Policy-driven self-management for an automotive middleware

Richard John Anthony
Department of Computer Science, University of Greenwich, Greenwich, London, UK
R.J.Anthony@gre.ac.uk

Cecilia Ekelin
Volvo Technology Corporation, Sven Hultins Gata 9C, Göteborg, Sweden
Cecilia.ekelin@volvo.com

Abstract

This paper describes work towards the deployment of self-managing capabilities into an advanced middleware for automotive systems. The middleware will support a range of futuristic use-cases requiring context-awareness and dynamic system configuration.

Several use-cases are described and their specific context-awareness requirements identified. The discussion is accompanied by a justification for the selection of policy-based computing as the autonitics technique to drive the self-management. The specific policy technology to be deployed is described briefly, with a focus on its specific features that are of direct relevance to the middleware project.

A selected use-case is explored in depth to illustrate the extent of dynamic behaviour achievable in the proposed middleware architecture, which is composed of several policy-configured services.

An early demonstration application which facilitates concept evaluation is presented and a sequence of typical device-discovery events is worked through.

1. Introduction and background

The work presented in this paper forms part of a substantial project to design and develop an advanced self-managing middleware for automotive systems.

The project targets a suite of futuristic use-cases with high demand for context-awareness and which collectively represent a very wide behaviour space. Because of this diversity, and depending on the specific use-case and execution context, the system may need to perform any or all of: self-configure, heal, optimise or protect itself, in various ways and at various logical levels.

Many different techniques can be used to implement self management. After detailed investigation, policy-based computing has been selected as the core self-management technology. The reasons for this are discussed in section 3.

A detailed review of the state of practice in policy-based computing has been provided in [1]. The various technologies can be differentiated in two key ways.

The first of these is based on the meaning ascribed to the term ‘policy’. In a significant proportion of policy deployments the actual policy logic (the rules) is statically embedded into the application code. During run-time (or at initiation) configuration parameters (such as counts, thresholds and flags) are passed in. [1] defines this form of configuration as a template; and this is the definition used in the remainder of this paper. Typical examples are described in [2,3].

Template configuration is 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.

The other common implementation approach to policy-based computing is to store the configuration values and the policy rules externally to the application code. This is the meaning implied by ‘policy-configuration’ in the remainder of this paper. This approach is much more flexible as the actual policy logic can be changed after the deployment of software systems, and in some systems these changes can even occur at run-time.

The second way in which policy technologies can be differentiated is in terms of the flexibility with which policies can be changed. There are several facets to this: 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.
The Policy Description Language (PDL) [4] is an example of the use of statically configured policies. PDL is used to specify policy agents, which perform distributed network management tasks [5].

The Policy Management for Autonomic Computing (PMAC), [6] from IBM, is an example of a policy technology that supports open-loop adaptation. An automated policy management and deployment application is provided to assist manual policy updates. A further tool, the Policy Schedule Advisor assists in refining a policy schedule to ensure efficient execution on the PMAC middleware [7]. Policyscape [8] also supports automated policy creation. It provides a set of building blocks from which more-complex policies can be created.

Ponder [9] supports the use of meta-policies which define semantic constraints on regular policies, thus providing a context-driven behavioural adaptation aspect. A security policy implementation described in [10] uses a meta-policy to perform context-aware selection of the most appropriate security policy.

Current policy technologies do not tend to support automatic updates of actual policy logic. Several policy schemes are self-adaptive to the extent that they automatically deal with conflicts that arise. Examples include [11] which detects and resolves clashes between the obligations of security policies, and [12] which supports automatic detection and resolution of rule conflicts. However, run-time policy loading, and dynamic updating are necessary abilities for real-time systems in which several policy-based components interact; such as in the proposed DySCAS middleware. In such a system it is not realistic to require that policies are only loaded or updated at initialisation; due to component dependencies it might not always be possible to restart a component so that a new policy can be loaded. The system must be able to dynamically configure itself in real-time to handle unforeseen events such as new external-device attachment, component failures and changes in driver preferences for infotainment services.

2. The project: goals and challenges for self-management

DySCAS (Dynamically Self-configuring Automotive Systems) is an European Commission sponsored project which aims to introduce self-managing behaviours into automotive control systems so that they can be dynamically configured depending on environmental conditions and to handle unexpected events, new use-cases and types of external device not known at deployment time.

Current and in-development automotive standards, target static configurations; AUTOSAR (AUTomotive Open System ARchitecture) being the most prominent of these [13]. Such systems require that important system aspects such as operational modes and use-cases are fixed at the point of system deployment. Whilst the introduction of such standards is very welcome in the automotive industry (allowing re-use of components and services across different manufacturers for example); the static aspects are problematic for dynamic environments.

DySCAS intends to build on such standards to enable dynamic self-adaptive behaviour, whilst interoperating or at least co-existing with certain deployed static technologies.

The fundamental architectural approach that has been adopted for DySCAS can be described as a specialised real-time middleware that enables run-time configuration of its component services. There are several motivations for the self-management focus, these include:

1. The need to automatically deal with unexpected events, including the failure of sensors or Electronic Control Units (ECUs). Given that modern vehicles contain many ECUs and that some services have higher importance than others; individual hardware failures should no longer directly imply that particular services should cease to function. High priority services should be automatically relocated, possibly at the cost of dynamically selecting low priority services to shut down.

2. The need to dynamically reorganise and balance workloads to maximise the efficiency with which resources are used, since resource constraints are a major deciding factor in the extent of functionality (i.e. the number of services that can co-exist).

3. The need to enable users to interconnect their various intelligent devices that hold data and can provide additional services, to their vehicle in easy and flexible ways whilst ensuring data integrity and system security.

4. The need for flexible personalisation of vehicles; both from the driver’s perspective and also from the owner’s perspective (for example in the case of a fleet of lorries, coaches etc. which need to have customised but fleet-consistent infotainment services).

5. The possibility to adapt to support new use-cases (and extensions to use-cases) that were not perceived at the time of vehicle deployment. For example a new type of device, or a known device type but offering a new type of service, attempts to connect to the system.

6. The need for reliable field-upgrades of software to enable the latest innovations to be added post manufacture.

The platforms on which DySCAS will operate (i.e. ECUs in vehicles) have several restrictions which increase the challenge of deploying self-management. These restrictions include: limited processing and
memory (many ECUs have only 8-bit processors which run at low clock speeds and have limited memory addressing capabilities, although vehicles are likely to have some 32-bit ECUs as well); the typical communication busses used in automotive systems have restricted data rates (e.g. the popular 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); and the fact that the systems are embedded and thus it is difficult to change the run-time code (operating system and application code).

3. Selection of an autonomics technology

This section first provides an overview of the types of context-aware functionality to be achieved in DySCAS in terms of the generic categories of use-cases that have been identified. It then discusses the overall suitability of policy-based computing as a basis for self-management in the project. This discussion is followed up by describing one specific functionality example from each of the generic categories, identifying the types of run-time decisions that must be made. Finally, special considerations with respect to the use of policies in DySCAS are discussed.

3.1. DySCAS generic use-cases

The use-cases have been carefully chosen so that collectively they cover all of the key novel self-adaptation characteristics of the new middleware and its services (i.e. the use-cases are a specimen, rather than exhaustive, set). At the time of writing, at least 16 specific use-cases (SUC) have been identified, separated into four groups based on a combination of the type of behaviour exhibited, and the type of functionality required of the middleware. The groups are referred to as generic use cases (GUC). The four groups are behaviourally defined as:

GUC1. A new device is attached to the vehicle. The ‘device’ can be a physical object such as a mobile phone, PDA, MP3 player etc, but can also be a wireless network hotspot. The system must perform a series of actions which, depending on the SUC, 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.

GUC2. Integrating new software functionality. The scope covers both operating and application software, and can involve adding totally new components or applications, as well as upgrades of existing components and applications. The use-case might be initiated to perform an update because a problem has been detected with the current software configuration, or because new functionality has become available. The use-case also supports software downgrading, for example, when a time-limited licence has expired for a specific add-on service, or when certain personalised / personalisation services are removed when the vehicle changes 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.

GUC3. Closed reconfiguration. This includes resource allocation, detection of unexpected events, self-diagnosis and graceful degradation in the presence of component failures (hardware or software) or power shortages.

GUC4. Resource optimization. This includes dynamic selection amongst redundant ECUs for reliability and power saving (if unused devices are shutdown). Selection amongst redundant sensors is made for reliability and quality reasons.

3.2. The suitability of policies for DySCAS

As is evident from the above discussion, the generic use cases of the project cover a wide space of behaviour, requiring diverse self-managing actions. Ideally the final architecture should satisfy these GUCs as a minimum set, i.e. it must be extensible into future scenarios not yet fully perceived or understood. Thus flexibility, of the architecture itself and also of the self-management capability of the architecture, is a key concern.

After extensive analysis of the suitability of a range of self-management technologies, taking into account the capabilities of the technologies, the constraints of the deployment environment, and the identified use-cases; it was decided to deploy policies as the fundamental technology for achieving self-management.

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.

Of particular importance for the specific application domain is the fact that policy-based computing does not require that the policy writers are experts in self-managing systems. They only need to be able to express the desired behaviour of the system using the policy language grammar; and even this requirement
will be eased with the eventual introduction of graphically-based policy-development tools.

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 an alternative decision-making technique, such as Utility Functions (UF), to be supported in addition to policies. 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.

3.3. Examples of specific use cases

For each GUC, a specific behavioural example is discussed. These are System Functionalities (SF) associated with the GUC in general (SF are the finest-level of defined behaviours that can be composed to build SUC functionality; SFs are discussed in more detail in section 4).

The discussion highlights the variety of ways in which policy-based computing will be used. The spread of examples across the GUCs illustrates the broad applicability of policies across the whole project.

GUC1.S2: Negotiation and contracting of functionalities. This SF is concerned with the type of device that is being connected, what application services it offers and how the device and its services can best be integrated into the vehicle system, if at all. The types of decisions that policies would be employed to deal with include: 1. Deciding whether the connection is allowed; in terms of resources required of the host system (such as power supply, memory, processing etc); in terms of security; and in the context of possible interoperability conflicts. 2. Deciding how to facilitate the connection; for example determining whether a device driver needs to be dynamically installed, and applying security restrictions specific to a particular device. 3. Prioritizing attached devices in terms of their functionalities and importance.

A specific use-case which uses this system functionality is addressed in detail in section 6.

GUC2.2: Choice of software package to be downloaded. This SF is concerned with selecting an appropriate software component for download (for example when upgrading an application), and checking that the download will not cause any system integrity violations. Due to software versioning, component interdependencies and interoperability problems it is not a trivial matter to select the correct component to download. Policy reasoning would include: 1. Determining if a particular reconfiguration is allowable; based on the extent of dependencies on the component and the number of other components with which it interacts. 2. Identification of the most appropriate version of a component to download taking into account the versions of other components. 3. Whether the risks are acceptable; in terms of the ratio of predicted costs (including overheads) and benefits; and in terms of stability (e.g. the probability and cost of roll-back). 4. Whether the previous version should be deleted or retained.

GUC3.S6: Planning new system configurations, taking into account resource restrictions. This SF is concerned with determining whether internal reconfiguration is necessary (especially when a problem or exceptional event has been encountered), and if so how to achieve it. Policy reasoning would include selecting the most appropriate reconfiguration strategy in terms of: urgency, prioritisation (if several reconfigurations are possible), and resource availability. There is also scope for a UF to be used in risk management (evaluating the risk of each of several possible reconfiguration sequences, and the extent of possible introduced problems versus known existing problems).

GUC4.6: Optimisation Intelligence. This SF is concerned with determining ways to dynamically optimise the configuration of the system, in particular in terms of resource usage. Policies will deal with issues such as ensuring an optimisation would not threaten access to critical services or resources. This involves mapping dependencies between applications and services, and predicting or detecting conflicts in the access to resources. UFs could be deployed to determine the optimality of a particular relocation: in terms of the extent of the benefits versus the overheads, the perceived ‘instantaneous’ relative importance of the various applications / services, and the expected permanence of the reconfiguration.

3.4. Special considerations for DySCAS

The use of policy-based self-management in the DySCAS project brings some technical challenges with respect to the use of the policies themselves. There will be multiple policies deployed into multiple components (the services that constitute the middleware). The composition of the various components may itself be policy-configurable to allow the middleware to deal with different scenarios, so there may be a mixture of middleware-controlling and application or use-case specific policies. Additional functionality, including the use of templates to configure policies, and UFs will be required by some of the components. Dynamic updates of policies, and dynamic creation of new templates will be necessary for some use-cases; and it may be necessary to maintain several versions of some policies. Some middleware services will employ hierarchical policies so that the complexity of individual policies stays low. Different policies will be
contributed by various parties: the DySCAS developers; the vehicle manufactures who deploy vehicles; third-party software and application producers; and (possibly) the vehicle owner / driver.

4. A proposed self-managing architecture

This section describes the proposed middleware architecture in overview, and in particular explains the need for embedding self-configuration into multiple middleware services.

GUCs are logical groupings of SUCs; which in turn comprise SFs. SFs are defined as the atomic behaviours that are identified when the SUCs are broken down into fine detail. The SFs are abstracted in the sense that they describe the functional requirements but not how they will be realized. An example of an SF is the function which enables a device (such as a Personal Data Assistant) to connect to the car information systems.

However, the architecture would be inefficient if SUCs and/or SFs were directly mapped to middleware functions. There would be duplication of internal behaviours, because various SF’s derived from different SUCs, (possibly across different GUCs) may describe the same actual functionality. To resolve this, System Requirements (SR) are defined as the set of required functions that the system must support, after the SFs have been normalised and duplicates removed.

The middleware comprises numerous services which actually realize the SRs at a sufficiently fine-granularity such that duplication of functionality is avoided and re-use is maximized. SRs each map onto one or more of these middleware services.

Figure 1 shows how the use-cases and SFs are related, and how the SFs are mapped onto SRs, which in turn map onto the middleware (the part between the horizontal lines).

An example of a middleware service is the Discovery Service which is responsible for recognising the presence of a new device; determining which type of device it is; and what services and/or resources it offers. Other middleware services will be brought into play once a device has been recognised. For example the Security Service must decide whether the device and the application services it offers are to be accepted into the system. Each middleware service needs to be context-aware, thus there is scope for all of them to be policy-configured. The general concept of context-aware services is illustrated in figure 2. The above scenario is explained further in section 6.

Through composition of its various dynamically configurable services the DySCAS middleware facilitates run-time reorganisation to balance workloads, expedite urgent processes, and to reconfigure components for survivability despite hardware failures. Components will also be able to automatically discover new components and establish service level agreements based on the resources and services that they are able to provide to each other.

[14] Provides further information concerning the DySCAS draft architecture, the use cases and services.

5. Using AGILE in DySCAS

The policy implementation is to be based upon the AGILE policy expression language and integration framework [15]. AGILE is particularly suitable for several reasons which include:

1. It enables hierarchical policies; in which meta-policies can dynamically select which of several lower-level policies is currently the most appropriate to use. For example this is needed in services which deal with different types of entity, or work across a wide state-space. The higher-level policy determines the current context and then switches to a specialised policy. This multi-policy approach keeps the complexity of each policy low, and thus promotes extensibility.

2. It has in-built support for integrating UFs directly into policy logic. UFs are particularly useful for making context-sensitive selections amongst several possible options; and their applicability to several use-cases has been identified.

3. It directly supports dynamic self-adaptation of the controlling policy as well as the controlled system. This enables policies to be reconfigured to suit context;
for example automatically skipping certain rules or dynamically adjusting the weights of an embedded UF.

4. It enables adapted policy state to be persisted in the form of newly-created templates. This is ideal, for example when new policy configurations must be dynamically created to handle newly-discovered devices.

5. Templates and policies are both supported and can be used in combination. The various use-cases and middleware services require different levels of policy sophistication. Templates are sufficient when the policy logic is fixed and it is only necessary to pass in configuration parameters such as flags and thresholds dynamically. Most DySCAS components will require that the entire policy logic is held externally so that it can be changed easily. Some components will load both a policy (e.g. to deal with a particular use-case scenario) and a template (e.g. to configure the policy for a specific device or user).

6. Dynamic loading of policies and templates is supported so that behaviour can be changed very flexibly. This is important because it will not always be possible or desirable to shutdown components (because of interdependencies) in order to load a new policy or template.

7. The policy evaluation mechanism is instrumented and supports run-time logging to aid policy diagnostics.

AGILE has been developed in C++ / .NET. The implementation library, documentation (including grammar specification and API usage), sample applications and policies are available at [16].

However, AGILE is not optimised for resource-constrained environments (such as that represented by the DySCAS middleware) so it is intended that a lightweight version, AGILE-Lite be developed. This will have the same flexible grammar as AGILE, but will be designed specifically for deployment in embedded applications. The AGILE-Lite implementation library will be internally optimised for performance and resource efficiency, and will be written in C, as this has high portability and code efficiency.

6. Detailed exploration of behaviour

The dynamic configuration of the middleware (the services within) is illustrated through the exploration of a subset of the behaviours of GUC1. In particular the exploration is concerned with the discovery of a new device and the subsequent decisions, made by several components, to determine whether the device can be attached to the system and if so, which of the device’s application services could / should be used. This functionality cuts across most of the SUCs of GUC1; and has been purposely chosen because it involves a collection of policy-configured middleware services that each use policies in different ways.

At the time of writing, the project has a remaining 20 months to run. Thus the design specification is subject to finalisation. The goal of this section is to illustrate the types of dynamic behaviour that the system will exhibit in dealing with a particular use-scenario.

6.1 Service composition

Consider that a new device is brought into wireless communication range of the vehicle. The middleware needs to dynamically decide whether the device can be attached to the system, how this can be achieved, and what application services offered by the device are to be used. Each of these aspects requires that a number of context-aware decisions are made. The composition of middleware services required in this case is illustrated (in simplified form) in figure 3. The context information available to each middleware service, and the interactions that occur between the various services are also shown.

Four middleware services are involved for the purpose of the demonstration presented in this section. Some peripheral interactions with additional middleware services are not included for brevity and clarity. These other services would provide some aspects of context which are assumed available in the demonstration. The middleware services are discussed in the order that they are invoked.

The Discovery Service. The category and type of device must be determined, as well as the application services it offers. Categories include wireless network and physical device. Types include mobile phone, satellite navigation equipment, PDA, auto-toll (such as at a bridge or toll road), and wireless network hotspot. Application services include music streaming, auto-toll-payment, Internet access, and satellite navigation. Devices not conforming to known values of category, type and services, or advertising an illegal combination (e.g. an auto-toll is not permitted to provide Internet access) will be rejected by the discovery service.

The Resource Management Service. This is responsible for determining whether the system has sufficient resources to support a new device / service.

The Security Service. In this scenario, the security service must determine whether the device is allowed to be attached to the vehicle’s information system taking into account details such as whether the specific device is already known or not, and whether or not it has been authorized by the current driver / user.

The Dynamic Prioritization Service. This determines which application services should be supported when there are insufficient resources to support all currently requested services. A particular need for this middleware service is when an ECU that
is running a high priority application service fails, requiring that some other application services be shutdown so that the high priority application service can be re-established.

6.2 The policies

Each middleware service embeds the policy library and loads a policy at run-time. The services each demonstrate different aspects of AGILE’s functionality.

In the demonstration of the new-device scenario, the behaviour of all of the middleware services have been simplified to permit accessible demonstration of the key dynamic configuration concepts. In turn, the policies have been implemented with only the functionality required for the specific use-scenario.

The discovery service policy demonstrates the use of hierarchical policies. The policy script contains two levels of policy, as depicted in figure 3. The top-level policy determines the category of the new device, currently one of WirelessNetwork, PhysicalDevice or Unknown. A category-specific policy is then invoked or, if the category is ‘unknown’ the top-level policy rejects the device. This approach keeps the actual logic of each policy quite simple and enables the component’s behaviour to be extended without changing its deployed code. For example it is easy to accommodate a completely new device category by changing the top-level policy and providing a new category policy.

The resource management service uses a single policy which is contextualised by the level of resources in the system, the number of application services already running (and their resource usage), and whether external processing (e.g. on powerful attached devices such as laptop computers) is permitted.

The security service demonstrates the combined use of policies and templates, as well as run-time updating and persisting of state via loading and saving templates. The security policy is configured by a device-specific template that provides authorisation status. If the device is not known to the system, a default template is used. For devices that have not been pre-authorised (including new devices) the user is prompted to authorise (or not) the device. For new devices the authorisation decision is persisted in the form of a new device-specific template. If a user answers Accept ALWAYS or Reject ALWAYS for a previously known device, its template is re-written so that the decision is remembered and the security policy does not prompt for this information again.

The dynamic prioritization service uses both policies and templates, in essentially the same way as the security service, to maintain the user priorities for the various application services offered by devices. For example a user might decide that the navigation application on her PDA is a high-priority service whilst its music-streaming capability is low-priority.

Run-time prioritisation of application services is a...
complex combinatorial problem. This arises from the many different service-types and instances (each with user-assigned priorities), hosted on many different devices, and is subject to resource availability and current services running (for example the benefit of a music service is temporarily depreciated if another music-steaming service is already active). The problem is difficult to express using policy logic, but UFs are ideally suited to such optimizations. Thus a UF is embedded in the policy.

6.3 The demonstration application

A middleware-simulation application has been developed to demonstrate and explore the types of decision behaviour that will be exhibited by the final system. Each middleware service has been built as a separate component and each embeds the real AGILE version 1.2 policy library. The middleware services have been implemented according to the current (outline) design specification of the target system. The services load real AGILE policies which are actually executed and drive the behaviour of the application in the same way as intended for the real DySCAS system.

As the project still has 20 months to run and the actual middleware is yet to be built, the environment in which the services operate (comprising: external devices, ECUs, user interfaces, and wireless communication between the external devices and the middleware) has been simulated. A series of screen images animate the discussion of operation, which is based around a specific sequence of device discovery and attachment events.
prioritization’s UF to rank the relative importance of services under certain resource shortage conditions.

Figure 11. Another known, but not permanently authorized device (a specific laptop computer) is authorised by the user

Figure 12. The Resource Management Service prevents a fifth device from attaching

Figure 13. The current resource configuration only supports four services

Figure 14. The fifth device is now attached

In figure 11, a previously known laptop computer is accepted (after the user was prompted). Consider the resource status after the laptop computer was accepted. There are four devices integrated into the system (for the demonstration application it is assumed that this implies that exactly one application service from each device is in use, i.e. four services are in use). Figure 12 shows that a fifth device is rejected by the resource manager, before even reaching the security service, due to insufficient resources. Figure 13 shows the resource manager configuration. There is one powerful ECU available (supporting two services) and two low-performance ECUs (supporting one service each). So the decision to prevent attachment of a fifth device is correct. However note that the ‘Allow processing on external devices’ checkbox was not checked. This is now changed so that ‘external processing’ is enabled. Figure 14 shows that when the fifth device is represented to the system it is accepted. This is because the resource manager has detected that the laptop computer supports external processing (the ‘Processing’ service shown in figure 5), so it is aware that a fifth service can be accommodated.

The demonstration application, policies and default templates are available at [16].

7. Direction

This section discusses the current direction and some planned further enhancements, with respect to the policy-configured aspects of the middleware.

There is a need to achieve a balance between avoiding driver interruption (and removing the burden of manual configuration) through automation, yet facilitating personalisation. The demonstration application has shown (for the attachment of new devices scenarios) how the middleware will in most cases be able to determine what to do without user input. Once user authorisation and preferences have been set up for a particular device and its application services, the used need not be consulted again. If a device type is not recognised, or the current resource configuration cannot support the attachment of the device it can be rejected silently. By extending the policy logic to take into account additional context information, it would be possible to ensure that drivers are not consulted for input whilst the vehicle is moving, or whilst the engine is running, for instance.

Driver-specific personalisation (for example where several different drivers share a vehicle) should also operate silently once initially configured. The specific driver could be detected in some way (outside the scope of this paper) and that could cause the template directory to be switched so that the middleware obeys the current driver’s preferences. It would also be possible to download policies and templates from one vehicle onto an attached device (such as PDA) and transfer them to another DySCAS-enabled vehicle, so that the customised configuration follows the driver.

The UF in the demonstration application permits high flexibility in the way service prioritisation is decided (i.e. which system, environmental and user preference factors are considered). AGILE facilitates dynamic adjustment of the weights that are assigned to each of these factors (not done in the current demonstration). This will enable the service prioritisation itself to be contextually adjusted.

The middleware will require a standard method of naming and versioning policies, and a compact storage format so that several policies can be held in an on-line repository (the DySCAS draft architecture has such a repository but its capacity is yet to be confirmed).

The deployment of a policy-server (see for example [17]) is under consideration. This would be increasingly beneficial as the scale of deployed systems (in terms of the number of services they support) grows. Meta-data will be added into the policy script to ensure that the correct type of policy is loaded
into a specific component.

A suite of tools are needed to configure, validate, and diagnose policies. These tools must be intuitive and produce correct policies. The main authoring tool will support graphical based policy creation.

8. Conclusion

The work described in this paper brings together two projects, each pushing forward the state of practice in policy-based autonomies: DySCAS with its demanding requirements for self-managing capabilities; and the AGILE policy technology which has some advanced features that are of direct relevance to DySCAS.

The DySCAS GUCs are have been discussed in overview, illustrating their broad and varied nature. For each GUC, a SF was described in more detail, with specific focus on the role of policies in the determination of behaviour. The impact of the embedded deployment platforms of DySCAS (i.e. a heterogeneous mix of mainly very resource-constrained ECUs), on the feasibility of implementing self-management, was discussed.

A demonstration application has been developed, as part of an in-depth exploration of the self-management requirements involved with device discovery and attachment. The application serves as a concept evaluator. Several of the perceived DySCAS services have been emulated; these embed the real AGILE library, and thus load and execute real policies. The application shows some of the various ways in which policies will be used in the DySCAS middleware. The emulated services employ several advanced features of AGILE, illustrating its high suitability. These features include: 1. the ability to embed UFs directly into policies (which enables the optimisation capabilities of UFs to be combined within a rule-based logic); 2. the ability to combine policies and templates, and to automatically create new templates; and 3. the use of a meta-policy to select the most appropriate business policy, this promotes high flexibility and extensibility whilst keeping individual policies simple.

Finally, some discussion of direction and next-steps with respect to the policy technology deployment in the embedded middleware, was provided.

Further information about DySCAS

The partners are: Volvo Technology AB (the project coordinator), DaimlerChrysler AG, Enea Services AB, Robert Bosch GmbH, The University of Greenwich, The University of Paderborn, The Royal Institute of Technology (KTH), Systemite AB, and Movimento. The project started in June 2006 and runs until November 2008. Further details are available at the project website [18].

Acknowledgements

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

References

[18] DySCAS project website: www.DySCAS.org