
Mariusz Pelc∗ Richard J. Anthony∗∗ Paul A. Ward∗∗ James Hawthorne∗∗

∗ Opole University of Technology, 45-271 Opole, ul. Mikołajczyka 5, Poland, (e-mail: m.pelc@po.opole.pl).
** The University of Greenwich, Greenwich, SE10 9LS London, UK, (e-mail: {r.j.anthony,p.a.ward,j.hawthorne}@gre.ac.uk)

Abstract: Embedded systems are becoming truly ubiquitous in society, with new applications and usage domains appearing at a rapid rate. There is an increasing interest in, and need for, solutions which give these systems the ability to adjust their behaviour to the current environment conditions. Although context-awareness and self-configuration features are obvious responses to the needs mentioned, each potential implementation of them is inevitably a compromise between level of flexibility achieved and the accompanying resource costs. This balance is especially vital in the case of resource constrained embedded systems, which dominate in many application domains. In this situation it is extremely important to provide solutions for which the resource configuration can be accurately predicted for a given scenarios, and thus validation can be extended beyond behavioural correctness, to include real-time performance characteristics and resource usage such as memory, even though the precise operating circumstances may be unknown in advance.

In this paper we present a middleware architecture and accompanying software component model supporting self-configuration, context-awareness and dynamic decision making within embedded systems. The solution is scrutinised from the point of view of resource usage and efficiency and real-time performance, using typical hardware and operating platforms.

Keywords: reconfigurable architectures, dynamic decision making, context-aware systems.

1. INTRODUCTION

In the currently applied embedded systems there is an increasing need for context-aware behaviour, so that systems can operate in complex environments with various external influences. In order that these systems can maintain correctness and effectiveness it is necessary to enable their decision making to be informed by external state information. This kind of context-aware reasoning gives these systems desired features, such as flexibility, dynamic reconfiguration, self-* features, and ability to automatically handle a wide range of operating conditions. There are a variety of techniques and algorithms that can be used to achieve the necessary dynamic configuration, including artificial neural networks (ANN), fuzzy logic (FL), and genetic algorithms (GA). However, these popular algorithms have some substantial disadvantages with respect to their potential use in embedded pervasive systems. For example ANNs require large training data sets and / or a long training period; tuning of FL systems (such as Mamdani or Takagi-Sugeno) can need a lengthy tuning process; fuzzy-neuro systems suffer from the same problems as ANN; and GA systems require many iterative steps to achieve a final decision. The consequences of using these approaches include high resource usage, and often redundancy in this respect, as well as consuming much computational bandwidth and having associated decision latency. Furthermore, though all of them may have the advantage of prior configuration using expert-based knowledge, they actually do not support a much desired feature for pervasive systems which is customization.

Another issue being the subject of intensive research are seemingly conflicting architectural requirements to achieve simultaneously high flexibility and high efficiency, whilst supporting dynamic reconfiguration and context awareness of pervasive systems. For embedded systems the heavy-weight solutions are automatically excluded because of obvious reasons mentioned above. This leaves a narrow margin for use of existing solutions (in their entirety or at least some parts of them), which are known to be acceptable for big scale systems (such as self-configuration of server farms and grid schedulers where there is adequate processing and storage resource) and opens quite a big space for new solutions. However, in the case of development of new architectures for components and middleware, it is extremely important to guarantee their compatibility. Thus it actually makes sense to develop components and middleware architecture in parallel and provide interfaces enabling their integration. For
In case of pervasive systems there are various architectural solutions towards supporting context-awareness and self-* features in order to enable these systems to adjust their behaviour to the current situation on one hand and guarantee a level of autonomy needed to avoid permanent human supervision on other hand.

The LATIS pervasive systems architecture is presented in (Tadj and Ngantchaha (2006)). Within this architecture one of the crucial components is the multi-layered architecture of the Context Manager. Within this architecture the bottom layer consists of Context Sensors containing sensor hardware and software driver for collecting context data. These data are processed (this means conversion, pre-fusion, etc.) within the second layer by Sensor Information Manager and then sent to Context Information Manager, which is responsible for context storage (in the context database) as well as context learning (categorisation), context fusion (merging two or more measurements of the same context feature) and organisation. The fourth layer is the Context Manager handling context queries and working as proxy for context sharing between Context Agents, the top layer of the architecture. Context Agents serve as a context broker for any context information request from any entity available on the device. Context services available in the Context Managers are published in the Context Agents service interface, when they are registered to the Context Manager. The LATIS architecture from the point of view of context management is much more complex than our solution, thus may not be optimal for embedded systems. The main difference between these two solutions lies in the number of layers needed to handle the context information flow. Using LATIS architecture as the reference architecture we would say that our architecture has only two layers: simplified context management layer (context producers and subscribers need to register here in order to establish dependency links between them) and the layer, where all context providers and subscribers are located.

A Context Management Services architecture which additionally supports a context bridging functionality is described in (Hesselman et al. (2008)). This architecture distinguishes three different categories regarding context-related functionalities: context producers, which publish context, context consumers, which retrieve context from context producers either via subscription or query, and context brokers. Context producers register with brokers, which are responsible for managing of life-cycle of the context producers (they can start or stop them). The architecture assumes also that context producers may also be context consumers. The context broker is responsible for identity management and context discovery, whereas the context producers act as proxies and take care of context adaptation, reasoning, and mapping context information to another format. Both the broker and the context producers are involved in the enforcement of privacy policies. Functionally this architecture is similar to ours. However, in our solution the Context Manager role is restricted only to mapping context information between providers and consumers and the context adaptation or mapping context information to another format is being performed at the context processing stage by the policy evaluation library.

Another Context Service architecture is presented in (da Rocha and Endler (2005)). This service was designed in order to deal with the following requirements: 1) adoption of a generic and flexible context model that could be dynamically deployed in the middleware architecture; 2) improved performance, allowing fast access to and dissemination of context information, avoiding the intensive use of network and memory in limited-resource devices; and, 3) support for interoperability, facilitating the use of the service on different devices, operating systems and programming languages. The Context Service is composed of Context API (provides unified access to any middleware service, to the application), Event Service (responsible for providing asynchronous communication, delivering contexts and events to interested clients) and Type System Manager (maintains the dynamical context type system,
solving and recognizing the available context types at runtime) and Repository, storing context information. This architecture is much more complex than our solution. The common part of the two concepts seem to be existence of context repository, which is maintained in our solution by the Context Manager service (actually, it is integrated within the Context Manager). We have not implemented anything like Type System Manager within our architecture as the issues related to context types are being solved by the policy evaluation library.

A Reconfigurable Context-Sensitive Middleware (RCMS) is presented in (Yau et al. (2002)). RCMS provides an object-based framework for supporting context-sensitive applications. RCMS models context-sensitive application software as context-sensitive objects, which consist of two parts: a context-sensitive interface and a context independent implementation. The interface encapsulates the description of the application’s context awareness, whereas the implementation remains context free. For context-sensitive application software, RCMS provides adaptive object containers (ADCs) for runtime context data acquisition, monitoring, and detection. During runtime, the ADC communicates with the underlying system to acquire context data and then performs periodic context analysis as specified in the context-sensitive interface. To support this architecture a context-sensitive object request broker (R-ORB) is used as the key mechanism for providing communication transparency for context-sensitive application software. R-ORB hides the intricacies of ad hoc networking. It also performs device and service discovery on behalf of the context-sensitive objects. The described architecture is claimed to be similar to well known CORBA, COM and TAO standards, which makes it a very reliable solution, but as well a bit heavy weight for typical embedded systems application. Furthermore this solution provides also a designated communication protocol for ad-hoc communication, whereas our solution is based on much simpler but very efficient message passing between middleware components.

A middleware architecture applicable to soft-ware pervasive systems is described in (Edwards et al. (2007)). The architecture addresses such challenges as dynamic software update (with concern to not to violate real-time constraints), service discovery, transparent replication of service providers (in case of failure any of them) and logical mobility (considered as the capacity for software components to migrate from host to host after initial deployment). For each of these capabilities, it is first described its relationship to challenges like real-time concerns, scalability, decentralization, safety-critical aspects, and then it is explained how each capability is supported by leveraging Prism-MW (Malek and Mikić-Rakic (2005)) middleware platform. The architecture supports also inter-component communication via a designated pair of interfaces/components called SEngine and SConnector. There is a substantial difference between this architecture and ours related to the way the components register within the middleware. In case of our solution each component has to register itself by sending a special message type, whereas the Prism-MW architecture uses dynamic service discovery.

In (Gui et al. (2008)) a component framework is proposed to allow configuration of both components and inter-component interfaces whilst providing real-time performance in an embedded system. A further similarity to the work described in this paper is the use of a markup language (XML) to configure components and their use in the system. The framework is implemented in Java (in the contrary our solution is developed in C/C++), so the potential embedded platform has to run Java Virtual Machine. This results in some higher requirements regarding the deployment platform on the one side and will negatively impact the system performance.

With embedded systems it is possible to build reconfigurable software starting with a component framework at the operating system level (Polakovic et al. (2007)). Here an architecture of custom reconfigurable operating system is presented together with higher level reconfiguration language FScript.

Applications capable of dynamic configuration are added, with the configuration changes being specified using a purpose designed script language. Each script describes the steps required for a reconfiguration action and is subject to compile time checks to negate trivial programmer errors. The use of a script language has a similar advantage to the policy approach advocated in this paper; both allow the system to be altered by simply passing in data. Their effects differ in that these scripts alter component interactions whereas AGILE policies alter the actual behaviour of components.

3. SUPPORT FOR CONTEXT-AWARENESS AND SELF-CONFIGURATION IN EMBEDDED SYSTEMS

For reasons of flexibility and dynamic upgrade, it is necessary to deploy the high-level strategic control logic independently of the lower-level, static implementation mechanisms. In this section we discuss both, the component and middleware architecture that support context-awareness and self-configuration.

3.1 Software Component Architecture

In our scheme the static and dynamic aspects are separated out at design time. Places at which dynamic decisions will be made (called Decision Points - DPs) are embedded into the static code, as illustrated in Fig.1. The arrows in the Fig.1 represent respectively control flow in case of the component evaluation was successful (the solid line) and alternative control flow depending on error occurrence or dynamic decision making (the dashed line).

At this point it is not necessary to know what decision logic will be placed there in the future, but simply to know that this is a place where upgradable logic will be beneficial. The DPs are populated at run time with actual decision logic. The architecture has been designed to be flexible so that in fact almost any dynamic decision making technology could be supported within a DP. However, for the purposes of flexibility, low run-time resource requirements and decision latency, as well as support for customisation, policy-based computing has been used in our work to date.

The DP approach enables the dynamic decision making aspect to be modular and is a significant strength with
Fig. 1. Code-level and schematic views of a software component containing a Decision Point respect to versatility and future-proofing. The current implementation uses the AGILE policy technology, documentation is available at (Anthony –).

The addition of dynamic configuration brings with it potential new failure modes (for example, perhaps there is no policy available for a given DP). Such failure modes will not be detectable at design time because of the very nature of the scheme (in that, the decision logic can be provided dynamically). In order that the system can automatically locally detect and handle the new failure modes we introduce a Dynamic Wrapper (DW) which encapsulates the DP, as shown in Fig.1.

The DW integrates the various parts of the architecture, providing transparency at design time (the software developer does not deal with the internal composition of elements within the wrapper) and at run-time (the host component is not concerned with internal decision process of the DP, it is given a decision output, via the wrapper).

3.2 The Middleware Key Services

The component can communicate with other components using API interface provided by middleware services using message passing interface provided by Communication Manager. The wrapper shields the component from any faults that occur with the DP at run-time; different actions are taken depending on the specific type of fault (such as the new policy cannot be located in the repository, the new policy causes a parse error, or requires additional context information that is not available, etc.). At design time, one can define the default behaviour, i.e. the result returned from the DW after an error is detected. This default return mechanism ensures that under no circumstances does it fail to return a legal decision, thus ensuring that the self-managing application is just as robust as the equivalent static system, despite having dynamically selected policies and dynamically identified context requirements.

Each DP has its own DW and is configurable to have a unique default behaviour. A single component can deploy as many DPs as necessary to distribute the required dynamic configuration logic (Ward et al. (2008)). Each software component interacts with two key services: the Context Manager Service (CS) and the Repository Service (RM).

3.3 The Communication Manager (CM)

The Communication Manager implements message passing service with functional interface supporting low-latency communication handling between registered software components. It is entirely transparent for components and other middleware services (Anthony et al. (2008)). This transparency gives the impression that all elements within the communication system (software components and middleware services) interact directly each other.

The Communication Manager uses two means of communication:
• System socket - CM listens on a selected system port for all components that want to register with CM; they can register as context provider, context subscriber or as fixed component (component with fixed logic).
• System message queues - once a component gets registered with CM, the CM creates a unique message queue and runs a separate thread for handling communication between the newly registered component and those previously registered.

A separate message queue is used for each component to simplify and accelerate communication such that a message gets to the target component with low latency. Using a single shared message queue would incur overheads including assigning unique message type identifiers, which in turn significantly reduce flexibility because the mapping between message type and component actually can only be static (fixed) due to message queues implementation in Linux. Otherwise a component wouldn’t know which message is addressed to him. In the implemented message queues version, the CM creates the link between a component and its queue dynamically in run-time.

To make the communication system work properly, the Communication Manager maintains a special table containing the mapping between components and corresponding queues IDs. In case of any communication system element wants to refer to any other element, this table supports the implementation of Name Service functionality - the target component is first allocated within the table, then the queue associated with the component is used to pass the message to this component.

3.4 The Context Manager (CS)

To support incremental upgrade of software components, as well as run-time behaviour reconfiguration by changing policies it is necessary to have a context management approach that supports a dynamic mapping between pairs of components and a dynamic association between each component and the context information it uses (Anthony et al. (2008)).

The CS has been designed such that it decouples context providers from context consumers; this means that any component can produce context information and any other component can consume that context when making dynamic self-management decisions. There is no need to know the context provision or consumption characteristics of components at design time - this is a key requirement for dynamic configuration flexibility since these aspects can change with individual component upgrades (especially context provision) and with changes of dynamic policy logic loaded into DPs (especially context consumption).

The CS dynamically manages context information sent by providers and keeps a table of consumers which have subscribed to specific context items. The CS facilitates dynamic associations between components and context items (as one component may subscribe to more than one context item, and the context requirements may change when a policy is changed). All associations are stored in special Translation Tables implemented within the CS. The tables are used to dynamically link context subscribers (in terms of context items needed by the subscribing components) and providers.

The CS operates on a subscription basis in which providers and consumers register an interest in the context information they can supply, or will consume, respectively. Using the CS as a context intermediary in this way significantly cuts down interaction and communication complexity. Context information is pushed out on a state-change basis and cached at consumers so it is immediately available for use; facilitating very low latency context-aware decisions which are required in real time environments.

As within each component there may be more than one Decision Point, CS deals internally with Decision Points multiplexing in order to deliver required context information to requesting Decision Point.

3.5 The Repository Manager

The repository provides persistent storage for policies. The RM manages the repository, provides the correct policies to DPs within components, and contributes to the validation support and robustness aspects of the architecture (Anthony et al. (2008)). Each policy includes information (metadata) that will enable it to be placed into the correct DPs, examples of metadata include the policies function and identification of components / DPs for which the policy is suitable. Policies are represented as short data files that can be developed and validated offline and loaded into the repository at any time and by a variety of means - for example in the automotive domain policies could be copied from an attached USB memory stick enabling driver / owner customization of a vehicle. For some applications such as factory or home automation, new policies could be delivered via a network connection, possibly even directly from the device or system manufacturer’s back-end system. Additionally, the RM also enables policy versioning and rollback operations should a DP report an error during policy load.

There can be several DPs within a single component, and multiple DPs can possibly use the same policy. To deal with this mapping and multiplexing the RM maintains Translations Tables. The policies table holds information concerning: the service type for which each policy is applicable (useful for validation purposes), policy version and a timestamp. The components table holds currently known details of the components and an array of DPs within each specific software component. Each DP entry in this table holds a link to the currently loaded policy and previous policy if known.

The RM can provide the DPs with a policy at various times, such as component initialization, or during run-time to dynamically change behavior. If a policy load fails (for example because of failure of internal validation), the RM provides the component with the policy that was previously successfully loaded. It is possible to extend the repository service to include a larger history of policies for each DP in future implementations.

4. IMPLEMENTATION

There is a very wide variety of embedded systems hardware available, and a very wide range of application domains
in which it is deployed. Our work began as part of the EU-funded FP6 DySCAS project (DySCAS (−)) which focused on automotive systems, and thus we selected our implementation platform to be representative of some of the typical Electronic Control Units (ECUs) found in modern vehicles.

4.1 Selected Aspects of Hardware Platform and OS

Our work is concerned with some of the more challenging aspects of embedded systems; in addition to achieving dynamic configuration we are also very concerned about real-time performance, resource efficiency and validation. Therefore the choice of platform on which to base demonstration and validation activities is a very important consideration.

The middleware has been implemented using two hardware platforms TS-7260/7400 with the Linux operating system preinstalled (for details the reader is directed to (TechnologicSystems (−)). The main features of these boards include:

- 4MB RAM,
- 512MB space on SD Card configured as file system,
- Preinstalled Linux Debian system,
- On-board temperature sensor,
- Real Time Clock,
- 2 USB ports making possible interfacing external devices (such as WiFi cards, Bluetooth dongles, GPRS modules, etc.),
- 10/100 Mbps Ethernet interface.
- Additional extension ports for custom hardware expansion and direct I/O peripheral interfacing.

For experimental evaluation purposes the TS-7260 card is configured to be a device running the control algorithm (i.e. it hosts the dynamically configured software component(s)); the TS-7400 is configured to serve as the remote policy repository.

In order to run the middleware, the deployment platform has to run an operating system which meets some key requirements. These include:

- Networking support (socket mechanism),
- Message queues support,
- Multithreading support.

The above requirements are fully satisfied by the embedded Linux operating system, so this is the system platform the described implementation uses (but it is not limited to the Linux platform).

4.2 Component Development Kit

The middleware was developed using C++ language, as was the AGILE_Lite++.++ library, a module responsible for policy evaluation and thus dynamic decision making. In order to hide implementation details from a component developer, and to simplify development of context-aware components, the interface for using middleware services was embedded into the accompanying Component Development Kit (CDK), also developed in C++. This makes the creation of components and decision points as well as a)

```cpp
ControlComponent *comp =
=OpenDecisionPointComponent(char *newCompID,
int dcmCheckPeriod);
```

b)

```cpp
DecisionPoint *dp =
=AgileDecisionPoint(char *newDPID,
ObjectList*newOVarList, char *defaultReturn, ...,
NULL);
```

Fig. 3. Creation of software components and decision points using the CDK
decision evaluation extremely convenient. In Fig.3 (a, b) one can see prototypes used for creation, respectively, a component and a decision point within the component. Creation of an open decision point component requires specification of its unique name and policy check period; each component will then check every `dcmCheckPeriod` evaluation cycles whether a new policy is available in the policy repository for any of its DPs. An evaluation cycle is a single processing of current context information in order to evaluate a decision. At the DP instantiation it is required to specify, a list of Output Variables (which in addition the the actual evaluation decision of the policy, facilitate returning additional contextual information to inform the host component’s behaviour), the default return value (this is the value returned whenever the Dynamic Wrapper detects any error during the dynamic decision making process), and a NULL terminated list of valid outputs (based on this list the Dynamic Wrapper is able to distinguish valid policy decisions).

5. PERFORMANCE AND RESOURCE USAGE TESTS

For all embedded systems it is extremely important to guarantee the most efficient usage of available resources (operating memory, processing power, etc). In this section we provide some evaluation details regarding these aspects. As of today there is actually no existing methodology to be applied to measure the complexity of policies such as those supported written in the AGILE policy language. In this situation we decided to use a `policy complexity index` and relate it to the number of functional blocks (Rules, ToleranceRangeChecks - TRCs and UtilityFunctions - UF's) a given policy contains. Though this is somewhat arbitrary measure (and should be treated only as estimation rather than an exact measure), at this stage of our research we are trying to provide some rough estimations regarding impact of the policy complexity on the critical aspects from the point of view of real-time embedded systems.

This approach implies naturally that the more functional blocks the policy has the more complex it becomes and the higher the policy complexity index is. In general, AGILE policies can contain various functional blocks, such as `Rules`, `ToleranceRangeChecks`, `UtilityFunctions`, etc. As the blocks are of different structure and thus complexity, this may impact on resource usage and overall decision system performance. For example, the Rules themselves can be of simpler or more complex form, UtilityFunctions can have different number of `Terms`. Policies can also differ in the amount of context information they use. AGILE policies also support indirect addressing, so it is also possible that the evaluation of one functional object...
changes the execution path, thus changing the run-time complexity of the logic evaluation. This all affects resource usage and the time taken to process policy logic and come to a decision.

5.1 Estimation of Policy Evaluation Time

In order to provide an estimated policy evaluation time depending on policy complexity, we did as follows:

(1) A suite of policies containing 1, 5, 10, 15, 20, 25 and 30 Rules, TRCs and UFs respectively, was created.
(2) An experiment was carried out to measure the policy evaluation time for the different complexity policies.

In this experiment we assumed a sequential evaluation of all Rules (and other functional blocks) in a policy. This is sufficient to determine typical performance of a policy of an appropriate size but caution is needed when interpreting the results in the context of any specific policy. Fig. 4 shows the results of the Decision Point evaluation time (denoted as DPET which is the sum of the Rules, TRCs and UFSs evaluation time and time needed by the Dynamic Wrapper to verify correctness of the returned result). The results indicate a near linear relationship between policy complexity and the DP evaluation time. This is important because it gives developers a means of predicting the worst-case execution time of the DP (i.e. when all rules fire in sequence and none are skipped). One can see that Decision Point evaluation time in case of a policy containing 30 TRCs is the longest and takes about 0.25s. This highlights the issue of resource (performance) limits of the board we have used.

5.2 Estimation of Memory Usage

The other tests we did were related to the impact of a policy complexity on the memory usage. Similar to the tests presented in the previous subsection, we did the following:

(1) A previously created suite of policies containing 1, 5, 10, 15, 20, 25 and 30 Rules, TRCs and UFs respectively, was created.
(2) An experiment was carried out to measure the resource usage (specifically memory) for the different complexity policies and DP evaluation time.

And similar to the previous tests, we assumed here a sequential evaluation of all Rules (and other functional blocks) in a policy Fig. 5 shows the near linear relationship between policy complexity and the memory usage after the policy is loaded (denoted as MU). These results are more absolute than the evaluation-time results (which are effectively a worst-case prediction), because when a policy is loaded, all functional blocks represented in the policy script are instantiated as populated objects and thus impose a memory cost regardless of the execution path taken through the logic. Also, the memory usage should be the same for all platforms, whereas DP evaluation time is dependent on the underlying platform. The memory usage by the Software Component itself (as instantiated object) with one Decision Point is about 25kB, but the Linux process running the component needs an additional 500kB of RAM. This is extremely important to point out, that the 500kB of RAM needed to run a software component is not determined by the component or middleware architecture, but is the result of the operating system overhead. In case of less sophisticated embedded systems (which aren’t target for our solution) the efficient memory usage would be about 25kB per component.

These results indicate that the method and mechanism used to achieve context awareness are suitable for performance critical and resource critical embedded systems. They allow up-front prediction of the impact of the context-aware reasoning, even when the actual policy logic and specific context items used are not known a priori.

6. CONCLUSIONS

In this paper we have presented architecture for both: context-aware software components and context-aware middleware for pervasive embedded systems. These architectures were tested from the point of view of performance and memory usage, depending on policy complexity. These aspects are crucial for potential target platforms, such as...
as vehicular systems or other systems, where there is requirement for adaptation to environmental conditions and desire for a level of autonomous behaviour.

For evaluation purposes we have used a simple policy complexity index as the number of rules a policy contains. The results we have obtained indicate that the architectures are appropriate for pervasive embedded systems, in which real-time performance constraints and limited processing and memory resource are common issues. We have determined that both system performance and memory usage are linearly-dependent on the policy complexity. This feature is very advantageous as it enables the possibility of prior estimation of resource and processing power requirements in the case of assumed complexity of decision logic implemented in a policy.

Our performance analysis has also revealed, that the presented architectures of component and middleware can be implemented in many, though not all, currently used embedded systems configurations. Actually, all embedded platforms working under Linux operating system and equipped with at least 2MB RAM are perfectly capable to run our middleware (it requires 1.14MB to run three key middleware services) together with a number of software components. In case of less advanced platforms the requirements regarding memory amount or processing power would be much smaller.

7. FUTURE WORK
There is no doubt that the near future will be dominated by rapid development of pervasive and embedded systems and expansion of their applications into many new domains. The most obvious directions of future development include architectural and functional challenges regarding context-awareness, human-machine interactions and support for increasing autonomy of pervasive systems.

Taking into account the prospect of development of pervasive embedded systems one can see that the increased flexibility and autonomicity will result in increased resource usage. From the technological point of view the most critical constraints will refer (and may be reduced actually, as the resource constraints are a function of technological development) to real-time aspects and performance-related issues. Thus for the future ubiquitous embedded applications / systems it is very desirable to develop architectural solutions which will be the best possible trade-off between real-time constraints and resource constraints.

Our research has also risen a very important issue regarding application of a valid methodology for measurement of policies complexity. The measure applied in this paper is simplified and can only be used as rough estimation of resources needed for decision making systems with AGILE technology. However, the work towards providing much more exact policy complexity measure has begun.

ACKNOWLEDGEMENTS
The DySCAS project (DySCAS (–)) was funded in the 6th framework program “Information Society Technologies” of the European Commission. Project number: FP6-IST-2006-034904. The partners were: Volvo Technology AB (the project coordinator), Daimler 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 ran until February 2009.

REFERENCES

ACKNOWLEDGEMENTS
The DySCAS project (DySCAS (–)) was funded in the 6th framework program “Information Society Technologies” of the European Commission. Project number: FP6-IST-2006-034904. The partners were: Volvo Technology AB (the project coordinator), Daimler 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 ran until February 2009.

REFERENCES

ACKNOWLEDGEMENTS
The DySCAS project (DySCAS (–)) was funded in the 6th framework program “Information Society Technologies” of the European Commission. Project number: FP6-IST-2006-034904. The partners were: Volvo Technology AB (the project coordinator), Daimler 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 ran until February 2009.