Analysis and Design of Dynamic Behaviour for Embedded Systems Using Policy-Based Design

ANTON HOU

Examensarbete
Stockholm, Sverige June 2014

XR-EE-RT 2014:015
Acknowledgement

Firstly, I would like to give my deepest gratitudes to Detlef Scholle and Dimos Dimarogonas who made this master thesis work possible.

I would furthermore like to give my thanks to Cheuk Wing Leung and Detlef Scholle for providing me with feedback, tips, and knowledge during my thesis work at Alten. Their dedication to my progress has been both inspiring, helpful, and invaluable.

A special thanks also goes to the other thesis workers Andreas Hammar and Fredrik Frantzén for their support.
Sammanfattning

Policy based computing is a technology which aims to abstract away decision making from the source code of a program. The technology provides flexibility/changeability and adaptivity in terms of software behaviour. This is allows behaviour to be adapted in runtime.

This thesis contributes with an investigation and analysis of the technology with a focus on realtime multicore embedded systems. The study identified several challenges in terms of policy based systems, and resulted in a suggested design.

This thesis also includes a prototype implementation which showcases the usage of the suggested design. The implementation involves a policy engine which allows a low overhead, soft realtime performance, and context aware decision making for a targeted software component. The implementation is delimited to Linux based systems.

It was concluded that realtime performance can be applicable to policy based systems, and that the suggested model gives high portability, and low over head. This makes the policy based approach highly applicable to embedded systems.
# Innehåll

1 Introduction .................................................................................. 5  
  1.1 Background ........................................................................... 5  
  1.2 Team Goals ........................................................................... 6  
    1.2.1 Use case ........................................................................ 6  
  1.3 Problem Statement .................................................................. 6  
    1.3.1 Points of interest ............................................................. 6  
  1.4 Method .................................................................................. 6  
    1.4.1 Theoretical Study ............................................................ 6  
    1.4.2 Practical Part ................................................................. 6  
  1.5 Delimitations .......................................................................... 7  
  1.6 Report outline ...................................................................... 7  

2 Policy-based Systems .................................................................. 8  
  2.1 Introduction .......................................................................... 8  
    2.1.1 Explaining self-management and MAPE-K ...................... 8  
    2.1.2 A more concrete model for realizing the MAPE-K .......... 9  
  2.2 Principle of policy-based design ............................................. 9  
  2.3 Policy expression ................................................................... 10  
  2.4 Policy behaviour ................................................................... 11  
    2.4.1 Three restriction levels of policy behaviour .................. 11  
    2.4.2 Scopes of adaptation ..................................................... 12  
  2.5 Policy Types .......................................................................... 12  
    2.5.1 Action policy ................................................................. 12  
    2.5.2 Goal policy ................................................................. 13  
    2.5.3 Utility policy ............................................................... 13  
  2.6 The RRR Model ................................................................... 13  
  2.7 A Policy standard ................................................................. 15  
  2.8 Realtime support in policy-based systems ............................. 16  
  2.9 Discussion ........................................................................... 17  

3 Challenges in policy based design ............................................. 18  
  3.1 Verification and validation ..................................................... 18  
  3.2 Resource sensing ................................................................... 19  
    3.2.1 Close threshold fluctuations ......................................... 19  
    3.2.2 Temporary and random spikes ...................................... 19  
  3.3 Conflicts ............................................................................... 20  
  3.4 Policies in terms of timing ..................................................... 21  
    3.4.1 Determinism ............................................................... 21  
    3.4.2 Realtime scheduling .................................................... 22  
  3.5 Conclusions ........................................................................... 24  
    3.5.1 Validation and Verification ........................................... 24  
    3.5.2 Resource sensing ........................................................ 24  
    3.5.3 Conflicts and realtime support ....................................... 24  
  3.6 Discussion ........................................................................... 24  

4 Related work in policy-system design .................................... 25
4.1 Introduction ........................... 25
4.2 A framework for adaptivity in pervasive computing ........................... 25
4.3 Dyscas/Shape ........................................ 26
  4.3.1 Introduction ........................................ 26
  4.3.2 Adaptability concept ........................................ 26
4.4 MUSIC ........................................ 27
  4.4.1 Introduction ........................................ 27
  4.4.2 Adaptability concept ........................................ 27
4.5 PohSAM ........................................ 28
4.6 Discussion ........................................ 28
  4.6.1 Handling of the challenges ........................................ 28
  4.6.2 Ways of triggering adaptation ........................................ 29

5 An Interesting Framework 30
5.1 Introduction ........................................ 30
5.2 Agile ........................................ 30
  5.2.1 Policy mechanics ........................................ 30
  5.2.2 Policy Structure ........................................ 31
  5.2.3 Library functionalities ........................................ 32
  5.2.4 Decision Points ........................................ 32
  5.2.5 Discussing Agile’s adaptation support ........................................ 32
5.3 Agile Lite ........................................ 34
  5.3.1 Realtime support in Agile Lite ........................................ 34
  5.3.2 Conclusion ........................................ 35

6 Requirements for policy engine 36
6.1 Overview ........................................ 36

7 Specification for Design 37
7.1 Introduction ........................................ 37
7.2 A suggestion for design ........................................ 37
  7.2.1 Discussion ........................................ 39
7.3 Use case for implementation ........................................ 39
  7.3.1 Policy-based decision making for MemSched ........................................ 39
  7.3.2 Policy-based decision making for FaceRec (Face detection) ........................................ 39
  7.3.3 Choosing the approach ........................................ 40

8 Implementation of the policy system 41
8.1 Overview ........................................ 41
8.2 Policy Engine Implementation ........................................ 42
8.3 API for Policy Engine ........................................ 43
  8.3.1 API headers ........................................ 44
8.4 Use case implementation ........................................ 44
8.5 Results from the implementation ........................................ 44
  8.5.1 Execution times ........................................ 45
  8.5.2 Validation of the policy system ........................................ 45

9 Summary, Conclusions and Future Work 46
9.1 Summary ........................................ 46
9.2 Conclusions ........................................ 46
9.3 Future Work ........................................ 47
## Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>DySCAS</td>
<td>Dynamically Self-Configuring Automotive Systems</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>SW</td>
<td>Software</td>
</tr>
<tr>
<td>RT</td>
<td>Real time</td>
</tr>
<tr>
<td>ECA</td>
<td>Event-Condition-Action</td>
</tr>
<tr>
<td>RRR</td>
<td>Reaction, Routine, Reflection</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organization for Standardization</td>
</tr>
</tbody>
</table>
Kapitel 1

Introduction

1.1 Background

Embedded systems are used everywhere today, and the demand for more sophisticated embedded systems is growing. The market desires an increase in the efficiency of products in terms of e.g. power consumption, realtime performance, and computing power. To cope with this, improvements on both hardware and software levels has to be considered. This indicates that overall efficiency is also affected by how software applications are integrated with the underlying hardware. A middleware is commonly used as a bridge between hardware and software that abstracts away the hardware structure when developing software applications. It usually consists of multiple components such as a scheduler, and other adaptive components. Having a middleware simplifies the process of developing efficient applications. Furthermore, it implies that much of the overall efficiency improvements can be gained through enhancing the middleware. Design decisions in a middleware can therefore be heavily influenced by the desired use case. However, the use case might change in the lifecycle of an embedded system, alternatively, the system might have several use cases. This makes it desirable to integrate a flexible and dynamic behaviour to a middleware so that it can adapt to use cases and current system contexts in runtime.

One solution for this kind of behaviour in realtime embedded systems is through the use of policy based concepts and design. This allows a dynamic behaviour in runtime through instantiating particular implicit policies based on input parameters (e.g. available memory or CPU time). This master thesis is an academic study on the topic of dynamic behaviour using policy based concepts. The task is to evaluate the state of art in policy based concepts and to verify this technology by suggesting a design that is implemented to control an use case application.

The work will be performed at Alten AB in Kista, Sweden, in collaboration with the Royal Institute of Technology, KTH. Alten is a consulting and engineering company with main customers within energy, telecommunication, industry and automotive. The thesis is part of the ARTEMIS/PaPP project, the KKS/DAGGERS project, and to some extent also the ARTEMIS/CRAFTERS project. The overall goal of the PaPP project is to improve/develop tools and methods for developing many core embedded systems, while the goal for CRAFTERS is to develop a framework for realtime applications for embedded many core systems.

1.2 Team Goals

This master thesis is performed alongside two other theses. The overall goal for the team is to analyse different aspects of Linux based multicore platforms, and to showcase the technologies by implementing it into a prototype middleware. A verification of technology is used as well as a validation of our use case. The team planning has been executed through Scrum methods, both for coordination and transparency in our work.

1 www.alten.se
2 www.kth.se
1.2.1 Use case

The teams use case for the specific technologies is a face recognition system which has two distinct
components: face detection, and face recognition. The principle is to use face detection and recognition
to improve facility security by allowing realtime information presentation when surveying individuals
through security cameras. To do this, information from a database is extracted based upon the individuals
that the system recognises. The information is then displayed on the video feed which may be monitored
by security officers.

1.3 Problem Statement

This master thesis focuses on evaluating the technology of using policy based methods, and investigating
how it can be applied to a multi core embedded system that runs the team’s use case. A policy based
system is implemented as a middleware for the thesis use case. The implemented policy system is also
evaluated with regards to concluded requirements from the study.

1.3.1 Points of interest

Based on the problem statement, the following subjects are addressed in this report:

- The current state of art in policy based design.
- How policies can be implemented with realtime support.
- How a policy system could be implemented for high performance embedded systems based on iden-
tified challenges and findings.
- How the overhead caused by the addition of a policy based feature can be minimized.

1.4 Method

The ambition of this work is to follow the quality assurance guidelines for thesis works set by UKÄ.

The work is divided into two major parts; a theoretical study and a practical part.

1.4.1 Theoretical Study

The theoretical study is a literature study which involves reading available academic articles, books,
manuals and other theses that are relevant for investigating policy handling. The findings are introduced,
analysed and presented in this report.

Concepts and methods are compared and evaluated with regards to the requirements and the given
use case that has shaped the resulting specification for design. The specification includes the technical
requirements and a verification basis.

1.4.2 Practical Part

The practical part includes designing a middleware based on the specification for design and implement-
ation. The implementation was carried out using test driven methodology together with the other two
theses workers at Alten. Testing and verification was conducted to ensure that the requirements are met.
The work included static code analysis with CPPcheck, dynamic code analysis with Valgrind, unit
tests with Googletest and block tests. The implementations and conclusions are presented in this
thesis report.

http://www.uk-ambetet.se/utbildningskvalitet/kvalitetenpahogskoleutbildningar.4.782a298813a88dd0dad800010221.html
1.5 Delimitations

Based on all above, this project is delimited as follows:

- The time limit of the work has been 20 weeks, each week consisting of approximately 40 work hours.
- Theory is limited to concepts applicable to realtime middleware designs and usecase.
- The implementation is made in C code and the targeted software component for the policy system is the facerecognition system.
- The middleware is implemented on a system running Linux.

1.6 Report outline

This report begins with a short background introduction to adaptive systems, followed by the current state of the policy concept. Some implementations of policy based systems is then introduced and discussed based on the findings. This yields a foundation for a proposed design which is a convergence of all findings. An implementation is then verified and analysed.
Kapitel 2

Policy-based Systems

2.1 Introduction

Flexibility and adaptability are characteristics that has been adopted in multitudes of research areas in today's technological paradigm. It is also the core ideology behind self-management. In this context, flexibility relates to system adjustments in the presence of change. Policy based systems are closely related to self-management and adaptation. Many aspects of policy based design has been inspired by these concepts. This chapter will therefore begin with an introduction to the concept for self-management and adaptation before discussing the aspects of policy-based design.

2.1.1 Explaining self-management and MAPE-K

Self-management is a term that was first defined and adopted by IBM in 2001 by IBM's senior vice president of research, Paul Horn, when addressing the issue of the constantly escalating complexity of their software [7]. The concept of self-management was inspired by the human body. The human body is able to maintain, operate, and adapt itself without the need of a controlling mind, which is precisely what IMB's research thrived for, but for software systems. He defined self-management as a collective word for different aspects of flexibility in software architectures. The word self indicates that the system has an autonomic functionality to dynamically adjust itself in run-time. This concept was then further defined in 2003 by J. Kephart [7] together with IBM. The property of self-management can be collectively referred to as self-* [8].

IBM has defined self-management as a collective word for:

- self-configuration: The ability to autonomously install, configure and integrate new complex systems.
- self-optimization: The ability to autonomously tune itself for better performance.
- self-healing: The ability to autonomously find, log, and patch faults and bugs in the system.
- self-protection: The ability to autonomously anticipate and mitigate outside threats based on sensors and previous problems not covered by self-healing.

To realize these characteristics for a software system, IBM produced a system strategy called MAPE-K which is depicted in Figure 2.1 [7]. MAPE-K stands for Monitor, Analyze, Plan, Execute - Knowledge. It is a principal strategy for processing information and propagating it to an adaptive behaviour. Their idea is that MAPE-K should be the core of any autonomic element that thrives for self-management.

The monitoring element is used for reading information that is relevant to the autonomic element. This information could be either environmental information such as CPU utilisation, or system specific information such as system states or function calls. The first being passive monitoring, and the latter being active monitoring [9]. Passive monitoring can be done through the system interface, such as Linux's /proc pseudo-file system. Active monitoring is usually done by inserting coded probes in the system structure.
Planning is about processing the monitored data to produce a series of changes in the managed element. Different methods for applying these are discussed throughout this report.

### 2.1.2 A more concrete model for realizing the MAPE-K

Kramer and Magee [8] discusses an architectural approach based on the MAPE-K model. They define that a flexible system based on the MAPE-K model should contain three layers: **Component control**, **Change management**, and **Goal management**. The three tiers are depicted in Figure 2.2, where each tier treats the follow question:

- **Goal management** - How will on-line system planning be performed?
- **Change management** - How will the adaptation plan be executed?
- **Component control** - How will each target component be affected?

The component control layer can be seen as the *Monitoring* and *Acting* element, while the change management stands for the *Analyzing* component. Lastly, the goal management layer does the *Planning* procedure.

### 2.2 Principle of policy-based design

Using policies is an approach for achieving a flexible and adaptive system behaviour. The term policy is referring to the use of user-defined rules to shape the system’s adaptive behaviour [10]. As an example, a policy can consists of a set of conditions, actions and rules; "If (THIS) THEN DO (THAT)", where "THIS" is a condition, "THAT" is an action, and the surrounding logic is the rule. A condition could be a temperature threshold, and a consequential action could then be to turn on a cooling system as a counter measure.

The rules are per practice separated from the main source code and are instead accessible to the user via some kind of human readable interface. This storage, together with its interface is collectively seen as the *policy repository*. To propagate the rules into actions of adaptation, there has to be some kind of a policy engine that bridges the policies into low-level system behaviour.

The principle of policy handling on an abstract level can be described as an event flow (also depicted in Figure 2.3):

1. User defines adaptation rules for the system. E.g. Lower power consumption if battery is low”. These rules are stored in a policy repository.
2. When the system needs to adapt, it calls the policy engine.
3. The policy engine plans the next action based on the rules and current system information.
4. The decided action is propagated as a decision or change to the targeted software via its effectors.
To further detail the concept of using policies, the following terms will be used for explanations:

- System adaptation refers to the flexibility of the targeted system.
- Policy-system refers to the policy entity that includes all policy-related functionalities and components.
- Policy behaviour refers to the policy-system’s own flexibility.
- Contexts refers to environmental attributes that can be monitored/affected.
- Policies refers to rules together with their internal values that shapes the adaptive system behaviour.
- Policy-engine refers to the policy handling system.
- Policy repository refers to a database of stored policies.

2.3 Policy expression

The concept of using policies is very flexible and broad, and can be expressed in many ways. This has been shown by the many research designs that has been implemented over the years. Some implementation approaches will be discussed later in this report.

There are two distinct layers of policy types, namely low-level and high-level. Low-level policies are simple rules that are directly transparent to the underlying system. An example being: "If temperature at CPU is high, then lower CPU frequency". This has a very clear condition and a very clear consequential action which makes it simple and predictable. A downside of these types of policies is that they are very specific for a local component. Low-level policies are therefore quite analogous to specific policies.

Higher level policies are interesting because they allow more general control of the system. A simple example could be If battery is low, lower power consumption. Here the condition is clear, but the action is not as clear. The action is instead a high-level defined state that the policy engine must interpret and propagate to low-level decisions. In this case: lower power consumption might lead to Lower CPU frequency + disabling of non-critical components.
2.4 Policy behaviour

2.4.1 Three restriction levels of policy behaviour

A policy system’s internal behaviour (not to be confused with target system adaptation) is mainly governed by the policy-engine and the policy rule definitions. Regardless of how the policies are expressed, the policy-engine decides how to treat the policies. The behaviour can strongly vary depending on design. Dr. Anthony [10] describes three fundamental classes of policy-design approaches which are encompassed by the general policy-concept as described in section 2.2:

- Static
- Open-loop
- Closed-loop

Each of these general approaches can be realized in a number of ways, which are further detailed below. Anthony also discusses the role of using templates which acts as policy rule configuration parameter sets. This concept abstracts away the parametric settings from the policy rules which then becomes statement shells. A template then works as an input to a policy rule set that together instantiates a complete policy set. This allow separate handling of parametric tuning of the policies.

**Static policy configuration**  A statically defined policy is governing the system adaptation. The policy system is not prone to change itself. Policy configuration can either be embedded in the policy logic, or separated in configuration templates that are statically loaded between executions. The latter allows policy settings to easily be altered/swapped between execution times.

**Open-loop policy adaptation**  The policy system is prone to identifying inefficiencies and acting upon it. However, the policy system will not autonomously alter the rules. This can be realized by notifying the user on detected inefficiencies, who then may manually alter the behaviour. A second approach is to allow the policy management system to choose the most efficient template between several pre-defined static templates. The latter solution is a more autonomous design, although the level of flexibility is determined by the coverage of the pre-defined template-collection. The latter solution can also be seen as an expansion to the second approach of Static policy configuration”.

**Closed-loop policy adaptation**  The policy system is prone to identifying both inefficiencies and conflicts. It then automatically acts to solve detected issues by changing the policy-behaviour to an appropriate state. The change can be parametric (e.g. modification of policy values/templates) or rule based (modification of executed policies). Six approaches are described:

- Use a meta-policy for automatic selection between rules. Conflicts are solved by statically prioritizing every policy.
- Allowing autonomous adaptation of a policy to solve inefficiencies or conflicts either by temporary policy removal/modification.
- The policy-system is allowed to persistently change the policy rules or policy template.
- The policy-system is allowed to persistently change the policy rules or policy configuration/template, as well as saving the current policy state for policy recovery.
- A meta-policy is implemented for appropriately switching between several pre-defined policy alternatives.
- A system using both multiple templates and a meta-policy together. This is a coalition of the previous approach and the latter open-loop policy adaptation approach”.

11
2.4.2 Scopes of adaptation

Jorge Fox [11] discusses some classification concepts in terms of adaptation. These categories will be introduced in this subsection.

**Static adaptation** Static adaptation means that possible changes to the system are defined before system initiation or that adaptation requires some level of redeployment.

**Dynamic adaptation** Dynamic adaptation is the opposite of static adaptation. It means that changes may be propagated after deployment, in runtime.

**Parametric adaptation** Adaptation is achieved by tuning predefined parameters in target system.

**Compositional adaptation** Adaptation is achieved by replacing, removing, or inserting functional blocks/components in the target system.

2.5 Policy Types

In this report, policy types refers to how a policy is propagated into decisions. This is not to be confused with policy behaviour in section 2.4 which is more about how the policy-system treats decisions. Kephart and Walsh [12] were prominent in defining the different policy types that can be applied to a system. They defined that there were Action, Goal, or Utility policies. These three type definitions have been widely used as an inspiration after they introduced the concept in 2004. The three policy types defined by Kephart and Walsh will be briefly introduced in the remainder of this section.

- Action policy
- Goal policy
- Utility Policy

2.5.1 Action policy

An action policy, also known as ECA (Event-Condition-Action) policies, is used to execute a defined action in the current system state. The system state is referring to the collective state of all the monitored and effected resources. The current state can go to a set of future states, as shown in Figure 2.4.
propagates from one state to another in every change that is caused by an action. The action policy is simple and is usually defined as an \textit{IF(condition)-Then(action)} function where the condition is a set of states and the action takes it into another state. The solution is simple, yet useful. However, it is susceptible to direct conflicts if one is not careful, e.g. policy 1 wishes to power a cooling fan due to high temperatures, while policy 2 wishes to turn off the cooling fan due to low available power.

\subsection{2.5.2 Goal policy}

A goal policy is used for propagating the system from the current state to a desired state. The desired state is not necessarily a valid next state, instead the system might go through several changes to reach its desired state. Such a policy is usually constructed as a desired constraint and an action which takes it closer to the desired constraint. An example being \textit{If} (\texttt{ServiceResponseTime}) > \textit{X}, \textit{then increase its CPU-timeslice}. The statement will yield true until the condition is met. Having multiple goal policies which correlate to each other might cause instability if they begin to counteract each other. This can happen if the desired system state is unreachable.

\subsection{2.5.3 Utility policy}

Utility policies are different from goal policies in the sense that utility policies do not have a desired state as a goal. Instead it attempts to continuously optimize the system by going to the most desirable next state among the valid states from the current state. An utility policy chooses the state by calculating an utility function value such as \ref{equation2.1}.

\begin{equation}
U = (R_1 \times W_1) + (R_2 \times W_2) + \ldots 
\end{equation} \hfill (2.1)

Where \(U\) is the calculated value, \(R\) is the related resource value, and \(W\) is the weights associated with a resource type. Because the utility policy does not attempt to bring the system into a specific state, it does not attempt to enter an unreachable state. However, an utility function can be difficult to specify.

\subsection{2.6 The RRR Model}

The MAPE-K strategy model described in \ref{subsection2.1.1} is a widely used strategy in self-managing systems. It offers a straightforward approach and a simple structure, however it is quite general. \cite{9} suggests that there is a need for a model which better handles the abstractions of an autonomic system.

For this reason, \cite{13} discusses a model which abstracts the adaptation in several layers. The model is inherited from autonomous machine designs and is part of the \textit{Intelligent Design architecture} \cite{14}. It consists of three layers: \textit{Reaction, Routine, and Reflection}, thus \textit{RRR}. These layers are depicted in \ref{figure2.5}. Each layer of the RRR model represents a level of autonomous consciousness.

**Reaction** This layer is the lowest layer and communicates with sensors and effectors. Its purpose is to monitor and react to changes based on hard decisions. Its decision making is straightforward, often being an IF-ELSE statement. The simplicity also makes this the fastest executing layer. As this is the lowest layer, it also holds precedence over the other layers.

Consider an example where the reaction layer holds a mechanism to cut all power supplies in case of a power surge. A reaction policy rule at this layer could then be similar to the example in \ref{equation2.2}.

\begin{equation}
\text{IF} \ (\text{VOLTAGE} \ \text{MAX-VOLTAGE}) \ \text{OR} \ (\text{CURRENT} \ \text{MAX-CURRENT}) \ \text{THEN} \ (\text{SWITCH-OFF}) 
\end{equation} \hfill (2.2)

This policy should hold precedence over other adaptation policies as its critical for system safety. It is therefore best suited as a reaction policy.
**Routine** The middle layer, routine layer, is called if the reaction layer does not find any suitable policies. The routine layer holds more advanced policies which often involves a set of policies being evaluated to determine which approach is most suitable. It has input from sensors, requests from reaction layers, and commands from reflection layer. The reflection layer is only called if the routine layer is unable to reach a verdict. This means that the routine layer holds precedence over the reflection layer. The routine layer can also access the policy repository, system effectors, and the reaction layer. This layer has a comparatively longer execution time compared to the reaction layer due to its more advanced analytical properties.

If a decision is made by the routine layer, it may propagate it to the managed system’s effectors, and also push the policy to the reaction layer. This policy can then be used by the reaction layer next turn it is needed. In this way, the reaction layer learns.

**Reflection** This layer is called if both the reaction layer and the routine layer fails to find a suitable policy, or can’t reach a decision due to conflicts. The reflection layer is then meant to deal with these abnormal situations. The reflection layer must create a new policy in the case that there are no suitable existing policies. This process will have to be based on learning technologies (e.g machine learning, genetic algorithms, artificial neural networks) and/or reasoning strategies (e.g. Fuzzy Logic, Bayesian reasoning). The new policy will then be written to the policy repository and executed by the routine layer.

In the case that multiple policies are applicable in the routine layer, the reflection layer will be called to resolve the issue and provide with a course of action. This procedure will therefore require some form of conflict handling.

It is a justified assumption that this layer is the most computer heavy element. However, this element is indispensable for large and complex systems. The alternative to the reflection layer is to demand human intervention for extending the policy support/coverage in the policy repository, although this is not a favourable approach for a self-sustaining autonomic system.
2.7 A Policy standard

There have been many attempts to standardize, categorize, and quantify policy systems. This is however difficult due to the broad usage-area where a policy system can be applied, and the diversity in policy behaviour and expression as seen in section 2.4 and section 2.5. One proposed solution is discussed in [9] which tries to categorize and quantify autonomic systems in 5 levels, based on IBM’s MAPE-K strategy discussed in subsection 2.1.1.

There is yet no standard for policy-based adaptive systems, although there are standards for neighbouring technologies such as the RFC-3460 [15] standard for policy-based network management. These standards are not discussed in detail in this report, some parts are however included.

The RFC-3460 standard is mature but still prone for improvements, meaning that it is open for inputs. It is mainly for adaptive network management using policies, and is therefore customized for that area. The policy types defined by this standard are: Configuration, Installation, Error/Event, Usage, Security, and Service Policies. These policy types have different usages and are distinctively unique to each other. Because of the specific usage area of the RFC-3460 standard, these policy types does not suite as a general foundation to policies [16].

An interesting aspect is that all policy types described in RFC-3460 are governed by the same policy structure which is composed of several subclasses. RFC-3460 describes a subclass structure for a generic policy. This generic policy structure can be applied to policies in general, even outside of the RFC-3460 scope [16]. This structure can be seen in a condensed format in Figure 2.6. The full structure is not included in this report, but it is essentially similar. The remainder of this section will include explanations to these subclasses.

**PolicySet** The PolicySet is the core of a policy which defines the policy decision structure. It is extended with two subclasses PolicyGroup and PolicyRule. PolicyGroup is a class which indicates which PolicyGroup blocks a policy is tied to, in order to distinguish related policies. An example might be to collect policies with related atomic operations. An atomic operation is seen as the operation with direct effect to the system, e.g. "increase voltage input to the cooling fan". All policies related to any of these atomic operations would then be tied to a respective PolicyGroup.

The PolicyRule subclass is a primary entity in the structure by which system goals are actioned. This could include a sequenced set of actions and an execution strategy. An execution strategy is a user-defined strategy for defining how a sequenced action set is executed. The PolicyRule might also contain a static priority setting and an enabling flag. The priority setting might be used for conflict handling, although this is discussed later in the report. The enabling flag is used for disabling/enabling a policy for execution, which is useful for conflict handling, administrative convenience, and/or debugging.
**PolicyCondition**  The *PolicyCondition* class is an auxiliary class to the *PolicyRule* class. For a policy to effect an action, certain conditions has to be met. These conditions are specified in the *PolicyCondition* class. This class is extended by the *PolicyTimerPeriod* subclass, and the *VendorPolicyCondition* subclass. The *PolicyTimerPeriod* adds a time constraint to the policy, which could for instance be a time interval during the day of which the policy is allowed to execute. The *VendorPolicyCondition* is used for device/component specific class extensions to the policy conditions.

**PolicyAction**  This class contains the actions resulting from the *PolicyRule* class. It can also be extended with a *VendorPolicyAction* similar to the *VendorPolicyCondition*.

**PolicyVariable**  This class defines internal policy variables used by the policy. These variables are invisible to the targeted system, and are only used for decision making.

**PolicyValue**  This class defines values for the *PolicyVariables*. These can either be hard-coded or user-specified. Anthony [10] suggests using templates for maintaining these values.

### 2.8 Realtime support in policy-based systems

Since embedded systems and real-time support are two closely related topics, real time support becomes important for policies for embedded systems. Policies govern some flexible or adaptive aspect of a system, meaning that policies can affect the real-time properties of its targeted system. In the same way, policy-based systems may also take advantage of real-time concepts to improve the service value that policy-based design provides. An issue with the current state of art in policy based design is that there is little support for specifically real-time performance, which is discussed later in the report.

The classical concept of realtime support will be introduced in this section. Realtime support is a crucial property for many critical systems in our society. It generally involves processing data, and giving a response before a set deadline. A car’s braking system is for example very dependent on getting quick system responses before a set deadline, failures to do so can lead to catastrophic misfortunes. Same ideology can be applied to many technologies, such as air-plane stabilisation, ticket systems, temperature controllers, mobile phones, and also face recognition systems. The impact of missing a deadline varies depending on application, but the concept is nonetheless the same.

For face recognition systems that processes data from a monitored live feed, it can be crucial for the system to respond with a result before the situation is no longer valid. A system which would detect a human face and graphically mark it with a square should consistently produce a frame feed which is still relevant to what is happening in the real world.

In classical realtime systems, two important characteristics are determinism and deadlines. Realtime computing can be separated into three categories which sets different requirements on the system [17]:

- **Soft realtime**  A task is not required to meet its deadline, but its value degrades after passing it. This is usually used for non critical tasks, such as video encoding.

- **Firm realtime**  A task is not required to meet its deadline, but its value becomes zero. This is usually used for non critical, but timing dependent tasks, such as video streaming.

- **Hard realtime**  A deadline must be met, or the system fails. This is usually used for system critical tasks, such as a car’s braking system.
2.9 Discussion

There are many aspects to consider when implementing a policy-based system, both in terms of system design, expression language and use case. The above theory has shaped the current state of art in policy-based design. Originating from self-managing and adaptive systems, policy-based design has developed its own framework of definitions. The definitions from this chapter are important for the reader to be aware of, as these will be reoccurring throughout the report. There are several challenges that needs to be taken into account when considering to design a policy-based system. Some challenges such as conflicts and real-time has already been introduced. In the following chapter, these will be discussed more in detail together with some other challenges.
Kapitel 3

Challenges in policy based design

When designing a policy-based system, one needs to consider some aspects that may prove to be important for the design. In this report, these aspects are referred to as challenges since these elements may cause undesirable system behaviour unless being accounted for. The challenges that this report will discuss are:

- Verification and validation
- Resource sensing
- Conflicts
- Real-time performance

3.1 Verification and validation

Autonomous systems are developed to work independently of human intervention, meaning that any action taken by its adaptive behaviour is dictated by the system itself. Policy systems suffer from an additional uncertainty, namely the policies themselves. Since policies are designed to be changeable even after the product delivery (i.e. after when the developers has completed the product) it is hard to verify the system during design time. Because of these difficulties, quality insurance using general standards might not suffice [13]. This makes verification and validation difficult for the construction of dependable policy-based adaptive systems.

One step towards counter-measuring faults is to constrain the presence of unexpected states that a policy may introduce [10]. This can be done by explicitly defining the potential outcomes of a policy, e.g. output must be an integer between 1 and 5. This makes each policy more reliable as the output can always be statically tested. The downside of this effect is that it also constrains the potential flexibility of a policy. Furthermore, it is not a viable option for the reflection layer which holds more advanced algorithms.

Another potential solution is concluded in [18], which discusses a model-based run-time approach for policy-based adaptive behaviour. The principle is to beforehand create a system model that comprises the parameters that can be monitored or controlled in the system (e.g. power consumption), as well as the characteristics of each parameter (e.g. relation between power consumption and CPU-utilisation). The policy-management system will then evaluate the policies to the system-model before any real adaptive behaviour executed. If the outcome of the model is outside of the model constraints, then the policy system is forced to re-evaluate based on the model-test outcome until a valid state has been found. Using this technique allows system validation to be statically analysed by evaluating the system architecture together with the system-model. The idea is that this should abstract away the validation process from the policies. The downside of this technique is that it requires a great deal of computing power.
3.2 Resource sensing

Systems that adapt to optimize the instantaneous utility of the system based on the exact current state are likely to show unstable characteristics due to system fluctuations [19]. There are two types of system behaviour which should be accounted based on [19] and [10]:

- Close-threshold fluctuations
- Temporary and random spikes

3.2.1 Close threshold fluctuations

Close-threshold fluctuations might cause an adaptive system to undesirably frequently switch between system states [10]. This is caused by frequent threshold crossing of a monitored system property. For example: a temperature control system which toggles fans on/off at a given temperature might begin to repeatedly switch between on and off if the monitored temperature is close to the toggle threshold. The same problem could occur for a meta-policy if one assumes that the meta-policy toggles between different policies depending on a resource parameter.

A simple solution is to introduce dead-zones for every decision threshold in order to reduce the switching rate [10], as depicted in Figure 3.1. However, this means that dead-zone levels has to be defined.

3.2.2 Temporary and random spikes

Temporary and random spikes are harder to address due to the unpredictability of such occurrences. It can happen when a monitored resource momentarily jumps. This could be because of e.g. sensor-faults or system-glitches [19]. The issue lies in the fact that the system might undesirably adapt the system to the instantaneous resource state in a spike of the monitored system resource. Depending on the adaptation implementation, this may cause unpredictability in the system. Since the adaptation system requires time to evaluate and propagate the system modifications, it might happen that the triggering conditions for the adaptation is more short lived than the deployment time for the adaptation, potentially resulting in unnecessary resource costs and unwanted service disruption. An example is illustrated in Figure 3.2.

An approach to more reliable resource sensing in terms of countering spikes is to predict the resource value based on historical data. A very simple approach would be to calculate the mean-value of N

---

A policy which acts as a switch for selecting a policy within a set of policies.
Figur 3.2: A adaptation is evaluated based on a context caused by a temporary spike

measurements \cite{20}, like Equation 3.1 which takes the 5 latest measurements. However, more advanced algorithms might be interesting depending on system requirements.

\[
X = \left( \sum_{i=1}^{5} X(t-i) \right)/5 \tag{3.1}
\]

\cite{19} also discusses three aspects of resource prediction:

- Linear recent history
- Relative Move
- Bounding

**Linear recent history**  Linear recent history involves linear prediction of historic data. This could be any algorithm such as the trivial solution mentioned above, or maybe a more sophisticated ARMA model.

**Relative Move**  Relative move is about knowing the future based on historic trends. In this context, trends are seen as behaviour which is frequently occurring when triggered by a specific source. An example being: *If a server is frequently overloaded in the morning, one can assume that this is a trend. Once a trend if found, one can suppress its negative impact by adapting accordingly. In this example it could be to increase the server capacity during the morning period.*

**Bounding**  Bounding can be used to specify maximum and minimum values of a resource. A value above the maximum or below the minimum will be assumed to the extremity value of the boundary. Meaning that all values below or above are suppressed to a controllable value.

### 3.3 Conflicts

Conflicts are undesirable, but can occur when multiple policies counteract each other. This can occur due to multiple reasons, as described in section 2.5. The risk of conflicts increases as the number of defined adaptations are increased \cite{12}. Avoiding conflicts also becomes more difficult depending on the system complexity and the number of adaptation points. The consequence of conflicts are mainly dependent on the relation between the adaptation points of the involved policies. Since this is dependent on the system design, the resulting consequences are very unpredictable when policies are defined. A system could easily end up in an unstable state in this way.
One solution for solving conflicts is through the use of priorities. Priorities then acts as a basis for making decisions regarding which policy should hold precedence and dominance (may not be overwritten by lower prioritized policies). This is discussed further in the following section.

3.4 Policies in terms of timing

To deploy an adaptation, time is consumed to evaluate and propagate the system modifications. This means that online system adaptations always has a negative impact element on overall system performance in terms of timing. The worst case scenario in terms of timing is synonymous to the longest possible policy evaluation time, i.e. decision making time [21]. The adaptation system should therefore have some kind of timing-awareness. This report will discuss the following two areas in this context:

- Determinism
- Scheduling policies

3.4.1 Determinism

An important characteristic of real-time system is the determinism of the system itself. Determinism is about knowing how much time a task requires.

A common approach to evaluating the determinism in a system is to decompose the system into its sub components. When it comes to flexible systems such as a policy-based adaptive system the question becomes: Where is timing uncertain?. This is always dependent on the implementation approach of the system. However, one common denominator is the policy-evaluation mechanism. Since policies have no fixed size or complexity, it becomes apparent that it is also not deterministic. Because policy evaluation has an undefined execution time, this report will focus on that element. As mentioned earlier, evaluating a policy will always consume time. The time consumed is dynamic and often unpredictable. It is therefore important to characterize the policy evaluation process in terms of execution time.

Policies execution times varies heavily on the policy-script that defines the adaptive behaviour. Policy-scripts ranges from very simple to very complex, e.g. a policy may be defined as simply a single ECA process, it may also be a long chain of actions, rules, and policies. In the latter case there might even be iterative loops contained in the logic, thus making the execution time even more unpredictable. There might not even exist a deterministic longest-execution-time, in which case the adaptation process becomes unreliable. This section will discuss three perspectives in terms of execution time and some related solution approaches, namely:

- Iterative loops
- Deadlines
- Policy complexity

Iterative loops

Iterative loops are in this report defined as chains of actions, rules, and policies that execute an undefined number of times within one adaptation step. As these are unreliable in terms of execution, i.e. unpredictable, they need to be carefully considered in the context of real-time systems. Unless the adaptation system includes a policy-generating/changing reflection layer, policies may be manually evaluated to avoid iterative loops. [22] Discusses the usage of loop-detection for the reason of conflict detection. Their implementation is based on a Trigger graph which shows all possible propagations of a policy. The graph can then be analysed to detect any form of loop. The details of this solution are not included in this report.

Another point of view is to always ensure a response within a set limit of time, i.e. a hard deadline. The hard deadline can in that case be seen as a worst case time scenario. However, this introduces the issue of always ensuring a response.
Policy complexity

Since the policy evaluation can be an indeterministic element in a policy-based adaptation system, the relative time-consumption for this task is also potentially the greatest. The greater the adaptation complexity is, a longer response time can be expected, and a higher flexibility is possible. The reflection layer is therefore not suitable for adaptive systems that requires short response times [23], however, it might be useful for an increased adaptive behaviour. The longest response time for an adaptive system is defined by the highest complexity layer. A system that does not contain a reflection layer is likely to give a faster response time than one with a reflection layer. Same goes for the routine layer, however this layer is often indispensable as it is the backbone of the dynamic adaptive behaviour.

3.4.2 Realtime scheduling

The timing required to execute an adaptation have similarities to required timings by tasks in a CPU-scheduling process. In task scheduling, the focus is on task deadlines where a deadline can be seen as a depletion of a resource. The critical resource is classically the CPU time-slice given to a task for which the task may execute. The task must then finish its execution before the time-slice ends, otherwise it counts as a missed deadline. Once a task is finished executing or its deadline has been met, a new task takes its place.

In terms of policy-based adaptation, one can consider the policies to have a time-resource which reduces the policy-value when consumed. The longer the policy evaluation takes, the less valuable is the propagated adaptation, i.e. the change is not worthwhile the time-cost. The value will diminish until the execution is done, or the adaptation is deemed useless for which the process may be interrupted in favour of system-wide real-time continuity.

A CPU task-scheduling strategy is commonly dictated by task priorities. The strategy involves associating every task with a priority. The tasks are then executed based on their priority-levels, allowing higher prioritized tasks to execute before lower ones.

[24] Discusses the use of priorities within policies. Their concept involves prioritizing the policies based on their criticality to the system, where critical policies are given a high priority, and optimizing policies are given a low priority. If the conditions for a high priority policy are met, that policy will have precedence over other competing low priority policies, e.g. turning off a system in case of overheating has precedence over tuning some parametric values. In their model, a policy can only be given the priority high or low, no values in between. This can be seen analogous to the reaction layer of the RRR model, where high priority tasks are reaction events. Similarly to their priority model, policies can be given a priority based on time-criticality and adaptation value.

There are two distinct types of classic scheduling strategies, namely Fixed priorities and Dynamic priorities [25]. The scheduling techniques are discussed below and put in contrast to policies. Note that priority based scheduling techniques are relevant to policies if for e.g. multiple policies are requested to be evaluated by the policy engine concurrently. The evaluation order might then be governed by priorities.

Fixed policy priorities

Fixed priorities means that priorities does not change over the the course of time. Two classical examples are rate-monotonic, and deadline-monotonic scheduling. Rate-monotonic scheduling involves prioritizing each task based on its rate, i.e. inverse of task’s period.

Analysis in terms of policies In terms of policies, deadlines are arbitrary per nature unless a deadline is manually specified in the code or policy. The period of a policy may also be arbitrary depending on implementation strategy. A policy which is only evaluated when it is needed, e.g. Anthony’s principle of using decision points, has a period equal to the period between calls from the associated decision point. Thus making it dependent on the structure of the targeted system, and the place of which adaptation is made. In that case, policies have a mixture of periodic and sporadic calls, thus making rate-monotonic scheduling difficult to adopt unless a periodicity is specified.
Another aspect is to consider a static priority distribution based on the reaction, routine, and reflection model. As earlier described, there is a distinct relation between RRR layer, execution time and adaptation potential, where higher layers indicate a longer execution time and higher adaptation potential. There is also a relation in terms of system criticality, where the lower layers should hold rules that have a higher system criticality than the above layers. This is because system critical decisions should have precedence over other adaptations. System critical decisions are defined as decisions that has to be enacted to avoid a system crash or malfunction.

For real-time systems, response time and system continuity is of utmost importance. One could therefore consider priorities with respect to policy response time and/or system criticality. The reaction layer holds the highest priority in both cases, followed by the routine layer, and lastly the reflection layer. One could consider adopting concept of RRR to policies by considering each layer as a subset of all policies where each layer holds a corresponding priority set. The concept would be to associate routine policies with a high priority, whereas routine and reflection policies would be associated with a medium and low priority respectively, as depicted in Figure 3.3. Each priority layer could also be a subset of priority levels. Priorities can be advantageous in terms of conflict handling, and real-time performance.

**Dynamic policy priorities**

Dynamic priorities means that priorities among tasks are changed over the course of time. Two classical examples are earliest-deadline first, and least slack-time first. The earliest-deadline first strategy involves shifting the priorities according to their absolute deadlines, i.e. which deadline will occur first among all tasks. The least slack-time first strategy involves giving the priority to the task who’s slack time is the smallest. Slack time is calculated as the time given by the time until deadline, subtracted with the remaining time of required execution.

**Analysis in terms of policies** In terms of policies, execution times are difficult to predict due to their dynamic behaviour. It would either require a static execution time analysis for each policy, or runtime logging and saving of longest execution times per each policy. The earlier assumes that policy does not change or are created in runtime (apart from parametric changes), while the latter requires a solution for a persistent modification of policy properties.
3.5 Conclusions

Four aspects of challenges has been discussed above. The importance of each aspect varies heavily on the specific implementation and use case. Nonetheless, each aspect should at least be considered on some level, even if it is not implemented. A short conclusion on each challenge is given below.

3.5.1 Validation and Verification

Validation and verification is a difficult topic, and there are no ideal solutions as of yet. Since policies are defined by human operators, they become more error prone. Unless the operator is carefully considering the correctness of the policies and correlations between each adaptation point, verification becomes difficult and requires a built in feature.

3.5.2 Resource sensing

The importance of considering issues with resource sensing is very dependent on the specific implementation. If a context is read from analogue sensors, then spike handling should be required. If a context is responsible for a big adaptation change, then close-threshold fluctuations should be considered.

3.5.3 Conflicts and realtime support

As earlier discussed, conflicts are usually handled by some kind of a prioritization. Priorities can be set either statically or dynamically. Both options are viable for the final design. However, dynamically defining the priorities using utility functions requires more knowledge of the point of adaptation. In some cases this can be difficult.

One also needs to consider which characteristics that should govern the priority settings. For real-time embedded systems, the prioritization concept based on the RRR-model subsection 3.4.2 was discussed as a potentially suitable approach. Other approaches such as rate-monotonic scheduling, or earliest deadline first scheduling were also deemed suitable. A form of prioritization for execution should be included in the implementation to introduce a real-time aspect to the system.

3.6 Discussion

Agile Lite is a viable alternative to coding a custom framework. Since the scope of this thesis isn’t about parsing, creating a custom framework would give very little yield. Therefore, Agile Lite has been chosen as a basis for the script handling system in this thesis. Agile Lite also holds several functionalities which approaches some of the requirements, which is another big benefit.
Kapitel 4

Related work in policy-system design

4.1 introduction

Including self-* behaviour into systems is an interesting field of study as it excludes the need of human interaction to some extent. This concept has been widely researched and some has discussed the potential of using policies. As self-* systems are interesting for many fields, policies have been used in many different levels. Therefore, many researches in policy-based computing are not related to resource constrained embedded systems for real-time applications, and are instead implemented for other reasons such as: smart home systems, smart mobile service selection, etc. All implementations might therefore not be fully suitable to be included in this thesis, however, some design aspects might be interesting to investigate. The rest of this chapter will discuss a selection of policy-based adaptive systems and relate them to earlier chapters.

4.2 A framework for adaptivity in pervasive computing

A framework for policy-based self-adaptivity in pervasive computing is discussed by Jian Quan in [24]. Their goal was to propose a policy ontology and language which can be used in pervasive computing.

Their work was motivated by four characteristics: expressiveness, well-defined semantics, usability, and lightweight. For this reason, their framework is based on the ECA concept (Event-condition-action), separation of concerns (abstract away the decision making logic from the application).

The core idea for their approach is to have a context manager, which sends system information to a policy controller which matches the data to a set of policies. Those policies whose conditions are deemed to have been met will trigger an event monitor. The event monitor will in turn notify a policy executor which evaluates the policy and propagates it into actions. Figure 4.1 illustrates their model. The adaptation is triggered by context changes, meaning that all relevant policies to a changed context will be checked. Since multiple policies can be triggered by the same context, it also means that multiple policies can yield a change per iteration. For this reason, they implemented a priority-check which is embedded in the policy XML-script itself. Each policy can be associated with a priority, contexts, timings, conditions, rules, and actions. Priorities may be defined either high or low, where a high prioritized policy will always have precedence over a low prioritized policy. Their research resulted in a prototype implementation which was deemed successful and useful, however, it was never released as a product.
Discussion

The model suggestion is context change triggered, meaning that policies are evaluated based once a related context is changed. To avoid conflicts, static priorities are defined for each policy where the highest prioritized policy holds precedence and dominance. The model can be seen as parametric, dynamic and open-loop adaptation. The challenges of ensuring real-time performance, and validity & verification are not explicitly addressed by this framework.

4.3 Dyscas/Shape

4.3.1 Introduction

Dynamically Self-Configuring Automotive Systems (DySCAS) is an European research project for developing a self-configuring system targeted for automotive that began 2008 (and ended 2008). The project was funded by the European Commission’s Sixth Framework Programme for Information Society Technologies. Project partners involved: Volvo Technology, Enea, Bosch, The University of Greenwich, University of Paderborn, and the Royal Institute of Technology. The project had an emphasis on dynamic self-configuring middlewares for embedded systems.

In contrast to more conventional static design-time middlewares, the DySCAS solution introduced adaptability to system changes both offline and online. The system would respond to environmental changes in terms of installed hardware, loaded software, and resource availability. This solution would boost a system’s robustness, performance, and fault-tolerance, much due to the ability to change in runtime.

4.3.2 Adaptability concept

A solution given by DySCAS project involved policy-based system configuration using decision points who’s definition and behaviour were abstracted away from the internal source-code. This would allow deferring some design decisions during development, as well as providing high customizability, having small foot print, and high portability.

The policy handling system would be separate from the rest of the core system. It would consist of:

- Decision points for interacting with the core system.
- Dynamic wrappers to support each decision point’s interaction with policy handling system.
A decision evaluation module (DEM) for evaluating policies that has been assigned to a specific decision point.

- A context manager for handling environmental variables.
- A repository service for maintaining policy rules.
- A policy manager for overall policy system maintenance.

Decision points were implemented for deciding upon which static code block would be executed, e.g. a choice could be to execute `Load_Software` or `Deny_Software`. The decision point would then be inserted at the choice junction. A policy would be evaluated to determine which path to choose. Policy evaluation is based on the policy rules defined in the repository, and context information supplied by the context manager. Each decision point would also be accompanied by one dynamic wrapper. The dynamic wrapper acts as an interface between the decision point and the policy handling system, while also protecting the decision point from faults [27].

**Discussion**

The system is decision point triggered, in the sense that the policy evaluation is performed as a certain point in the software is reached. The solution can be seen as dynamic, compositional, and open-loop adaptation. Conflict handling is inherited through the use of policy suites and decision points. The challenges of ensuring validity & verification are not explicitly addressed by this implementation.

**4.4 MUSIC**

**4.4.1 Introduction**

In 2009 Romain Rouvoy et. al [28] introduced a middleware adaptation framework for self-adaptive system behaviour. Their solution is targeted for service oriented architectures (SOA), such as mobile hand-held smart phones. The multi-functionality and performance growth of today’s smart phones has been the motivation behind this research.

The research is embodied with the MUSIC project which has resulted in an open-source middleware solution written in Java. The project aims at developing an open-source platform which enables the development of self-adapting and reconfigurable applications.

**4.4.2 Adaptability concept**

The MUSIC middleware solution allows dynamic and adaptive self-configuration of services provided by the device. In this context, services relates to technologies such as GPS and WIFI that are utilized by running applications on a mobile device.

The flexibility of the system lies in the dynamic service/component realizations. Each service might be realized in different ways by deploying different technological solutions that provide similar functionality but with different QoS characteristics. The system does not take advantage of any policy-based concept, however, it is interesting in aspect of adaptation. Adaptations are propagated as contexts change.

- Each service realization is determined based on the current needs (contexts).
- Needs are determined by the user via applications / device configurations
- A planning module is used for meeting the system needs.
- A planned configuration is chosen with regards to its utility-level for serving the running applications.
- The utility-level calculation is based on the QoS properties predicted by the required services and applications.
• QoS property values for services/applications may be static or dynamic; utility-levels are recalculated between each iteration.

Discussion

The system is context change triggered, in the sense that application service-demands can be seen as contexts. To avoid conflicts, priorities are calculated based on utility functions, where the highest scoring state is enacted. The solution can be seen as compositional and static adaptation. The challenges of ensuring real-time performance, and validity & verification are not explicitly addressed in the framework.

4.5 PobSAM

PobSAM (Policy-based Self-Adaptive Model) [29] is a policy-based implementation in Java. In their implementation, policies are used for the adaptation mechanism in their self-adaptive system. This is done via a set of self-managed actor modules. To enact an adaptation, policies are used to adapt the governing managers of the actors. In their sense, a manager can be seen as a meta-actor, i.e. a overlying actor for the below actors. The use case for their implementation is a Smart Home Network where each actor controls a specific home environment, e.g. lighting, temperature, etc. Contexts are read to the managers, which in turn adjusts each actor in order to conform to a wanted state. An example given by them: A sensor detects a person, the manager evaluates related policies, the output suggests to set room lighting to default, the associated actor for this action is notified and executes the action. The manager is context-triggered by a context monitor that continuously reads system contexts. Multiple managers with separate policy sets can exist in the same system. Figure 4.2 illustrates their concept of managers and actors. The middle layer acts as an administrator between actors and managers in order constrain each manager’s information pool so that it is only given information and control over what is required by the manager. Their implementation showcases a real-life application for distributed policy-based managers. The properties of policy-based computing gives the system extensive flexibility for customization of a smart home system.

Discussion

The system is context change triggered, meaning that policies are evaluated based once a related context is changed. To avoid conflicts, static priorities are defined for each policy where the highest prioritized policy holds precedence and dominance. The solution can be seen as parametric and open-loop adaptation. The challenges of ensuring real-time performance, and validity & verification are not explicitly addressed in the framework.

4.6 Discussion

The implementations are interesting as they show some aspects of the current state of art. There is a lack of advanced closed-loop (self-adapting) policy systems. Most frameworks only support up to open-loop adaptation. They also show the broad use case spectrum which policy based design supports. Short conclusions regarding the handling of the challenges and their design approaches are given below.

4.6.1 Handling of the challenges

There is little consideration for ensuring realtime support and validity & verification among the discussed solutions. Conflict handling is usually handled through some kind of static or dynamic priority property.
4.6.2 Ways of triggering adaptation

Two distinct approaches for adaptation triggering can be discussed based on earlier findings. Using decision points to trigger adaptation is one approach, while using context changes to trigger adaptation is another approach. Both approaches yield a valid adaptation, however there are some constraints to each of the approaches.

**Decision points** Decisions points can be used like in the Dyscas/SHAPE project. An issue with using decisions points is that it forces the targeted system to halt in order to wait for a response. The implementation is therefore prone to cause big overheads in contrast to normal execution. However, using decision points allows high portability of a policy-based system since the policy system may be isolated from the rest of the system. Thus avoiding high re-factoring costs for legacy projects.

**Context change triggered** Context change triggered systems, such as MUSIC and PobSAM, are interesting since it eliminates direct halts to the adapted code as change is directly propagated when a related context is changed. However, it might be difficult to directly adopt this approach to a scheduler since it would require cross process calls between the scheduler and the policy system.

**Discussion** Both approaches are interesting in their own ways. The approaches should be further investigated, and ways to improve current design alternatives should be considered in the following chapters. Both current approaches are prone to high and/or unpredictable overheads in the system which is an undesirable property.
Kapitel 5

An Interesting Framework

5.1 Introduction

Using ready to use frameworks is an invaluable resource to shorten development time of a system, but only if the framework is appropriate to use, i.e. there has to be some support for addressing the given challenges of the system. The same can be said about policy systems. However, among the policy frameworks, there is a lack of support for embedded systems. Most implementations are dedicated for high performance systems running e.g. Java. Some examples being found in [29], [30], [31], [10]. The most interesting framework for this project is Agile, since Agile is a ready to use product that also has a version that is customized for embedded systems, namely Agile Lite. This report will therefore discuss Agile and Agile Lite in terms of suitability for this project. Because Agile Lite is a lighter version of Agile, this report will first discuss Agile before discussing Agile Lite. Both Agile and Agile Lite are closed source libraries.

The chapter will begin with a short description of Agile’s features. These will then (subsection 5.2.5) be compared to the challenges and requirements that has been previously stated.

5.2 Agile

Agile provides a policy expression framework, together with an API. The API is available as C# and Java libraries. It is designed as a support for self adaptive behaviour through the use of XML defined policies, both for the controlled system and the adaptation system itself. It provides functionalities to manage, update, and evaluate policy scripts. These functionalities are essential for any policy based system. Implementation time of a policy system could therefore be significantly shortened if Agile could be used, which is a strong motivation for investigating this possibility.

In Agile, policy rules are stored in user-friendly XML-script files. The XML files are kept separate from the generated code-binaries, meaning that it is still accessible after software packaging. This solution gives a user-friendly method of reading, altering or adding policy rules to an already developed system.

5.2.1 Policy mechanics

An Agile policy is governed by actions, rules, and variables as seen in Figure 5.1. Rules are used for evaluating current system contexts given by the external variables. The external variables are evaluated in contrast to internal variables, other external variables, and/or fixed values. The external variables are representations of system contexts from the target system.

Rules can enact other rules or actions. Actions can be seen as a consequence of an evaluation. Actions can in turn enact other rules, other policies, templates, or return a response. Templates defines a set of internal variables that overwrites the old internal variable values that are within the set. Once a
return response is given to the target system, the policy evaluation is terminated (policy evaluation is complete).

5.2.2 Policy Structure

The highest element in a XML-defined policy is the PolicySuite, as depicted in Figure 5.2. A suite is defined for a specific use-case e.g. control the time-resource parameter inside a CPU-scheduler. The policy suite can contain multiple policies that are of the same category, i.e. policies for the suite’s use case. Each policy will enact a set of policy rules and actions based on current system context. Only one policy may be active at once, however, sequential policy enacting is allowed. A policy is denoted as NormalPolicy. See section 9.3 for XML examples.

The policy suite may optionally include a meta policy. A meta policy governs over which policy in the policy suite should be activated. The decision making is done through regular actions and rules inside the policy suite. Additionally, an UtilityFunction may be used for this reason. The UtilityFunction is composed of a set of options which are associated with an utility function. The option whose utility function yields the greatest value is enacted. A meta policy is denoted as MetaPolicy in Agile.
Several templates may optionally be defined to tune the internal variable values. Templates may be applied at any time in the process, and are used for adapting the policies through parametric changes.

*ToleranceRangeChecks* are used by policies to prevent potential oscillations in the adaptation process. To understand this problem, consider a case where the outcome of a policy is responsible for another adaptation. The new adaptation might lead to a consecutive adaptation. The *ToleranceRangeCheck* will determine a new threshold for which another policy rule is enacted. It can be seen analogous to a dead-zone described in subsection 3.2.1.

5.2.3 Library functionalities

This section includes a short summary of the functionalities offered by the Agile library API [32]. Some minor functionalities are left out.

**Policy loading** Policies may be loaded from XML into an object instantiation, which can then be evaluated.

**State persisting** Current internal policy values can be saved as a new template which can then be used similar to regular pre-defined templates.

**Policy evaluation** Evaluates a loaded policy suite script, returns a decision output.

**Setting values** Both external and internal values can be set, where external values are contexts and internal values are internal variables.

**Changing instantiated policies** Some properties of instantiated elements can be dynamically changed, for e.g. policy type, ToleranceRangeCheck parameters, return values, etc.

5.2.4 Decision Points

Agile has been successfully implemented by Anthony in adaptive systems in the past. Two examples are presented in [33] and [34]. Both of which are implemented using decision points. Decision points can be seen as a point in the targeted system’s code that will execute a decision logic once the point is reached in the system execution. Each decision point in the system is encapsulated by a dynamic wrapper. The wrapper acts as a local management module for the linked decision point. The wrapper is used for loading policies, retrieving context information, and internal error handling. In their case, they have a decision evaluation module which evaluates loaded policies from a repository service. The repository service holds the policy related XML scripts. Context’s are loaded to the decision point in accordance to subscriptions that the dynamic wrapper requests. Figure 5.3 shows the concept of their implementation that was described.

5.2.5 Discussing Agile’s adaptation support

The adaptation support for Agile is an important characteristic to analyse in order to evaluate its usefulness in a policy-based adaptation system. In this report, adaptation support refers to the level of allowed adaptation. This section will discuss different features and constrains associated with adaptability.

**Expression language**

*Advantages:* The Agile expression language offers extensive customizability by supporting a broad spectrum of configurations in terms of action-rule chains. This allows high scheme complexity if necessary for the adaptation. Additionally, having a defined set of return value outputs increases the testability of the adaptation system, since it diminishes the potential system states to a finite number.
Constraints: The expression language requires a defined set of valid response outputs, meaning that policies may not return an arbitrary value. The output can therefore only propagate into a design-time defined set of effects, such as increase/decrease, or on/off. This restrains the possibility for e.g. control algorithms as policies since these usually returns an adjustment value. In that case, a policy is restrained to adjusting logical behaviours for a control algorithm, such an implementation is presented in [33].

Default values

Advantages: Agile uses default values to cope with faulty policies. If a fault is detected, the result will be the default value.

Another interesting aspect this brings is the possibility to create a hard deadline mechanism. Consider a policy with a pre-defined deadline and a pre-defined default value. The policy engine will attempt to evaluate the policy, but given a realtime clock, the policy engine could revert to the default value if the adaptation could not be made before the deadline is passed. Alternatively, the adaptation finishes before the deadline, and no further action is needed (adaptation has been successful).

Constraints: Default values must initially be statically set.

Policy suites

Advantages: Policy suites can be used as a clustering of policies which share the same effector. This is useful for quick access to related policies, and also for conflict handling since the collective of policies in a policy suite can only return one output to the effector.

An interesting aspect this feature brings is the possibility to define an array of twin-policies that enact adaptation on different complexity levels. Thus giving each policy a different ratio between response time and level of sophistication in its adaptation. The PolicySuite in Agile (not Agile Lite) also supports the use of utility functions, which can be used for determining the most suitable policy alternative based on e.g. time criticality, deadline, etc. Together, this could be used to implement a complexity-aware adaptation.

Constraints: The implementation of policy suites also means that the user will need to have precise knowledge of where to implement new policies for the expected results. As a result, the user needs to know how the other policies within the suites functions.

Meta-policy

Advantages: The use of a meta-policy is an useful concept since it allows an adapted change depending on the context. Together with an utility function it can also serve as an evaluation of policy selection. Thus giving the adaptation system itself the ability to make an informative approach to reach a decision.
It turns out, that a meta-policy together with a utility function is fundamentally what is defined by the routine layer.

**Constraints:** The meta policy does not have a pre-emptive fault detection. A chosen policy which turn out to be corrupt will yield the default value instead of the system evaluating a different policy in the policy set. This makes the adaptation system susceptible to human-related policy faults.

**State persisting**

**Advantages:** Persisting the current system state into a template is one of the fundamental features in Agile which allows persisting closed-loop adaptation, as discussed in section 2.4, i.e. a persistent change in the adaptation which yields an effect to the adaptation in the following iterations.

**Constraints:** New templates are stored in separate XML files. It will therefore not automatically inherited by the related policy suite script.

**Decision Points**

**Advantages:** Agile is well adapted for the implementation approach with decision points. The implementation approach has the benefit of allowing a very modular adaptation component which can be easily implemented in legacy projects [34]. Using decision points also support almost any dynamic decision making technology, thus making the approach very flexible. It also allows low run-time resource requirements, and decision latency. The latter depends on the implemented decision making algorithm [35].

**Constraints:** Since adaptation is made locally and triggered when a decision point is reached, there might be cases of where conflict handling is needed. Consider an adaptation made by a decision point. The adaptation is successful, but another decision point is reached before its context information is updated. This can bring the system into an unwanted state if the two decision points are related. Conflict could also occur if several decision points are correlated but independently evaluated without any consideration to their dependencies.

## 5.3 Agile Lite

Agile Lite is a light weight alternative of Agile that is instead designed in the C-language. The motivation for Agile Lite is the possibility to apply the technology to more constrained systems that does not support .net or Java, such as common embedded systems. The library is tailored for the use of decision points for controlling the targeted system using the same method/architecture as the one described in subsection 4.3.2. Agile Lite provides a ready-to-use policy handling interface which maintains, updates and evaluates instantiated decision points.

The policy expression language is compatible with Agile’s expression language as the syntaxes are the same. However, Agile Lite does not support the usage of utility functions within a policy suite.

In terms of library functionalities, Agile Lite has fewer available functionalities. A considerable difference is that the library does not support any state persisting features.

In terms of the defined requirements, Agile lite is a suitable option as it inherently handles some issues regarding.

### 5.3.1 Realtime support in Agile Lite

Agile lite could be used for realizing the policy system since it is designed for embedded systems as it is light weight and C-code compatible. There has also been dedicated tests to evaluate its performance in terms of timings which are found in [35] and [21]. However, there is a lack of support for ensuring realtime support since there is no feature to ensure determinism. Therefore, Agile lite is not fully adapted for realtime systems. But since there is an overall lack of other frameworks for implementing a policy system to embedded systems, Agile lite is still a prominent option.
5.3.2 Conclusion

Using Agile Lite is a promising approach to showcase a policy system. Although Agile Lite is closed source and might not hold all necessary functionalities, it should suffice for a prototype.
Kapitel 6

Requirements for policy engine

6.1 Overview

Several requirements can be defined based on the findings and conclusions from this thesis. These requirements are found in table 6.1. Test descriptions for each requirement are found in table 6.2.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWREQ1</td>
<td>Realtime support</td>
</tr>
<tr>
<td>SWREQ2</td>
<td>Policy repository must be easily accessible to the user</td>
</tr>
<tr>
<td>SWREQ3</td>
<td>Policy system must not fail due to faulty policies</td>
</tr>
<tr>
<td>SWREQ4</td>
<td>System must not fail due to failing policy engine</td>
</tr>
<tr>
<td>SWREQ5</td>
<td>Avoid conflicting adaptations</td>
</tr>
<tr>
<td>SWREQ6</td>
<td>Runtime changeable policies</td>
</tr>
<tr>
<td>SWREQ7</td>
<td>Policy engine should be able to handle multiple adaptation points</td>
</tr>
<tr>
<td>SWREQ8</td>
<td>Policy engine should be able to monitor multiple contexts</td>
</tr>
<tr>
<td>SWREQ9</td>
<td>Policy feature must not severely delay software component’s normal execution</td>
</tr>
</tbody>
</table>

Tabell 6.1: Table of requirements

<table>
<thead>
<tr>
<th>Tag</th>
<th>Test Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWREQ1</td>
<td>The policy engine should have adopted means to gain real-time behaviour</td>
</tr>
<tr>
<td>SWREQ2</td>
<td>Policy repository is separated from the code binary and accessible from a database service</td>
</tr>
<tr>
<td>SWREQ3</td>
<td>Any faults in policy evaluation still yields a valid response</td>
</tr>
<tr>
<td>SWREQ4</td>
<td>A valid output is always given regardless if the policy system fails to give a response</td>
</tr>
<tr>
<td>SWREQ5</td>
<td>Ensure that overlapping adaptations are not executed. A decision on adaptation must be made</td>
</tr>
<tr>
<td>SWREQ6</td>
<td>If the policy description changes in runtime, then the software system’s behaviour changes accordingly</td>
</tr>
<tr>
<td>SWREQ7</td>
<td>All implemented adaptation points in the SW component are able to be evaluated</td>
</tr>
<tr>
<td>SWREQ8</td>
<td>Policies are provided with all contexts that it needs</td>
</tr>
<tr>
<td>SWREQ9</td>
<td>A policy evaluation result should be possible to retrieve with minimal delay time</td>
</tr>
</tbody>
</table>

Tabell 6.2: Table of test descriptions for the requirements
Kapitel 7

Specification for Design

7.1 Introduction

Many aspects and challenges of policy-based design has been introduced and discussed in the previous chapters. A design for a policy system is suggested in this chapter based on the research findings and yielded requirements. This chapter can be seen as a conclusion in the form of a design model based on earlier discussions and analyses.

7.2 A suggestion for design

The suggested design can be regarded as a hybrid between the context triggered adaptation model and the decision point triggered adaptation model. The proposed design is illustrated in Figure 7.1.

The concept is to run a Agile Lite based policy system (engine) as a scheduled task parallel to the target component. Agile Lite is chosen to be used since it will reduce the development time and also handle some of the requirements. By running the policy system as a parallel process, i.e. a policy engine process, the executing policy system is abstracted away from the targeted software component. During its execution it will read contexts (SWREQ8), and re-evaluate (SWREQ7) policy suites whose contexts has changed. Decision points must therefore subscribe to defined context’s. If a new decision from a policy suite has been decided, the decision will be written to a buffer which the target component’s decision points may subscribe to (SWREQ9).

Policy suites will be governed by Agile Lite, meaning that the policies will be defined as policy suites (SWREQ5) and XML-based in separate files (SWREQ2). The policy system will also look for modified
policy suites during execution (SWREQ6). A lightweight server solution such as SQLite could be implemented to manage the XML-files. Since Agile Lite supports defining default values and constrained outputs, faulty policies will not cause critical faults to the policy system itself (SWREQ3). Policy suites also has an inherited property of conflict avoidance as long as each decision point is only tied to one policy suite.

When the target component is executing, decision points will be triggered as it reaches those points. Once a decision point is reached, it will retrieve the subscribed information from the decision buffer. This means that the policy system’s influence on the target component’s execution will be kept deterministic and at a minimum. The execution time required by the targeted software component to retrieve a decision is static, meaning that a change of decision making algorithm computed by the policy engine will not directly impact the software component. This isolates the policy system’s impact on the software components real-time characteristics (SWREQ9). The software component should also be able to validate the output from the policy engine (SWREQ4).

The policy system may also read from a buffer with context information that the target system writes. This enables active monitoring of system contexts with probes.

The four challenges in terms of this design are discussed below.

**Resource sensing**

Resource sensing will be required to some extent. Most importantly is to have a close threshold fluctuation solution, e.g. dead spaces for monitored contexts that tend to have a high frequency of change, such as CPU utilization. Spike handling using some form of estimation process is mainly necessary if any analogous contexts are being monitored, such as temperature. Both functionalities could be handled during the resource monitoring that is executed by the policy engine. However, this could also be done in terms of policy rules.

**Real-time performance**

The design has soft real-time performance by utilizing prioritized scheduling (SWREQ1). In this design, each decision point should be associated with a priority value.

With this model, decisions are continuously propagated to a mailbox which the SW component then reads. The time to retrieve a decision is deterministic and same for all decision points. The only element which affects the decision making variably is the time between the policy engine’s checking of a decision, and the SW component’s retrieval of the decision. This makes the frequency of updating a decision the impacting factor.

Since the frequency of updating a decision is based on the periodicity of evaluating a decision point, periodicity is a vital factor in terms of real-time performance. Scheduling could suggestively be made as rate monotonic, deadline monotonic, earlier-deadline, or least-slacktime.

**Conflict handling**

Agile lite should be used for the functionality of policy suites. Policy scripts could be extended with priorities and suite relations to increase the support for conflict handling.

**Validation and Verification**

Functionality to restrain the decision points outcomes and setting default values makes it easier to validate the system beforehand. This is a step towards validation and verification.
7.2.1 Discussion

The proposed design has considered each challenge on some level. Resource sensing challenges may be solved through using some prediction algorithm if needed. Soft real-time is achieved by prioritizing the evaluation frequency per decision point priority level. Conflict handling can be done by grouping policies in the repository (Policy Suites) and correlation comparisons in the policy system. Validation of the policy system is inherited by constraint factors.

The design has also taken into account the earlier defined requirements. The system supports adaptive behaviour through a combination between decision points and context change triggered policy evaluation. The buffer mechanism allows an asynchronous decision propagation which should enable minimal delays as decision points are reached. The system can be seen as an open-loop adaptation process which can be used for both parametric and compositional adaptation. A next step could be to make the system more closed-loop enabled by implementing means to persist states.

7.3 Use case for implementation

7.3.1 Policy-based decision making for MemSched

The MemSched kernel module is a two-layer hierarchical task scheduler designed for multicore system, see [Figure 7.2]. The scheduling is divided in two layers, a global scheduler, and several server schedulers. The global scheduler is dedicated to scheduling the servers, i.e. when a server may run to execute its task. The task of the server schedulers is to schedule sub-tasks. These sub-tasks are analogous to tasks in classical scheduling meaning. Each scheduler is a fixed priority scheduler (FPS), meaning that all priorities are statically defined upon task creation. Both layers also consider CPU-time and memory accesses as restraining resources for every task. A task is interrupted either once when it has depleted its CPU-time resource, its memory access resource, or passes its deadline.

Policies could be considered to be inserted in two places, namely in the global scheduler, or in the server schedulers. A valid approach would be to implement decision points at points in key places of the scheduler. These places could be implemented where the scheduler e.g. allocates new resources to its tasks. In MemSched, this is done every time a task is either interrupted or finished.

Some thinkable areas of where policies could be introduced are: dynamic resource allocation, manually redefinition of task priorities, periodicity, memory resources, and CPU-time resources.

7.3.2 Policy-based decision making for FaceRec (Face detection)

FaceRec is a face detection c-implementation made by our master thesis group. The FaceRec face detection implementation is based on a Ada-boost algorithm which processes a image frame by iteratively searching for feature-matches over the image with several scaling factors.

Policies could be considered to be inserted in this application to control certain detection parameters. Some examples of these are: scale stepping (step size per image scaling), image search stepping (step size per face detection area in image), and number of weak classifiers (how many classifiers are used to determine the likelihood of a face).

These parameters are strongly tied to required computation power where there is a trade off between accuracy and execution time. Since the face detection application is meant to be run on a live camera stream, execution time can be directly translated to frames per second, which strongly impacts the usability of the application itself. The application is highly parallelizable. Therefore, execution time is also strongly connected with the level of parallelization.

Policies could therefore be considered to be used to change the application’s parameters to adjust the ratio of accuracy versus execution time depending on e.g./ the level of parallelization (how many cores the application runs on).
7.3.3 Choosing the approach

Both approaches from section 7.3 are viable and relevant to this project. One notable difference in terms of this project, is the fact that MemSched runs in kernel space, while FaceRec runs in user space. Because Agile Lite is a built on user space libraries, the policy engine is also suitable to be run in user space. To control a kernel space module from user space, there must exist means of communication between the spaces, which in itself must be investigated. Because of this difficulty, this thesis will involve an implementation for the FaceRec application.

The policy system was implemented accordingly to chapter 7. Rate monotonic scheduling was implemented as a scheduling strategy. The targeted SW component was be the FaceRec application.
Kapitel 8

Implementation of the policy system

8.1 Overview

The implementation is based on the suggested model from chapter 7. The implementation source code is made in C, and provides an interfacing API that can be used by the software component. Agile Lite is used for parsing policy script files and storing the information to C-structures.

Figure 8.1 illustrates an overview of the implementation for this thesis work. It lists the functionalities that are required by the policy engine and the API.

The policy engine is the process that executes the policy related computations. Its functionalities includes: managing of subscribed decision points, managing of relevant contexts, prioritization of decision points, checking for modifications in related policy scripts, and evaluation of policy scripts.

These operations are based on subscription information given by SW components. E.g. if a SW component subscribes a decision point which should be evaluated with policy file X, and contexts C1 and C2, then the policy engine will retrieve contexts C1 and C2, and then evaluate the policy script together with those context values. The evaluation will result in a decision which will be sent to the decision point’s response buffer mailbox. As the SW component reads from its response buffer mailbox, it will find the latest given decision to that decision point. The policy engine will continuously iterate over the subscribed decision points and update decisions as contexts change.
The policy engine supports both active and passive resource monitoring (see subsection 2.1.1). Active resource monitoring is supported via the API, while passive resource sensing is dependent on the Policy Engine. A decision point is able subscribe to multiple contexts consisting of both context types.

The policy engine will continuously check for new subscription requests by reading the first element in the subscription buffer which is designed as a fifo queue.

### 8.2 Policy Engine Implementation

The policy engine execution flow is illustrated in [Figure 8.2](#). For each outer loop, the engine begins with reading all available subscription requests in the subscription mailbox which is a shared memory file in `/dev/shm`.

A subscription request contains information that specifies the XML-script path, default response, valid outputs, relevant contexts, and mailbox ID’s for response. Each decision point is registered to a decision point list containing all subscribed points. This allows support for having multiple decision points concurrently.

Each registration is read and a decision point instance is created based on the provided information from the SW component. All decision point instances are stored in a list, where each decision point is linked with specified context objects. The context objects are handled separately to avoid unnecessary computations.

As a precaution to ensure compatibility between decision point and script file, the script file must match the registered specifications. A mismatch between script file and registered information results in the default decision being sent to the decision point. A mismatch is detected if e.g. the script requests non-existing contexts, or the resulting decision is not a valid output, or if the specific policy script does not exist.

All registered contexts are then updated to their new values. This includes both passive contexts (e.g. from `/proc/`), and active contexts/probes (from SW components). If the decision point’s related script file has been changed, it will be reloaded. The decision point will then be evaluated with the new context values, and a decision will be propagated to the response mailbox of the specific decision point. Each mailbox is realized as a shared memory buffer stored in `/dev/shm`. There may exist 1 response mailbox and an arbitrary number of context mailboxes per decision point.

The policy engine also supports prioritization for real-time performance. Since all decision points are registered in the policy engine, scheduling is handled here. Every decision point is evaluated based on a periodicity using rate-monotonic scheduling. This means that a decision point with a short periodicity has a higher priority than a decision point which has a longer periodicity. A simple example of a rate
monotonic scheduling scheme is presented in Figure 8.3. The example has three tasks which have periodicities of 2, 4 and 6. Therefore, the hyperperiod for the scheme becomes 12. In this case, each task represents the evaluation of a decision point.

8.3 API for Policy Engine

The API for interfacing with the policy engine consists of three main functions, see Figure 8.4. To register a decision point to the policy engine, the SW component must send a subscription request with all required information about the decision point. The SW component may also send context values about itself which the policy engine will read if it is specified in the subscription information. The policy engine will send a decision to the response buffer specified in the subscription information which may be then be read by the SW component. Decision points are implemented as simple memory reads, which reads from the mailbox at the time that the SW component needs the relevant information.

Subscription information for each decision point in the SW component should be sent to the policy engine in the initialization phase of the SW component. The decision point may then be placed anywhere in the SW component code.
8.3.1 API headers

The API for the policy engine may be found in file Policy_engine_API.h. This subsection will also list
the three main functions.

```c
int register_DP_to_PE(struct DPsubscription *aDPsubscription);
```
A DPsubscription structure is sent to the subscription buffer which the policy engine reads from.

```c
int send_context(char *name, int value);
```
Sends an active context which may be used for sending context information from the SW compo-

```c
struct ResponseInfo read_decision(char *name);
```
Reads from the specified shared memory file and returns a response structure containing response infor-


8.4 Use case implementation

A real-time face recognition system was used to demonstrate the usage of the policy system implement-

The focus for the implementation was the face detection part which was fitted with the team’s respective
technologies. To detect faces, a trainer was implemented which takes a number of images on faces and
builds a strong classifier from them. The classifier can be seen as a description of a likely face. This
classifier is then used in the face detection process. The face detection process first reads from an image
stream from a webcam. The streamer sends images via shared memory message passing to the face
detection process. The face detection process takes the image and tries to detect a face by making sub-
window iterations on the image and applies the classifier on each sub-window. If a face exists in the
sub-window, the classifier will yield a high score. This is done on several scaling-factors to make the
detection more robust to the size of the face.

A decision point was implemented in a face recognition system (FaceRec). The decision point was designed
to set a certain iteration parameter based on a user-defined context value which represents the execution
time of the last frame processing. The iteration parameter impacts the accuracy for lower values, but
then also increases execution time. By running the SW component parallelized, execution time can be
lowered which allows a higher accuracy without lowering the frames per second that is processed. The
policy can be used to change the code behaviour to better suite the current parallelization settings (i.e.
how many cores it runs on). For this implementation, the decision points determines the stepping between
each sub-window.

8.5 Results from the implementation

A policy engine was successfully implemented and tested. However, it turned out that Agile Lite does not
support reloading of policy scripts. Therefore the currently solution is not able to reload policy scripts
that has been changed.

The implemented policy engine was successful apart from the above mentions. This is further discussed
in Subsection 8.5.2.

An interesting aspect of this technology is to study the impact in execution time for the SW component
as a result of using decision points. A test program was written which acted as a SW component. The
test program had two decision points, and would repeatedly call for new decisions while sending own
context information to the policy engine. Since the policy engine runs as an independent process and
the only link to the SW component is via the shared mailbox buffers, the interesting timing is the time
it takes for the SW component to retrieve a decision from the mailbox.
8.5.1 Execution times

Results from timing the tests are found in Table 8.1. The execution time for retrieving a decision from a mailbox was measured to an average of 49 $\mu$s. This timing can be compared to executing a normal program such as the face detection application, which analyses a video frame to detect a face which takes approximately 0.6 s on single core. All timings includes the timing calculations and print statements which are involved in extracting the time. The overhead caused by the decision point for this implementation becomes 0.008 percent. For the current implementation, the decision point is read once per outer loop which is the scope for this execution time. Therefore the overhead of the policy system for this implementation is equal to the execution time of a decision read. A decision point’s overhead scales with the number of occurrences of it during the execution time.

<table>
<thead>
<tr>
<th>Test</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read decision point</td>
<td>49 $\mu$s</td>
</tr>
<tr>
<td>Executing FaceRec classifier on single core without decision point</td>
<td>612000 $\mu$s</td>
</tr>
<tr>
<td>Executing FaceRec classifier on single core with decision point</td>
<td>612049 $\mu$s</td>
</tr>
</tbody>
</table>

Tabell 8.1: Table of requirements

8.5.2 Validation of the policy system

To validate the design and implementation of the policy system, each earlier defined requirement in section 6.1 is discussed in relation to the implementation, and results.

- SWREQ1 is met since the designed and implemented policy engine is real-time aware, and uses rate monotonic scheduling. Soft real-time was achieved.

- SWREQ2 is partially met. The policy scripts are separated from the code binary and can be put into a database service designwise. However, a database service was not implemented.

- SWREQ3 is met to the extent of Agile Lite’s capacity. Agile Lite includes validation of policy script files.

- SWREQ4 is met since implemented decision points are set with default values, and the policy engine runs as a separate process from the SW component.

- SWREQ5 is met to the extent of Agile Lite’s capacity. Agile Lite includes a functionality with policy suites.

- SWREQ6 is designwise met, however, not met in the implementation. Agile Lite was found to be incapable of reloading decision points with new policies.

- SWREQ7 is met since multiple decision points can be subscribed to the policy engine which processes them all.

- SWREQ8 is met since the policy engine is able to monitor and store information about multiple contexts.

- SWREQ9 is met since the suggested and implemented design minimizes the overhead on the SW components normal execution. Overhead is equal to a memory read.

---

1Tests were ran on a computer with an Intel pentium 4 HT, 3GHz, CPU
Kapitel 9

Summary, Conclusions and Future Work

9.1 Summary

A study, discussion, and analysis on policy-based design for real-time embedded systems has been covered in this report together with a prototype implementation. Four fundamental challenges were identified in terms of the scope of this thesis: conflict handling, real-time support, resource sensing, and verification and validation.

These challenges together with the analysis of related researches resulted in a suggested design that was then successfully implemented as a prototype. The prototype utilized rate monotonic scheduling for soft real-time performance. The prototype implementation is a step towards the concluded design model.

A face recognition system was used as a use case to validate the functionality of the prototype implementation. The use case implementation showcased that the policy system is able to provide the defined functionalities and meet most requirements.

9.2 Conclusions

From the studies and analyses in this thesis, several conclusions can be drawn. These conclusions are presented in this section.

Realtime performance As earlier mentioned, there is little research in the correlation between real-time performance and policy-based computing. This design and implementation has shown that real-time performance is in fact applicable. This thesis has contributed with a soft realtime implementation for scheduling the re-evaluation process of registered decision points. The rate monotonic scheme allows prioritization of decision points based on periodicity. This gives the policy engine awareness of the frequency priority of updating a decision. A fast changing decision point should be given high priority, while a slow changing decision point should be given a low priority. A slow changing decision point could for example be a decision point which monitors the weather changes in a region, while a fast changing decision point could be one that is incorporated in a realtime control loop.

Overhead The suggested design utilizes buffer mailboxes which enables decisions to be retrieved with minimal overhead. The overhead is equal to the execution time of a normal memory read. In this way, the SW component does not have to wait for a policy evaluation to be made when calling a decision. Tests showed that the decision retrieval timing is significantly smaller than the execution time of a normal application. This makes the suggested design applicable to embedded systems, as the overhead losses are negligible.
Design  The suggested design was implemented. It was shown that the design has a high portability and is able to handle multiple decision points and contexts. The decision points allows the policy system to be easily incorporated in any legacy code. The design also allows the policy engine to control policy based features over multiple processes. The suggested design can therefore be concluded to be an effective approach for policy-based computing.

9.3 Future Work

For future work, the following should be considered:

- Investigate means to communicate between kernel and user space in order to enable policy based services to kernel components, such as a Linux scheduler.

- Implement a custom library similar to Agile Lite, but with added support for runtime changeability.

- Since decision retrieval is a matter of a memory read, one could investigate the possibility for better message passing, such as shared memory message passing.

- Investigate the possibility to incorporate machine learning in order to achieve better closed-loop functionality.
Litteraturförteckning


Appendix

A. XML Expression of Policies in Agile

Any policy object, i.e. policy, rule, action, utility function, and ToleranceRangeCheck may call most policy elements, i.e. external variables, internal variables, templates, return values, actions, rules, ToleranceRangeChecks, and policies.

Environment Variables

<EnvironmentVariables>
  <EVariable Name="SampleEnvironmentVariable1" Type="long"/>
  <EVariable Name="SampleEnvironmentVariable2" Type="bool"/>
  <EVariable Name="SampleEnvironmentVariable3" Type="PolicyObject"/>
</EnvironmentVariables>

Internal Variables

<InternalVariables>
  <IVariable Name="SampleInternalVariable1" Type="long"/>
  <IVariable Name="SampleInternalVariable2" Type="bool"/>
  <IVariable Name="SampleInternalVariable3" Type="PolicyObject"/>
</InternalVariables>

Template

<Template Name="SampleTemplate">
  <Assign Variable="SampleInternalVariable1" Value="7"/>
  <Assign Variable="SampleInternalVariable2" Value="true"/>
  <Assign Variable="SampleInternalVariable3" Value="SampleRule"/>
</Template>

Decision output set (Return value)

<ReturnValues>
  <ReturnValue Name="RetAlert" Value="-1"/>
  <ReturnValue Name="Ret1" Value="1"/>
  <ReturnValue Name="Ret2" Value="2"/>
</ReturnValues>

Action

<Action Name="SampleAction">
  <Assign LHS="SampleInternalVariable1" RHS="8"/>
  <Assign LHS="SampleInternalVariable2" RHS="false"/>
  <Assign LHS="SampleInternalVariable3" RHS="SampleRule"/>
</Action>
<EvaluateTRC TRC="SampleTRC"/>
<EvaluateRule Rule="SampleRule"/>
<EvaluateUF UF="SampleUF"/>
<Yield Policy="SamplePolicy"/>
<Return ReturnValue="Ret2"/></Action>

**Rule**  Null will trigger the consecutive event.

```xml
<Rules>
  <Rule Name="SampleRule" LHS="SampleEnvironmentVariable2" Op="EQ" RHS="true"
    ActionIfTrue="SampleAction" ElseAction="null"/>
</Rules>
```

**ToleranceRangeChecks**  Example will check a 10% tolerance zone.

```xml
<ToleranceRangeChecks>
  <TRC Name="SampleTRC" Check="SampleEnvironmentVariable1"
    Compare="SampleInternalVariable1" Tolerance="10" ActionInZone="null"
    ActionLower="Ret1" ActionHigher="SampleAction"/>
</ToleranceRangeChecks>
```

**Utility Function**

```xml
<UF Name="SampleUF" Terms="2">
  <Option Action="SampleRule" T1="SampleInternalVariable1" W1="5" T2="7" W2="10"/>
  <Option Action="SampleTRC" T1="SampleEnvironmentVariable1" W1="5" T2="12" W2="10"/>
</UF>
```

**Normal Policy**

```xml
<Policy Name="SamplePolicy1" PolicyType="NormalPolicy">
  <Load Template="SampleTemplate"/>
  <Execute Action="SampleAction"/>
</Policy>
```

**Meta Policy**

```xml
<Policy Name="SamplePolicy1" PolicyType="MetaPolicy">
  <Load Template="SampleTemplate"/>
  <Execute Action="SampleUF"/>
</Policy>
```