T. Abdelzahler, H. Hjalmarsson, A. Robertsson, Roadmap on Control of RealTime Computing System, EU/IST FP6 ARTIST NoE, Control for Embedded Systems Cluster.

[2] Auto Catalog, Pass 2000, McKinsey & Company.

[3] M. Törngren, D. Chen, D. Malvius, J. Axelsson. Chapter - Model based development of automotive embedded systems. Automotive Embedded Systems Handbook. Editors Nicolas Navet and Francoise Simonot-Lion. Taylor and Francis, CRC Press - Series: Industrial Information Technology. 2008.

[4] DySCAS project website: http://www.DySCAS.org

[5] AUTOSAR initiative: www.autosar.org

[6] Sun Developer Network. Enterprise JavaBeans Technology. Available <http://java.sun.com/products/ejb/index.jsp>

[7] Microsoft. Microsoft .Net. Available <http://www.microsoft.com/net/default.mspx>

[8] Object Management Group. CORBA 3.0. Available <http://www.omg.org/ technology/documents/formal/corba_2.htm>

[9] E. Anceaume, G. Cabillic, P. Chevochot, I. Puaut, HADES: A Middleware Support for Distributed Safety-Critical Real-Time Applications, Intl. Conf. on Dist. Computing Systems, 1998.

[10] T. Abdelzaher, M. Bjorklund, S. Dawson, W.-C. Feng, F. Jahanian, S. Johnson, P. Marron, A. Mehra, T. Mitton, et al., ARMADA Middleware and Communication Services, Real-Time Systems, 1997.

[11] C. Gill, J. M. Gossett, D. Corman, J. P. Loyall, R. E. Schantz, M. Atighetchi, and D. C. Schmidt, Integrated Adaptive QoS Management in Middleware: An Empirical Case Study, 10th Real-time Technology and Application Symp., May, 2004, Toronto.

[12] R. Klefstad, D. C. Schmidt, and C. O'Ryan. The Design of a Real-Time CORBA ORB using Real-Time Java, Proc. IEEE Intl. Symp. Object-Oriented Real-Time Dist. Computing, April 2002.

[13] F. Kon, J. R. Marques, T. Yamane, R. H. Campbell, and M. D. Mickunas. Design, Implementation, and Performance of an Automatic Configuration Service for Distributed Component Systems. Software: Practice and Experience, 2005.

[14] C. Mascolo, S. Zachariadis, G. Pietro Picco, P. Costa, G. Blair, N. Bencomo, G. Coulson, P. Okanda, T. Sivaharan. Runes Middleware Architecture. D5.2.1, RUNES Project. RUNES/D5.2.1/PU1/v1.7, FP6. Inf. Society Technologies. EC.

[15] P. Cointe, editor. Meta-Level Architectures and Reflection: 2nd International Conference, Reflection '99, St. Malo, France, volume 1616 of Lecture Notes in Comp. Science. Springer, 1999.

[16] D. C. Schmidt, Adaptive middleware: Middleware for real-time and embedded systems Communications of the ACM, Volume 45 Issue 6, June 2002.

[17] SUN Microsystems, Jini Network Technology, <http://www.sun.com/software/jini/>

[18] HAVi. Home Audio Video interoperability. http://www.havi.org.

[19] OSGi Alliance: The OSGi service platform – dynamic services and networked devices. http://www.osgi.org

[20] L. Sha, R. Rajkumar, M. Gagliardi. Evolving Dependable Real-time Systems. Proc. IEEE Aerospace Conference, 1996.

[21] L. Blair, G. Blair, A. Andersen and T. Jones. Formal Support for Dynamic QoS Management in the Development of Open Component-based Distributed Systems. IEE Proceedings Software. Vol. 148 No. 3. June 2001.

[22] IBM. Policy technologies. http://www.research.ibm.com/policytechnologies/.

[23] O. Ronen and R. Allen. Autonomic policy creation with singlestep unity. Proc. 2nd Intl. Conf. on Automatic Computing, pp. 353-355, Seattle, USA, 2005. IEEE Computer Society.

[24] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. In POLICY '01: Proc. of the International Workshop on Policies for Distributed Systems and Networks, pp. 18-38, London, UK, 2001.Springer-Verlag.

[25] L. Lymberopoulos, E. Lupu, and M. Sloman. An adaptive policy-based framework for network services management. Jnl. Netw. Syst. Management., 11(3), pp. 277-303, 2003.

[26] R. J. Anthony. The agile policy expression language for autonomic systems. ITSSA, 1(4): 381-398, 2006.

[27] R. Anthony and C. Ekelin. Policy-driven self-management for an automotive middleware. In proc. 1st Intl. Workshop on Policy-Based Autonomic Computing (PBAC), At ICAC '07: 4th Intl. Conf. on Autonomic Computing, Florida, USA, June 2007.

[28] R. Anthony, A. Rettberg, I. Jahnich, M. Torngren, D. Chen, and C. Ekelin. Towards a dynamically reconfigurable automotive control system architecture. In International Embedded Systems Symposium, Irvine, CA, USA, May 2007. IFIP.

[29] P. Ward, M. Pelc, J. Hawthorne and R. Anthony. Embedding Dynamic Behaviour into a Self-Configuring Software System (*To appear*) In Proc. 5th Intl. Conf. on Autonomic and Trusted Computing, Oslo, Norway, June 2008.

[30] R. Anthony. Policy autonomics website. http://www.policyautonomics.net/.

[31] R. J. Anthony. Policy-centric integration and dynamic composition of autonomic computing techniques. In ICAC '07: Proc. 4th Intl. Conf. on Autonomic Computing, Jacksonville, Florida, USA, June 2007, IEEE Computer Society.

[32] J.S. Albus, F.G. Proctor, "A Reference Model Architecture for Intelligent Hybrid Control Systems", Proc. Intl. Federation of Automatic Control, USA, 1996.

[33] I. Jahnich and A. Rettberg. Towards Dynamic Load Balancing for Distributed Embedded Automotive Systems. In: A. Rettberg, M. Zanella, R. Dömer, A. Gerstlauer, F. Rammig, (Editors) Embedded System Design: Topics Techniques and Trends, Irvine, DA, USA, May 2007.

[34] I. Jahnich, I. Podolski, A. Rettberg. Integrating Dynamic Load Balancing into the Car-Network. In 4th Proc. Electronic Design, Test and Application (DELTA 2008), Hong Kong, January 2008.