Characterising the Ripple Effects of Introducing Energy-Awareness Functionality in Cyber-Physical-Systems Software

MASTER THESIS

A. (Atze) Bouius
CSC-MS-SE - Software Engineering

January, 2015

EEMCS University of Twente
P.O.Box 217
7500 AE Enschede
The Netherlands

Abstract

This research investigates what impact energy-awareness functionality has on an existing software design. Energy-awareness has become increasingly important in software development; especially for Cyber-Physical Systems. Nowadays society is more conscious about energy consumption; optimising energy usage has become an important selling point. Self-adaptation of system behaviour can heavily influence energy usage; especially for battery powered systems. Information about power consumption can be used to plan ahead; the system can try to react and adapt before it runs out of power. A lot of existing software systems have no notion of energy; the introduction of energy to an existing system may result in side effects. These effects or ripples are modifications that are caused when energy-awareness functionality is added to the existing system. Energy-awareness functionality has a crosscutting nature; it is present throughout the system from hardware driver to high level planning and adaptation components. This may result in an implementation where this functionality is scattered and tangled with multiple components; every component may contain statements concerning energy consumption.

To investigate the effects of introducing energy-awareness functionality to an existing design, we conduct a case study. Initially we designed a base design which is energy unaware. However, best efforts are made to prepare the design for future evolution. The base design is split into two parts: the base system and the control system. The control system adapts the behaviour of the base system. The initial design was composed of various design patterns. The Observer pattern and Visitor pattern served as interfaces for the control system to the base system. The control system follows the MAPE-K terminology which is commonly used to describe control loops. The control system contains a state space (State pattern) to determine the most suitable adaptation. A domain analysis is conducted to extract energy evolution scenarios. The energy-awareness functionality introduced by these scenarios is added to the base design. We observed the design ripples that occurred after implementation of the functionality by various techniques; an object oriented (OO) and an aspect oriented (AO) design are evaluated. Both designs are evaluated using various basic metrics (lines of code (LOC), number of components and operations). To observe the diffusion of a concern over a design we use concern diffusion metrics. A scenario-based analysis technique SAAM is applied to show evolution impact for each separate evolution scenario.

The OO implementation suffered from the introduction of new events and states which are needed to apply newly introduced adaptations. A lot of evolution ripples appeared in the control system due to interface changes caused by the introduction of new observable and visitable components. The measurements revealed that the AO implementation introduced energy-awareness functionality mainly through new components instead of changing existing components. AO allows the dynamic extension and implementation of existing interfaces through aspects which solve the problems related to the introduction of new events and adaptations. The metrics confirm the assumption that energy-awareness functionality has a crosscutting nature. Although best effort is made to modularise the initial design, OO and AO programming cannot prevent the scattering and tangling of energy-awareness functionality throughout the designs. New introduced events and states lead to state space evolution within the control system. State space evolution cannot be achieved without rewriting the context class (OO) or context aspect (AO).

We investigated a number of established and experimental event-based approaches to determine if our design can benefit from them. EventReactor and its event module model look promising in this regard. The event model of EventReactor provides loosely coupled event modules which are unaware of each other. The modules are linked together by events, they can react to input events and can fire output events to notify other interested event modules. Future research should address language level support as well as scaleability. Case studies on other types of CPS should be conducted to be able to generalise the findings of this research.

Contents

1 Introduction

Energy-awareness has become increasingly important in software development; especially for Cyber-Physical Systems (CPS). Society has become more conscious about energy consumption and optimisation of energy usage has become an important selling point. Self-adaptation of system behaviour can heavily influence energy usage; especially for battery powered systems. Battery-powered systems deal with a limited power supply and need to save resources to be able to complete their tasks. The information about power consumption can be used to plan ahead, the system should try to react and adapt before it runs out of power. Running out of power in the middle of a task can be undesired; especially if the task cannot be resumed if the system restarts. A lot of existing software systems have no notion of energy. The introduction of energy-awareness functionality to an existing system may result in side effects. These effects or ripples are modifications that are needed to add the energy-awareness functionality to the existing system. This may reveal flaws in the design or break down the design entirely. Therefore, modularisation of this functionality will be a difficult task; energy-related code fragments might be scattered throughout the entire code. Energy-awareness functionality has a crosscutting nature; it is present throughout the system from hardware driver to high-level planning and adaptation components. This may result in an implementation where this functionality is scattered and tangled with multiple components; every component may have statements concerning energy consumption.

1.1 Research Questions

Software evolution is a challenging topic, designers must take into consideration what will happen if a new concern is introduced. The introduction of an unexpected concern may break down the design completely. The question will rise if this is the fault of the design or if it could have been prevented by taking different design choices. We will investigate the introduction of energy-awareness functionality to Cyber-Physical Systems. The following research questions are formulated:

- What are the effects on software modularity that follow from the introduction of the energy concern to Cyber-Physical Systems?
- Can the energy-awareness concern be introduced to an existing Cyber-Physical System without crosscutting existing concerns?

Answers to these questions should give insight in the effects that follow with the introduction of energy-awareness functionality to an existing software system.

1.2 Related Work

This research focusses on the design and evolution of CPS and the impact of energy-awareness functionality on the design. There are many topics related to CPS development. For this reason we will focus explicitly on research related to CPS design methods and challenges, and on energy-aware software development regarding CPS.

De Lemos et al. (2013) [1] propose a roadmap for developing, deploying, managing and evolution of self-adaptive software systems. Specifically, they focus on development methods, techniques, and tools that they believe are required when dealing with software-intensive systems that are self-adaptive in their nature. The research identifies four major topics crucial for engineering self-adaptive software systems: design space, software engineering processes, centralised to decentralised control, and practical run-time verification and validation (V&V). Especially the decentralisation of control loops which form the control system is interesting for this research. They use the MAPE-K [2, 3] terminology and present the following patterns: Hierarchical Control, Master/Slave, Regional Planner, Fully Decentralised and Information Sharing.

The following research challenges are formulated: the identification of what circumstances that decide the applicability of patterns, as well as which application domains or architectural styles are better managed by patterns. Another major challenge is to identify techniques that can be used for guaranteeing system-wide quality goals, and the coordination schemes that enable guaranteeing these qualities. The focus of their research is on the control system itself; instead of our research which focus is on the coupling between the base system and control system. Their research is presented as a roadmap for CPS development; instead of our research in which we conduct a case study to find out if our assumptions of energy-awareness being a crosscutting concern is valid.

Te Brinke et al. (2013) [4] present a method for developing energy-aware software. In green software development the reduction of overall energy consumption of the software is an important issue. This can be achieved by energy optimisers; software can be made energy-aware by extending its functionality with energy optimisers. Energy optimisers monitor the energy consumption of software and adapt it accordingly. Energy is considered as a special kind of resource; a component may consume various resources, which eventually may lead to the consumption of energy.

A method is proposed to design energy-aware software systems in such a way that modularity is achieved in the design of such systems. Key steps in this method are the identification of components, modelling of ports (interfaces), the modelling of resource behaviour, analysis and finally the selection of most suitable optimiser component. They illustrate the usefulness of their technique in a separate technical report where they apply the method to a media player case study.

The main focus of this research is on model checking where our study presents a complete implementation for a robotic system. Although the media player is also some kind of CPS it does have access to a fixed power supply; the robot instead deals with a limited power supply and needs more complex energy-awareness adaptation scenarios to ensure maximal operation time and performance.

Next to the two papers discussed above which focus on development techniques, more research has been done, of which three papers are relevant here. Salvaneschi, Ghezzi and Pradella (2013) [5] perform a language level support analysis for self-adaptive software systems. They state: "Self-adaptive software has become increasingly important to address the new challenges of complex computing systems. To achieve adaptation, software must be designed and implemented by following suitable criteria, methods, and strategies."

Their work focuses on finer-grained programming language-level solutions for CPS development. Three main linguistic approaches are analysed: meta-programming, aspect-oriented programming (AOP) and context-oriented programming (COP). Each approach is analysed and compared on the following topics: application, behavioural change, monitoring and finally variations and separation. The following research challenges were formulated: modularisation, extensibility, performance impact, adaption to unforeseen situations and impact on the development process.

The paradigms discussed in this article solve the problem of extending existing (mainstream) languages with the flexibility required by self-adaptive systems. AOP, COP, and meta-programming introduce new directions of variability to model behavioural adaptation; they support interception to inject monitoring code, and the dynamic modification of normal execution. They conclude that it is hard to provide conclusive arguments for the benefits they achieve through the use of specific linguistic support. Empirical observation should be conducted on application development to be able to provide conclusive arguments; our research may help to provide these arguments. Our research illustrates that CPS development can profit from applying AO programming. The AO implementation of our design reduced the impact of energy-awareness evolution scenarios to the existing design drastically. Existing components can be extended by aspects instead of modified.

McKinley, Sadjadi, Kasten and Cheng (2004) [6] conduct a survey about approaches that have been proposed for building software that can dynamically adapt to its environment (self-adaptive systems). Adaptations do not only involve changes in program flow, but also run-time recomposition of the software itself. When designing self-adapting systems three key technologies should be considered: separation of concerns, computational reflection, and component-based design. Together, they provide programmers with the tools to construct self-adaptive systems in a systematic and fundamental way. In addition they discuss how middleware supports compositional adaptation. Middleware effectively provides a level of indirection and transparency that can be exploited to implement adaptations.

They list a number of research projects and commercial software packages that support some form of compositional adaptation, and conduct a taxonomy that distinguishes approaches by how, when, and where software composition takes place in these projects. They identify the following key challenges: assurance, security, interoperability, and decision making. They conclude that compositional adaptation is very powerful; however, without appropriate tools to automate the generation and verification of code, it may have a negative impact on the integrity and security of systems, instead of improving it. Unlike our research they focus on analysing software composition in existing commercial software packages; our research instead focusses on the application of commercial software packages with respect to energy-awareness evolution.

Andersson, De Lemos, Malek and Weyns (2009) [7] present a paper in which they focus on the lack of consensus among engineers on some of the fundamental underlying concepts of self-adaptive systems. They try to solve this issue by exploring the role of computational reflection in the context of self-adaptive software systems. Reflection is about meta-computation, i.e., computation about computation. They identify several reflection properties and group them in a reflection prism with three sides: Self-Representation, Reflective Computation and Separation of Concerns.

Any computational system has a domain model, which corresponds to the type of the application domain (business problem) addressed by the system. In a reflective system, there is a distinction between the domain model and the self-representation. Self-representation is a key characteristic of any reflective software system. Self-representation has four key properties namely: type of representation, granularity, uniformity and completeness.

The reasons for using reflection in a system vary; the behavioural properties of a reflective system affect several system properties. Together these properties make up the reflective computation side of the prism. Properties here are: type of reflection, causality, level-shifts and frequency of these shifts. In the context of reflective computation, separation of concerns is vital as the reflective behaviour increases the systems overall complexity.

If the reflective system is able to support separate models of different system aspects, possibly at different reflective levels, the overall complexity can be reduced. Properties here are: disciplined split, transparency, hierarchy and extensibility. They apply the properties of the reflective prism to different case studies and identify key challenges when building self-adaptive systems. These challenges lie in expressiveness of self-representation, meta-level conflicts, uncertainty, autonomy and transparency. In this research we apply the self-representation side of the reflection prism by conducting a domain analysis, from which we extracted an environment model for the system. Our system needs the environment model to be able to navigate and move around.

1.3 Proposed Solution

In order to investigate the effects of introducing energy-awareness functionality to an existing software design we will follow this roadmap:

- Define a initial energy-unaware robot software design.
- Introduce energy-awareness functionality through evolution scenarios.
- Implement energy-awareness functionality in established techniques (OO and AO programming).
- Experiment by conducting measurements on the OO and AO implementations.
- Evaluate how established and experimental event-based modularisation techniques can help to solve the exposed problems.

Software engineering provides various established techniques to cope with evolution; OO programming provides design pattern to decouple and modularise components. It is assumed that the initial design is not aware of energy; however, best efforts are made to prepare the design for every possible evolution scenario. The patterns that we adopted for implementing energy-awareness functionality via control loops are Observer, Visitor and State pattern. The system can be made self-adaptive by adopting control-loops to implement high-level adaptation mechanisms. Together control loops form the control system; the control system is coupled to the base system by the Observer and Visitor pattern.

The impact of energy-awareness evolution is illustrated by a case study: a security robot patrolling an environment. The robot must adapt its behaviour depending on its position in the environment and security threats. A domain analysis is conducted to extract energy-awareness evolution scenarios. The energy-awareness functionality introduced by the scenarios is added to the existing design adopting OO and AO programming.

The OO model has many shortcomings, crosscutting concerns are hard to implement in the object model. A concern is crosscutting if its implementation is scattered and tangled within many components. The extension of fixed interfaces results in ripples throughout all classes implementing these interfaces. Adding new functionality leads to rewriting or adding new methods to existing classes. Dynamic class extension as well as the dynamic interface declaration mechanisms provided by AO, solve many of these problems. However, state space evolution and especially the introduction of new states cannot be solved by aspects. Context-dependent state information prevents aspects from isolating the state-context concern. Extension of each context is context dependent which makes it impossible to generalise it into a reusable concern.

This research will compare OO and AO implementation using various modularity metrics and scenario-based analysis. Experimental event-based modularisation techniques will be evaluated to see if a higher degree of decoupling and modularity can be achieved. Since some of these techniques are still in development and only partially implemented we cannot conduct measurements on it. However these experimental techniques will give an insight in the (future) possibilities for modularising and decoupling energy-awareness functionality from an existing design.

1.4 Thesis Outline

This research gives insight how the introduction of the energy-awareness concern to an existing CPS affects the modularity of the design. In *chapter 2* some background information about CPS, control systems, energy, software evolution, modularity and metrics is provided.

A case study is conducted to illustrate how an existing design is affected by the introduction of the energy concern. Various techniques are applied and evaluated to minimise evolution impact on the design.
In *chapter 3* we will present the case study scenario and the respective requirements and assumptions who have to be considered for the initial design. Further this chapter gives an overview of the various design alternatives and design choices for the initial design.

The initial system design should be energy-unaware; however, best efforts should be made to prepare the design for possible evolution. In *chapter 4* we will introduce new requirements through energy evolution scenarios. The scenarios are extracted from the conducted domain analysis. OO and AO implementations are presented; both implementations implement the energy evolution scenarios.

In *chapter 5* we will present the evaluation method. Metrics are calculated for both implementations and the results are compared. An overview of the applicability of event-based modularisation techniques is given in *chapter 6*. Esper and EventReactor are evaluated to determine how event-based modularisation can help to solve state space evolution problems.

Finally conclusions and future work are discussed in *chapter 7*.

2 Background

In this chapter we introduce *Cyber-Physical Systems* [8, 9] and provide background information to clarify the topic of CPS design. Typical CPS are closed-loop systems, they adapt behaviour according to feedback provided by sensors. These feedback loops are implemented by means of a *control system*. Some CPS like mobile systems deal with a limited energy supply. *Energy-awareness* functionality should be introduced to these systems to optimise their operation time. Energy management should be a self-adaptive process. *Self-adaptive*[2] software systems are typically implemented by the *MAPE-K*[2, 3] model. The key functions of self-adaptive control systems are: Monitor, Analyse, Plan and Execute over a common Knowledge base. Due to the *crosscutting*[10] nature of energy-awareness functionality, we assume introducing energy-awareness functionality to an existing system may compromise the *modularity* of the design. We assume energy-awareness functionality is introduced by evolution scenarios. To evaluate the modularity of a design as well as the impact of evolution scenarios *metrics* must be identified to be able to conduct measurements.

2.1 Cyber-Physical Systems

Cyber-Physical Systems [8, 9] are all around us, examples are traffic control and water resource management systems. Robotic systems also fall in the category of CPS.

– CPS systems are typically closed loop systems, where sensors make measurements of physical processes, the measurements are processed in the cyber subsystems, which then drive actuators that affect the physical processes –

[Edward A. Lee, UC Berkeley]

CPS form the bridge between the cyber world and the real world. More formally, they communicate with entities in the physical world and integrate computing with monitoring and/or control. They react to stimuli from the outside world and adapt their behaviour accordingly (emergent system behaviour) [8, 9].

Stimuli are triggers/events [8], *events* are occurrences/changes in properties of interest. To observe the real world, CPS make use of sensors. Sensors can observe a small part of the real world. Like distance to a wall, light intensity, battery power etc. Whenever such properties of interest change this is considered an event.

Events provide a natural way for reactive systems to observe behaviour and to specify component interfaces. Events can be used to specify coordination and composition of components. The usefulness of events degrades over time; the longer it takes for the system to react the greater the chance the response action is ineffective (soft-real time). For example, imagine a robot following a wall; the robot corrects his behaviour according to the distance to the wall. If a distance event is handled too late the actual distance is different from the value the robot is trying to adapt to. In this case the adaptation is only partially effective. After a certain amount of time the event becomes completely useless; for this reason events must be communicated in time.

CPS need to react to change continuously; the sequence of observing changes and adapting accordingly is a never ending loop. Usually *feedback loops* ensure *adaptation* to physical processes by the means of the computation of an advice/plan. This advice contains one or multiple adaptations that correct/adapt the system.

Developing CPS is a challenging task [9]; software component technologies like object oriented (OO) design and service-oriented architectures are based on abstraction that are intended for software rather than physical systems. The gap can partially be closed [9] with:

- advancements in formal verification;
- emulation and simulation techniques;
- certification methods;
- software engineering processes;
- design patterns;
- component technologies.

2.2 Energy

The electrical point of view of energy [11] is: energy is absorbed by or generated from an electrical circuit. Energy is expressed by the following formula:

$$E(t) = \int_0^t P(\tau)d(\tau) \tag{1}$$

Energy is power consumed over time, power can be defined as the amount of energy consumed per unit of time. Power is expressed by the following formula:

$$P(t) = V(t) * I(t) \tag{2}$$

Power P at a certain moment in time is the product of the voltage V and current I at that moment in time.

With this physical information we can consider energy from a software point of view. Before one can consider energy in a software system one should have a notion what energy is and how to measure it. Each hardware component may use a different amount of energy. Sensors can be used to measure energy levels; alternatively static values can be introduced based on datasheet information of the hardware components, in order to simulate energy consumption of components. In the last scenario energy consumption is based on assumptions. Either way this information should be added to software modules who directly interact with the hardware. More complicated energy scenarios affect planning and strategy modules. Software evolution is an ongoing research topic; writing and rewriting software is a costly process. A design fit for evolution can save a lot of effort and money.

2.3 System Control

Control systems are interdisciplinary; there exist physical control systems [12] as well as software control systems. Physical control systems, also called feedback or closed-loop systems, are used in mechatronics. Optimising machinery movements is an example of their application. These control systems can be implemented in either hardware or software and are applied in embedded systems. Software control on the other hand is implemented by feedback loops, keywords here are monitoring, analysing, planning and executing adaptations.

2.3.1 Physical Control

Physical control realised by so-called control or closed-loop systems [12] work with a closed loop. The error in the output is fed back to the input in order to determine if additional correction is needed. Control systems provide a way to modify dynamic system behaviour through a feedback mechanism. A typical physical control system is modelled in a block diagram a general model is displayed in figure 1.

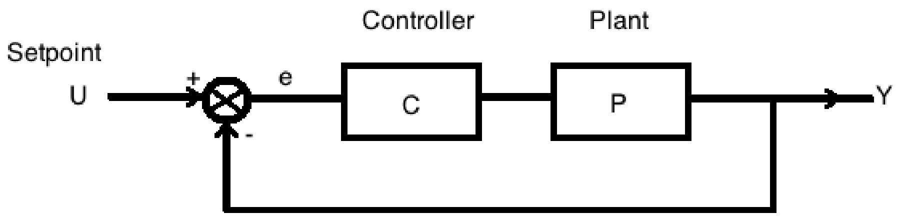


Figure 1: Physical control system

The preferred output value of the system is defined by the *set-point* or *reference value.* One must define a monitor to achieve this, the monitor compares the system output (Y) to the set-point value (U). The output of the monitor is called the error signal, the error signal (e) is the difference between the desired system output and the actual system output. The error signal is fed back as input to the system or plant to correct the output signal in order to decrease the gap between the actual output and desired output signal. The mathematical representation of the relation between the input and output can be expressed by a differential equation. This equation is called the system function or *transfer function* [12](p. 34-38). A control system can be turned into a *closed-loop system* if the output is measured before it is compared to the reference value. A non closed-loop system receives no feedback from the outside world.

These systems can be partitioned into two categories namely *linear* and *non-linear systems* [12](p. 39-52). If a system is modelled as a linear system, the output is proportional to the input. Linear systems are useful for modelling mathematical systems but not for modelling real world systems; most real world systems are non-linear systems. Control systems are applied in a wide range of fields, examples are climate control, neural networks and medical systems.

2.3.2 Software Control

Software control is not limited to controlling a systems output, software control can model many complex feedback mechanisms. Software control systems are often referred to as *self-adaptive systems.* Complex systems with a wide variety of sensors and actuators can be modelled by various overlapping feedback [2] mechanisms. System behaviour optimisation is the main goal of software control; examples are controlling computation or request load for processor and server control systems. Another example would be an energy-management optimisation system.

Self-Adaptive Systems A lot of software systems must adapt to a changing environment. A server system for example, which will get increasingly more requests and therefore needs to deal with an increased workload. By detecting changes in the environment the system can predict what is going to happen and deal with it accordingly. In the example of the server system this could be the increment of the capacity. In this case, adding additional servers to a virtual machine can be a appropriate adaptation. This server system is an example of a self-adaptive system [2]. The adaptation functionality can be implemented as a feedback loop. A typical cycle would be: measure deviations (observe), comparison to a reference value (analyse), planning the best possible action (plan) and the adaptation to a changing environment (act).

Base System and Environment The *base-system* interacts with the outside world called the environment; the environment is the relevant part of the real world. The system interacts with the environment through sensors. Sensors measure certain changing variables in the environment. These changing variables give the base system insight on what is happening in the environment. This information is crucial for self-adaption; without stimuli there is nothing to adapt to.

MAPE-K MAPE-K [2, 3] or Monitor-Analyse-Plan-Execute over a Knowledge base is a common terminology used to describe control loop systems. The principle is illustrated by figure 2; the figure shows the base system and the way it interacts with the autonomic control component. This autonomic control component or control system exists of four sub-components which have a common knowledge base. The interaction with the controlled resource is handled through sensors and effectors.

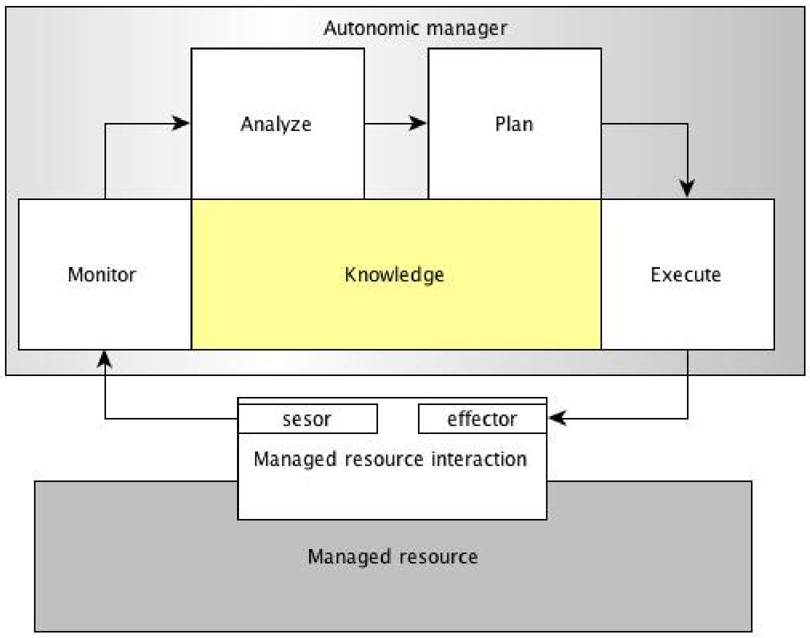


Figure 2: MAPE-K principle

We will briefly introduce the four control components as well as the knowledge base. The knowledge base can be accessed by multiple control loop components; however, it is in particular important to the analyser component. For this reason we will discuss the analyser and knowledge base together.

Monitor The monitor collects relevant information regarding the behaviour/structure of the base system (managed resource). Additionally it may collect information from the environment. Examples are: changing variable values and current state of the system. The base system and environment can be seen as subjects of the monitor. The monitor simply passes the data from the base system to the analyser. Therefore, it acts as an interface for the control loop to the base system. Complex systems that deal with many high rate data flows may have to deal with hardware limitations, e.g. insufficient memory and high CPU loads. In these cases data compression should be applied; average values can be used or some other correlations over a set of samples can be applied.

Analyser and Knowledge Base The data passed by the monitor component must be analysed. The analyser component processes the data and should be able to detect undesired values. First the analyser interprets the data; by giving an interpretation to the data it becomes possible to compare it against a reference model (Knowledge base). Deviations from the reference model indicate the system is not longer functioning as desired, given the apparently changing environment. Therefore the control loop should determine on an adequate action/adaption. The control system should plan how to adapt the base system.

Planner The planner component is closely related to the analyser; once the data is analysed there must be some coordination to adapt the base system. Analysers become more useful if historical information is taken into account. State-charts can be used to keep track of what happened in the near history. When the past is known, it becomes possible to spot trends and predict what can happen in the future. Taking this information into account, adaptations become more effective and the performance of the managed resource can be optimised. The planner takes new analysed data and current state information into account when determining the next state. New states may contain one or more adaptation actions; these adaptation values are send to the respective executors.

Execute Executors apply adaptations determined by the planner component. In practise this will mean the system switches behaviour; examples are optimal performance mode, energy saving mode or threat mode for tasks that require immediate attention. Adaptations are executed on the managed resource by so called effectors; effectors change the behaviour of the managed resource.

2.4 Software Evolution

Already in the early 70s Lehman et al. formulated laws/properties regarding software evolution [13]. These laws give insight on how software evolution shapes the program. The term E-type systems is used for software systems embedded in a real-world domain. Lehman formulated the following laws [13]:

- "Continuing Change" – an E-type system must be continually adapted or it becomes progressively less satisfactory
- "Increasing Complexity" – as an E-type system evolves, its complexity increases unless work is done to maintain or reduce it
- "Self Regulation" – E-type system evolution processes are self-regulating with the distribution of product and process measures close to normal
- "Conservation of Organisational Stability (invariant work rate)" - the average effective global activity rate in an evolving E-type system is invariant over the product's lifetime
- "Conservation of Familiarity" – as an E-type system evolves, all associated with it, developers, sales personnel and users, for example, must maintain mastery of its content and behaviour to achieve satisfactory evolution. Excessive growth diminishes that mastery. Hence the average incremental growth remains invariant as the system evolves.

- "Continuing Growth" – the functional content of an E-type system must be continually increased to maintain user satisfaction over its lifetime
- "Declining Quality" – the quality of an E-type system will appear to be declining unless it is rigorously maintained and adapted to operational environment changes
- "Feedback System" (first stated 1974, formalised as law 1996) – E-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base

These laws formulate the basics of software evolution. Empirical evidence [14] has been conducted to see if these laws still hold for modern day open source projects. There are indications that Continuing Change, Increasing Complexity, Self Regulation, and Continuing Growth are still applicable to the evolution of modern day open-source software projects.

Interesting are *'change hot spots'* [14], this are code fragments where changes concentrate. A small percentage of the total code that contain a high percentage of changes. Interface changes almost never happen after the initial design. Implementation changes are more likely to occur.

2.4.1 Scenarios

Software evolution in general is scenario-driven; but what exactly does this mean, we give the definition of scenario according to Merriam-Webster:

– scenario: "a description of what could possibly happen" –

[Merriam-Webster]

Scenarios are descriptions of possibilities that may occur. The evolution of software is shaped by newly introduced scenarios. Software modules must be rewritten or extended to satisfy a growing set of requirements introduced by evolution scenarios.

A design should be prepared to cope with evolution; evolution scenarios should be taken into account when modelling a software design. Evolution scenarios should be extracted by conducting a *domain analysis*. The domain analysis [15] should map all possible relevant entities and relations in the system domain; this process is called *domain scoping*. In many cases not only the technical (functional) aspect but also the economic aspect should be considered.

The domain analysis is just a small part of a larger process called domain engineering [15]. Systematic software reuse can be achieved by analysing the application domain. Analysing the application domain results in the identification of commonalities. Commonalities form the basis of reuse within a software product line.

Various domain analysis methods exist; FODA [16] and ODM [17] are examples. We will not describe these methods in detail since they are outside of the scope of this research, but we will leave them here for the interested reader.

With the cause for software evolution identified, it is now time to move on to *modularity.* Modularity is the keyword to achieve a highly decoupled design. Evolution scenarios will test the evolvability and modularity of a design, design flaws can be revealed if the design is static and non-modular.

2.4.2 Modularity

Modularity is an important topic in software design; generally, system functions are decomposed into reusable independent modules. A design is modular if modules have well defined interfaces resulting in loosely coupled components fit for reuse. Modules should be designed in such a way that they are loosely coupled to each other and communicate with each other via their interfaces. Modular designs are said to be more evolvable; future functionality can be added with less impact. A loose coupling and well defined interfaces create the possibility to reduce the ripple modification effects, once a module needs to evolve.

An indication of the modularity of a design can be obtained when looking at certain evolution aspects of its components [18]. The survival rate of a component indicates how important it is. It indicates the degree in which the component can be removed or replaced over time. *Maintainability* is another important aspect; maintainability of a component indicates the component's stability. The stability of a component is the degree in which changing requirements affect it. For the system as a whole one should consider component augmentation. Component augmentation indicates how easily new components can be added to an existing design. Together these aspects give insight in how a component will evolve, as well as the likelihood of the component disappearing from the design.

Tight coupling indicates a high dependency of other components; a high dependency means a higher survival rate. Killing such components results in a lot of unwanted effects. Throughout the design, many other components may depend on the killed component; all of these components need to be updated. This implies that these components are less adaptable in terms of substitution or exclusion. The impact of changes in these components may be amplified due to the high amount of dependencies. Which in turn may result in more maintenance effort. Due to their high dependencies augmentation will also be harder for such components.

Core components tend to deal a lot with these problems; *core components* are components that were present in the initial design; destined to be long lasting. These often tightly coupled components are vital to the system; naturally this comes with a higher survival rate. However, it can be expected that core system functionality changes quite often. Each new version of the system strives to improve performance, made possible by new advances in technology. This will make some core components obsolete or an opportunity arises to replace them. However, adapting tight coupled components is a difficult task and may affect many other parts of the design.

One must be aware of tight coupled components and the constraint they place on the design. These components are hard to remove and result in reduction of future flexibility. This results in more maintenance effort due to design changes [18] in future versions.

This problem is magnified by code reuse. *Legacy code* is used as a platform to build new versions on top of. This legacy code will hardly be rewritten; existing problems are magnified when used in the dependent code.

The *core components* for a self-adaptive design are the MAPE-K components which form the control system. The base system can be considered as legacy code; it is deemed undesired to make modifications to the base system made necessary by evolution. The control system components should be designed in a modular way so they can be flexibly coupled to the base system. Evolution capabilities of these components should be considered carefully. The energy-awareness functionality especially may affect these components; energy-awareness functionality most likely will introduce new events and adaptations. Events and adaptations directly relate to the monitor and executor components of the control system. However, not all concern can be modularised, this brings us to the next topic: crosscutting concerns.

2.4.3 Crosscutting Concerns

Concerns of a program that affect other concerns are called *crosscutting concerns* [10]. Trying to separate such a concern in a modular and decoupled way, results in a scattered concern throughout the design and implementation. Dependencies between modules can become polluted when trying to implement such a concern. This pollution is called tangling; in practise this results in modules that heavily depend on each other. There is evidence suggesting [10] that crosscutting concerns cause defects in software. The more a concern is *scattered* the more likely it will cause defects. This effect is evident and independent of the size of the concerns implementation (in terms of LOCs). The separation of such concerns are hard to achieve in an OO design. *Design patterns* [19] are already a big improvement. Design patterns are designed to decrease coupling between components and increase the components modularity. However, even with the help of these patterns OO falls short when implementing crosscutting concerns; this is illustrated by Eaddy, Zimmermann, Sherwood, Garg, Murphy, Nagappan, and Aho [10]. They explain the pros and cons of various design patterns and how they are suitable to modularise crosscutting concerns, based on modularity, uniformity, transparency and reusability. Further they show how these patterns can be improved with AO techniques.

2.4.4 Modularising Energy

A robotic/cyber physical system is a combination of hardware and software. The intelligence is encoded in the software whereas the interaction with the environment is handled by the hardware. Actuators like engines for moving and sensors for measuring changes in the environment are examples of hardware interaction with the environment. The software communicates with the hardware through third party libraries. Most components do not provide energy information since this requires additional sensors; adding energy sensors will increase the production costs. In order to monitor the energy consumption of each individual component, each different hardware component needs its own energy sensors.

The readings of these sensors should be coupled (software) to the hardware component they are measuring. Each hardware-driver component deals with energy; each component must implement energy-awareness functionality. Considering energy is one thing, but to actually use it for a high level scenario is something else. New plans and strategies based on energy need to be implemented. Sensor readings can be stored for optimisation of the system (improved performance based on history). Energy-awareness functionality is not only present in low level hardware drivers but also in the higher level planning components. These are the characteristics of a crosscutting concern. It is impossible to isolate such a concern into a single separate component. The modularity and coupling between components of a program determine if a design is modular and reusable. Crosscutting concerns break down the design and result in a tangled implementation. To be able to evaluate the modularity and evolvability of a system metrics are needed.

2.4.5 Metrics

Separating concerns is achieved by implementing modular and decoupled modules that deal with different concerns. Each module should deal with its own concern and ideally not be aware of other modules. To be able to determine if a good separation of concerns is achieved the coupling between the various concerns present in the system should be determined [20]. Metrics provide a way to evaluate modularity, separation of concerns and scenario impact.

Concern diffusion metrics There exist some metrics to evaluate the separation of concerns; these metrics are called *concern diffusion metrics* [21]. Concern Diffusion over Components (CDC) [21], Concern Diffusion over Operations (CDO) [21] and Concern Diffusion over Lines of Code (CDLOC) [21] indicate the impact of a concern in a program. The number of classes (aspects in AO) that contribute to the implementation of a concern are indicated by CDC. CDO gives an indication of the number of methods (advices in AO) that contribute to the implementation of a concern. CDC and CDO together give insight in the degree the concern is scattered at various granularity levels. CDLOC indicates the number of transition points of each concern in terms of lines of code. Code must be partitioned in two parts; code that implements a given concern and code that does not, transitions in both ways are counted. A high value indicates a high mingling of concern code within the implementation of the components, whereas a low value indicates that the concern is localised in the concern code. CDLOC indicates the degree in which a concern is tangled.

Scenario-based Analysis Since the new concern is introduced through various evolution scenarios, another evaluation technique called *SAAM* [22] might be more suitable. SAAM (Software Architecture Analysis Method) is a scenario based analysis technique. SAAM provides us with various minor metrics to validate the impact of specific evolution scenarios. This way we can isolate the impact of each scenario related to a concern, instead of the degree to which the concern is scattered or tangled. Metrics to consider are new, changed or deleted classes, methods or LOC. If one explicitly wants to measure the degree of reuse of classes or methods, inheritance metrics [23] should be considered. Inheritance metrics capture the reusability of class and method bodies.

3 Cyber Physical System Case Study: Patrol Robot

A typical example of a CPS is a system which interacts with its environment. For this case study a robotic system is chosen as example, more specific a robotic system which moves around through a *environment*. Sensor data provides the robot with feedback from the environment, adaptations can be made according to the incoming sensor data. Navigation through the environment requires an environmental model. The system should be able to visit points of interest in the environment; these points of interest or *waypoints* are locations in the environment where actions must be performed. The robot must be able to travel through the environment visiting a list of waypoints, a *route*. The robot used in this case study is a EV3 LEGO robot [24], this robot can be programmed in Java using the Lejos API [25], this API runs on a customised java embedded JRE [26].

In this chapter we will describe the application scenario and requirements for the initial system. The initial system design should be energy-unaware but best efforts should be made to prepare the design for possible evolution. *Chapter 3.3* will describe the initial design with the respective design alternatives and choices. *Chapter 4* will introduce new requirements through energy evolution scenarios. Evolution scenarios are identified by conducting a domain analysis. An OO and AO implementation are given; both implementations implement the energy evolution scenario. Finally in *chapter 5* the evaluation method is described, metrics are calculated for both implementations and results are compared.

3.1 Application Scenario

In some buildings like banks/research facilities, there may be a need for security patrolling. The task to check if there are any intruders is time consuming and repetitive; this task can be done manually or can be automated. Automation can be achieved by installing various cameras in the building but this may be quite expensive and overkill for a relative simple task. A simple robot could easily patrol a predefined environment and check for signs of intruders. In the most simplified scenario the robot traverses the environment by following a specific route. If the robot comes across an open door it will see this as a possible security breach. The robot may decide to check the route more frequently if a lot of doors are open. The robot has a limited power supply so it should consider energy usage when undertaking action. If the robot threatens to run out of energy before its shift ends it should adapt its behaviour accordingly. The patrolling frequency depends on the open doors encountered and energy left in the battery. A possible strategy could be to check areas without a security breach less frequently to save power.

3.2 Requirements

The system should fulfil various requirements in order to be able to navigate, patrol and adapt its behaviour. Since we are conducting research only functional requirements will suffice. Given the information from the introduction and the application scenario the following list of requirements is composed:

R1: The robot should be able to navigate from one point in the environment to another point taking the shortest path.

R2: The robot should be able to detect doors.

R3: The robot should be able to detect obstacles.

R4: The robot should be able to detect crossings and corners.

R5: The robot should be able to patrol the environment.

R6: The robot should be to adapt its resting time between two tours (patrol cycles), in case of few to none open doors in the last three tour the resting time should be doubled.

R7: The robot should be able to adapt its behaviour in case of a security breach (if there are open doors detected on a tour).

R7. 1: The robot should be able to stop patrolling and check the exit of the building in case of a security breach.

R7. 2: The robot should be able to sound an alarm in case of a security breach.

Requirements do not capture all functionality; for simplicity some assumptions are made. The following assumptions are made about the robot:

A1: It is assumed the environment model is known to the robot.

A2: It is assumed the environment is a maze of corridors which may cross each other.

A3: It is assumed that when the robot starts its first patrolling cycle the battery is fully charged.

A4: It is assumed the battery capacity is sufficient to patrol the environment at least once.

There are many different design options that can be used to model a CPS, however this research will place some constraints on the possible design options. The initial design choices must take into account:

RD1: The design choices must be based on evolvability, decoupling and modularity.

RD2: The design choices should take reuse into account.

Implementing software in a reusable yet modular way is the main goal of this case study. A design without reuse capabilities cannot evolve without rewriting large chunks of code.

3.3 Initial Design Of The Robot

Initially the system is energy-unaware so it does not take energy usage into account. The designed system exists of a base system which is controlled by a control system. This chapter will present design alternatives for the initial energy-unaware design. Based on the requirements design choices are made to maximise the evolution capabilities of the design. Finally, the initial base system design is presented which will serve as the base system in the OO and AO implementations of this research.

3.3.1 Design Alternatives

To prepare the design as best as possible for future evolution, many design options should be considered; we will present the design options for the environment model and the control part of the system.

Environment Model The environment model of the robot must be modelled into some sort of map which the robot can use to navigate, this process is called mapping. Research about environment mapping for the predecessor of the EV3 robot is conducted by Oliveira, Silva, Lira and Reis [27] . This research classifies robotic mappings as either *metric* or *topological.* Where a metric approach determines the geometric properties of the environment, the topological approach determines the relationships of locations of interest in the environment. Another way to classify mappings would be to partition them *world-centric* or *robot-centric.* Where world-centric mappings represent the map relative to some fixed coordinate system, robot-centric mappings represent maps relative to the robot.

This research compares the standard mapping method of the Lejos framework to their own optimised mapping method. The environment mapping implemented by the Lejos framework is a world-centric and metrical approach; the robot moves forward from a known starting point, until an obstacle is detected with the ultrasonic sensor. Once an obstacle is detected the robot will drive backward and rotate to avoid the obstacle; by doing this the robot is mapping the environment on a 80x80 matrix which represents the environment. In the real world this matrix covers 16 square meters, allowing a detailed representation of small environments. This method is improved by applying a system of *probabilistic mapping* based on the Bayes [28] method. This means that the points in the matrices receive values that represent the probability of an obstacle being present at that point in the environment. Low values indicate that the probability of an obstacle is high, whereas a high value indicates a low probability of an obstacle present at that location.

The Lejos API provides an even simpler environment mapping method namely the *line map* [29]. The environment is represented by line segments; every time the robot turns a new line is started and created in the map. This method is world-centric and a metrical approach, the position of the robot is tracked by observing motor movements. Tracking the robot position based on rotation and speed by motor movement is not very reliable. If an error is made there is no reference for correcting; this result is magnified by subsequent errors.

The conclusion of both of these methods is that we need a high probability of validating an obstacle or waypoint. Both approaches can only map a small sized environment, the patrol environment is too large for both approaches. For this reason a *graph* [30, 31] like mapping method should be considered. The limited sensors of the robot allow the detection of walls, doors and corners, which can be used as waypoints or validation points in the environment. If the robot has a predefined map of the environment available it can explore this environment from a known starting point. These waypoints can be used to validate the position of the robot in the environment model. Again this approach is world-centric and metrical, a topological mapping is not suited for simple environment mapping with as main goal navigation.

Navigation Navigation through the environment should be strategy-based; various strategies should be defined to calculate a route between two points. Possible strategies could be the shortest route, the most efficient route based on the number of obstacles, corners and doors on the route. There is only one possible option for this especially in the case of a graph-based model namely the *Strategy pattern* [19]. The Strategy pattern defines an interface defining all possible methods that must be affected by a certain strategy. Each concrete strategy must override the methods defined by the pattern's interface. Changing strategies is not only related to navigation but also closely related to applying behavioural adaptations.

Control loop Architecture For the control part of the system a control loop architecture should be adopted following the MPAE-K terminology. Properties of interest of the base system must be observed and adaptations must be executed to adapt the base system behaviour. We will discuss design choices for the following three aspects:

- observation
- control
- adaptation

Observation The base system must be observed to monitor changes in behaviour and environment. OO provides various options for the observation of so-called subject objects; we will discuss two of them. The first approach is a *message bus in combination with a queue*, the subjects can post updates to the message bus and the messages are buffered in a queue. The observer can read the incoming messages by applying the FIFO (first in first out) mechanism. A drawback of this approach is that messages are handled in a uniform way, but not explicitly forwarded to a specific observer. All observers must check the queue for updates even if the update is not relevant for them. The second approach is by applying the *Observer pattern* [19]. The Observer pattern excels in the decoupling of subject and observer. The subject must implement the subject interface which defines methods for attaching observers, detaching observers and notifying updates to observers. The observer must implement the observer interface which contains a single action method which is triggered when subjects notify an update. The attach- and detach-mechanism ensure that only interested observers are notified when an update is posted.

Control The control part of the software follows the *MAPE-K* terminology, resulting in four different components. These four components form a control chain or control loop which has as input observed values of interest of the base system and the output exists of adaptation applied on the base system. Challenges in modelling this part of the system come from the fact that multiple data types must be processed. To tackle this problem two solutions exist, namely creating one uniform data handling method or creating a handler method for each different data type.

The *uniform data handler method* results in monitor, analyser and planner interfaces defining one method. Each component, whether monitor analyser or planner, must perform a cast of the uniform argument before performing actual operations on it. Due to all arguments being cast the introduction of new data types by possible evolution scenarios would not be a problem anymore; any argument can be casted to the correct type respectively. Adopting this method which in essence takes over the role of the compiler comes with a lot of problems. Casting is deemed unsafe and encourages runtime errors, aside from that handling data in a uniform way neglects all the features the object model gives us.

The second option of creating a data handler for each possible data type (*unique data handler*) will result in monitor, analyser and planner interfaces defining handlers for every possible data type; each handler will have its own data type as argument. Each event from the base system which has another data type as argument results in a new handler.

Adaptation Applying adaptations to the base system can be achieved in various ways; the main problem here is the need to extend existing objects with new functionality. An option would be to simply extend existing classes by defining new methods and fields within already existing classes. However, by doing this existing components are directly affected by evolution. A possible solution is the adoption of the *Strategy pattern* [19], a new strategy can be used to introduce new behaviour for existing functionality. Depending on the strategy the object may execute the operation in a different way. The *Visitor pattern* [19] also provides a way to define new functionality for existing components. The Visitor pattern defines two interfaces, the visitor and the visitable interface. The visitable interface defines the accept method, which calls the visit method of the visitor interface. The visitable component passes itself as an argument to the visit method. The visitor interface defines visit methods for each of the visitable components, concrete visitors can implement these methods and enrich the functionality of these components in order to apply adaptations.

3.3.2 Design Choices

This section will present the initial design choices regarding the base system and control part of the system. The design choices regarding the environment model as well as the choices related to the interaction between the base system and the control system are given. The choices regarding interaction of the base system and control system affect both systems. However since ideally the base system is unaware of the control system these design choices are discussed together in the control system section.

Environment Model Requirements R1 to R5 specify properties which should be extracted and stored within the environment model. Requirements R2 to R4 mention that points of interest (*waypoints*) should be detected; these waypoints come in various forms like doors, crossings and obstacles. Requirement R1 states that the robot should be able to navigate from one point in the environment to another point. These requirements are properties best modelled in a graph-like environment. The probabilistic matrix mapping and line-map approach do not allow the explicit declaration of waypoints. Determining the positions of the robot on such maps must be determined from motor movement; this way of determining the position is likely to fail due to incorrect feedback from sensors data. Position errors are amplified with each new error, until the robot is completely lost. In general these environment models provide too much detail and do not provide enough certainty.

With the graph model the robot can verify its position on each waypoint; doing this the impact of multiple errors will be minimised since the position can be verified at fixed predefined points in the environment. A *metrical and world-centric graph-based* environment model satisfies all requirements related to the environment model.

Control System For the control system we will give the design choices for the three sub-aspects: observing, control and adaptation. Observing and adaptation choices do not only impact the control system but also the base system.

Observing Requirement R7 requires the observation of the security state of the system. The observation mechanism should couple the base system to the control part of he system. The Observer pattern provides a decoupled and modular way to introduce new observable subjects. New subjects need to implement an observable interface and define an update method to notify interested observers which can register and de-register. In contrast to the message-bus solution only interested observers are notified avoiding possible data race problems. The message queue buffering the uniform data messages must be synchronised since multiple listener threads will poll for new messages at the same time. The message-bus solution cannot be reused (RD1) without casting the messages after extraction of the queue. The Observer pattern is designed for reuse; the attach, detach and update mechanism can be reused for every new observer and subject respectively. The *Observer pattern* focusses on decoupling and reuse and for that reason it is more suitable for evolution.

Control loop coupling The design options for the control part of the system are not based on the functional requirements. The design requirements RD1 and RD2 state that reuse decoupling and modularity should be taken into account. However, since we are following the MAPE-K terminology the components are fixed, however this is not the case for the coupling between the components. Two alternatives were discussed in the design choice section, the uniform data handler and the unique data handler method. The uniform handler can be defined in the respective MAPE component interfaces and be reused without modifying interfaces. The only drawback is that a cast should be applied before the data can be accessed. The unique data handler method defines data handlers for each data type in the control components. However, without performing a cast, data can not be accessed. Both design options are possible according to the design requirements. As said, casting is deemed unsafe and encourages runtime errors, aside from that handling data in a uniform way neglects all the features the object model gives us. In a sense such a mechanism is an extension of the compiler which is forcefully told what to do. One could go even further and store every object in a bitmap and write a custom interpreter to be as flexible as possible, completely ignoring the object model. However, by doing this the implementation would not be object oriented anymore. For this reason the *unique data handling method* is adopted in the initial design.

Adaptation For applying adaptations two solutions were provided, namely the Strategy and the Visitor pattern. The Strategy pattern solution cannot define strategies for objects which do not share a common super type. According to R7.1 and R7.2 at least two different adaptations should be executed, changing route and sounding an alarm respectively. The adaptations in the requirements do not share any common features; if the Strategy pattern should be used Strategy patterns for both objects should be defined. Each new type of action will result in a new Strategy pattern for the object the action is defined on. The Visitor pattern provides a more dynamic way to apply adaptations for different objects. Each concrete visitor must implement the visitor interface which defines action methods for all visitable objects. Each concrete visitor can apply multiple adaptations for different objects. The looser coupling with the base system together with the handling of different objects makes the *Visitor pattern* more suitable to cope with evolution.

3.3.3 Initial Design

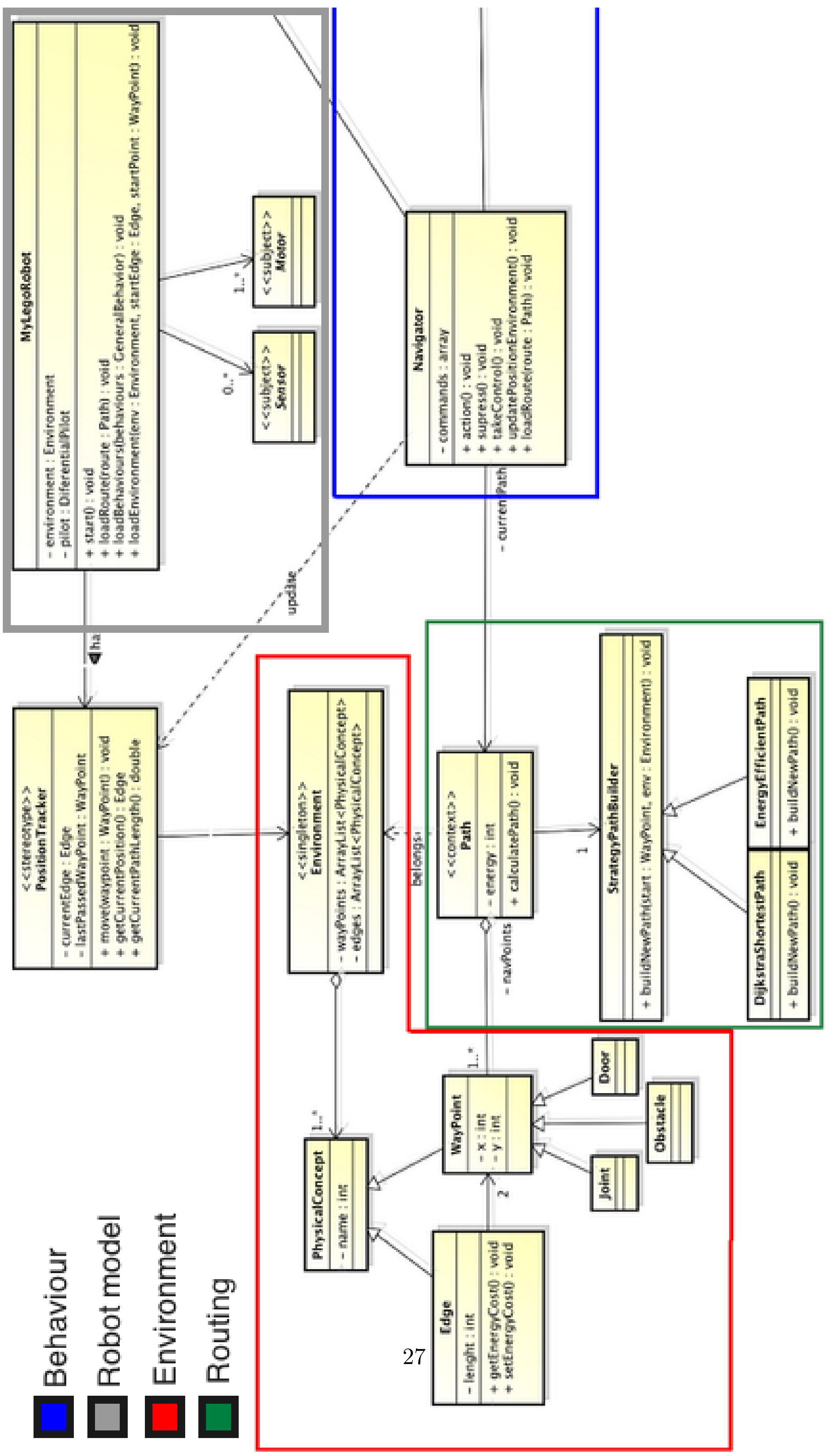
With the design choices known, the initial design can be implemented. This section describes how the base system design is implemented. The base system design is fixed and is used as a basis for both the OO and the AO implementation. Therefore the initial control part of the system is given in the form of a state space diagram; the reason for this is to abstract from language and method constraints.

Base system The initial design as depicted in figure 3 is energy-unaware. The base system contains all base functionality. The most important concerns are:

- Behaviour (blue)
- Environment (red)
- Routing (green)

The initial design is written in OO and provides all initial functionality to adapt system behaviour defined by the initial system requirements.

Figure 3 outlines the base systems design; various patterns were applied. The system/robot is composed of various components like sensors, ports, mechanisms actuators (engines). There is no predefined configuration; any sensor can be connected to any port. A pattern called the *AnyMorphology pattern* [32] is used to configure the system and the hardware components it is composed of. Since any part can be coupled to any other part, a high degree of decoupling is achieved. This makes it possible to configure the system as the user desires with an almost unlimited amount of freedom. Without this pattern new components must explicitly define how they can be coupled to other components. Without a clear general interface extending the system with new hardware the designs modularity would break down. The pattern's generic way of handling and coupling components maximises evolvability concerning new hardware components.



Behaviour
Robot model
Environment
Routing
MyLegoRobot
- environment : Environment
- pilot : DiferentialPilot
+ start() : void
+ loadRoute(route : Path) : void
+ loadBehaviours(behaviours : GeneralBehavior) : void
+ loadEnvironment(env : Environment, startEdge : Edge, startPoint : WayPoint) : void
<<subject>> Motor
<<subject>> Sensor
1..*
0..*
Navigator
- commands : array
+ action() : void
+ supress() : void
+ takeControl() : void
+ updatePositionEnvironment() : void
+ loadRoute(route : Path) : void
- currentPath
update
has
<<stereotype>> PositionTracker
- currentEdge : Edge
- lastPassedWayPoint : WayPoint
+ move(waypoint : WayPoint) : void
+ getCurrentPosition() : Edge
+ getCurrentPathLength() : double
<<singleton>> Environment
- wayPoints : ArrayList<PhysicalConcept>
- edges : ArrayList<PhysicalConcept>
belongs
<<context>> Path
- energy : int
+ calculatePath() : void
- navPoints
1
StrategyPathBuilder
+ buildNewPath(start : WayPoint, env : Environment) : void
EnergyEfficientPath
+ buildNewPath() : void
DijkstraShortestPath
+ buildNewPath() : void
1..*
PhysicalConcept
- name : int
WayPoint
- x : int
- y : int
Door
Obstacle
Joint
2
Edge
- lenght : int
+ getEnergyCost() : void
+ setEnergyCost() : void

Figure 3: Base system design environment, routing and robot model

The environment is *graph-based*, corridors can be seen as edges and waypoints as nodes, waypoints can be doors, crossings or obstacles. Edges connect nodes and form a graph [33]. The environment can be traversed following a path. A path is a list of nodes or waypoints and can be traversed by going from one node to the next. The robot needs some sort of verification points within the environment. Although the sensors are limited, it is possible to detect and follow a wall; corners and crossings can be detected and used as waypoints in the environment. After detection of a waypoint the robot can update the position in the environment. These waypoints which are modelled as nodes in a graph ensure a reliable way to navigate through the environment.

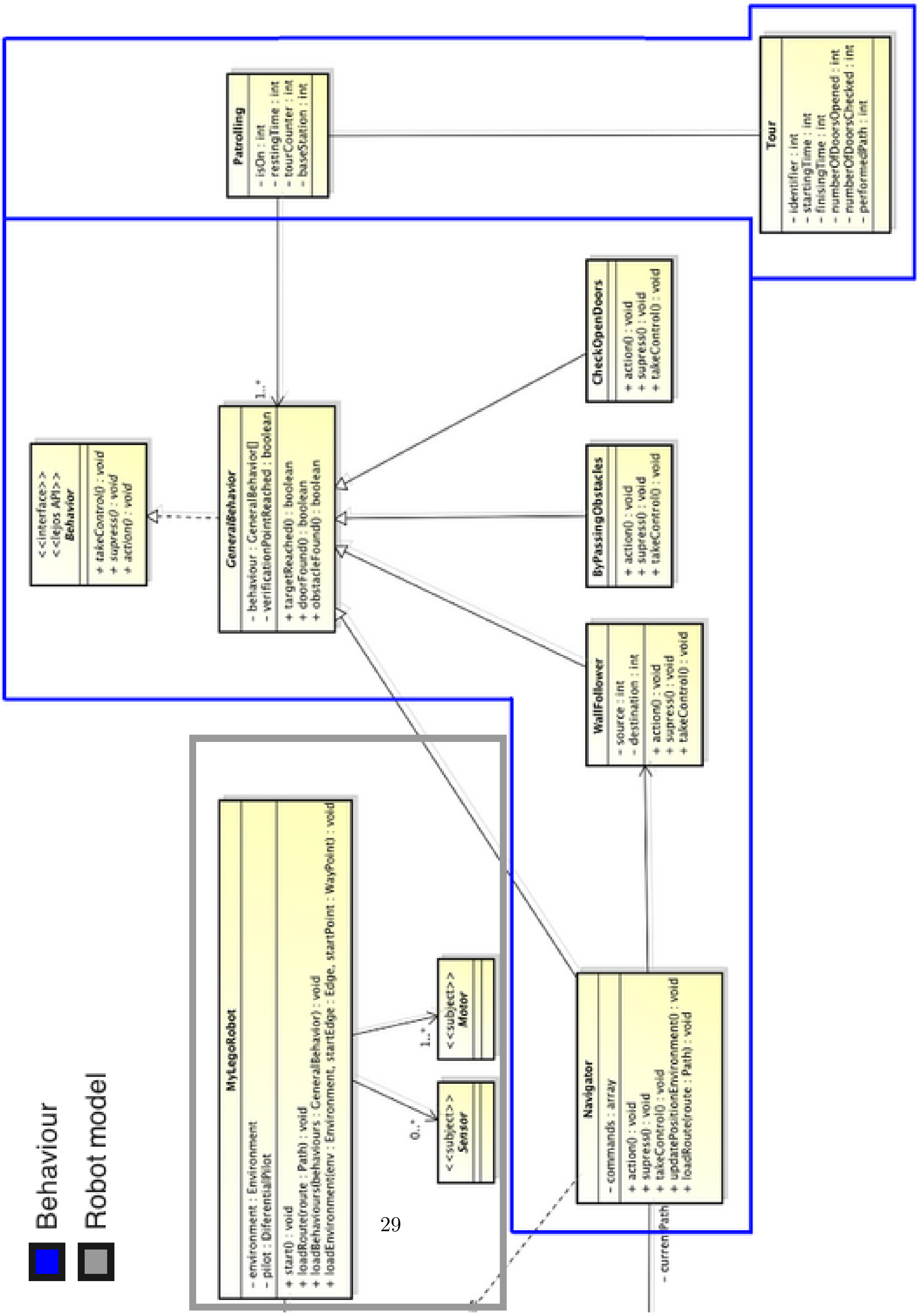


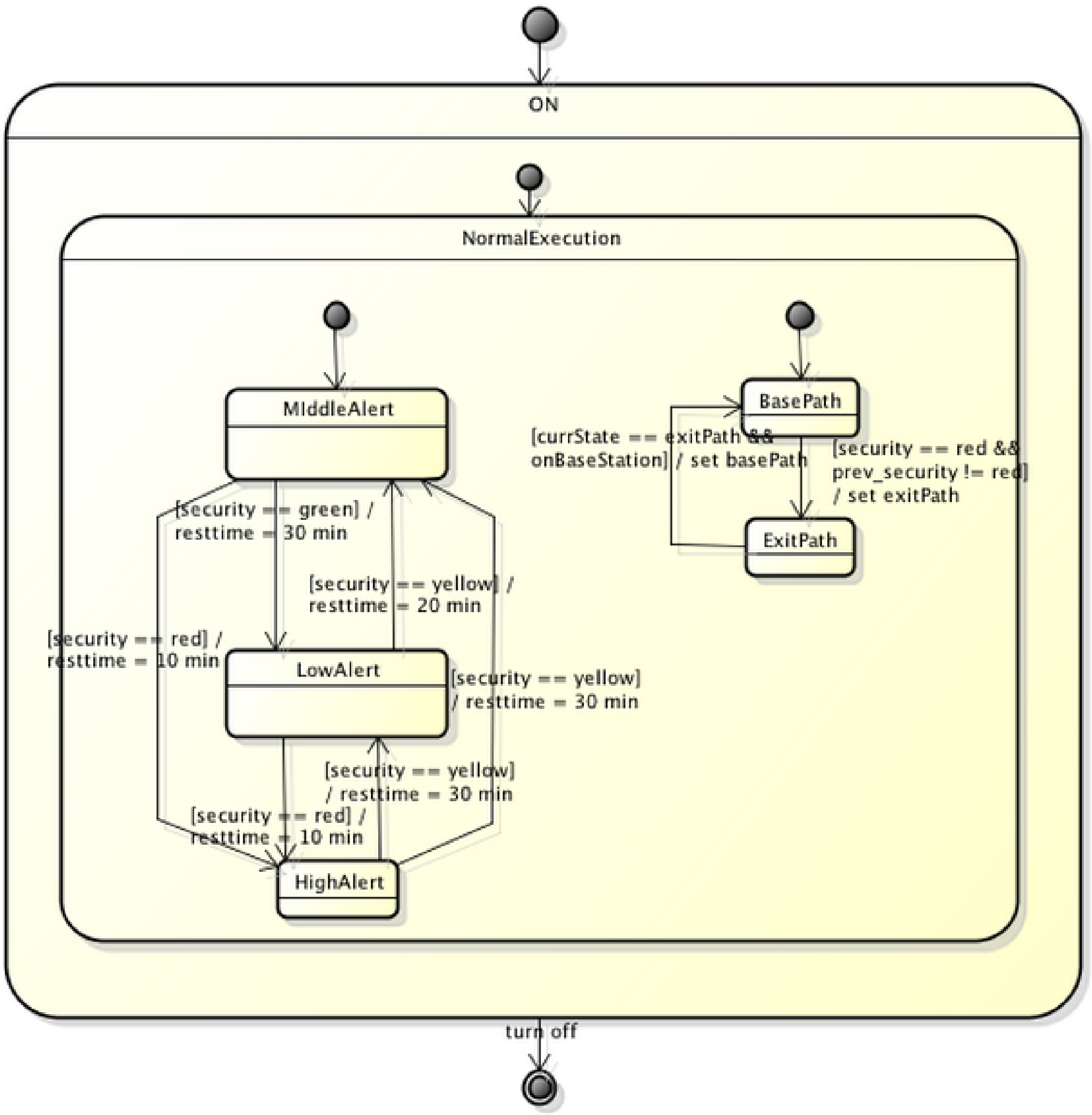
Figure 4: Base system design robot model and behaviour

The behaviour is modelled as a *subsumption architecture* [34] which is illustrated in figure 4. The arbitrator takes various behaviours as input: CheckDoors, EvadeObstacle, FollowWall and Navigate. Each behaviour becomes active once it is triggered. Since the wall serves as a orientation point the robot moves forward along the wall; therefore this behaviour is active most of the time. If a door, crossing or obstacle is detected, other behaviours will take over control of the system. The navigator will update the position of the robot after a new waypoint is detected and passed. A path can be generated in different ways using the *Strategy pattern* [19]. Initially the system uses the Dijkstra shortest path [30, 31] strategy to navigate from one point in the environment to another.

This system will function as the base system; it needs control to adapt to changes. Changes in the environment must be observed, analysed and adaptations must be applied depending on the state of the system.

Control loops The base system is controlled by several control loops. The control loop architecture will be described in *MAPE-K* [3] terminology: Monitor, Adapt Plan and Execute. The Knowledge Base will be hardcoded within the analyser components. The monitor component serves as the interface to observe the base-system. Concrete monitors must be defined to observe various parts of the system. The executor component serves to apply adaptations to the base system. These adaptations are triggered depending on the current state of the planner component. The initial requirements define various scenarios. Figure 5 gives an overview off all initial control states and their hierarchy. A state space diagram is chosen as representation method. The reason for this is to abstract from language and method constraints.

Base Scenario Initially two control loops are present, the *first control loop* monitors the security state of the system (figure 5 left) and regulates the resting time between two tours.



ON
NormalExecution
MIddleAlert
LowAlert
HighAlert
BasePath
ExitPath
[security == green] / resttime = 30 min
[security == yellow] / resttime = 20 min
[security == red] / resttime = 10 min
[security == yellow] / resttime = 30 min
[security == yellow] / resttime = 30 min
[security == red] / resttime = 10 min
[currState == exitPath && onBaseStation] / set basePath
[security == red && prev_security != red] / set exitPath
turn off

Figure 5: Initial state space

A tour is a patrolling cycle of the environment. The security state depends on the open doors found on the route. More open doors translate into a higher security threat. The overall security level of the system should be optimised. If there is a high security threat the patrolling rate should be intensified. A lower security threat means the system is safe so the interval between two tours can be extended. The interval between two patrolling tours will be adapted to the security state of the system.

The *second control loop* determines the system mode (figure 5 right); it also observes the security state of the system. Initially it has two states; patrol/tour or visit the exit of the building. If the system detects a high security threat it should stop patrolling and travel to the exit of the building; if the exit (a door) is open it should sound the alarm. Once the alarm is sounded the robot can resume patrolling. This control loop adapts high level system behaviour. All control loops are managed by a super control loop. The super control loop can switch control loops on and off; therefore it is in control of the system.

4 Energy-aware Robot Design

Software evolution is scenario-driven; scenarios introduce new requirements which force the change or extension of a software program. So far in this case study, energy was not considered in the design; however, the best efforts were made to prepare the design for every possible evolution scenario. Energy is a relevant topic for possible evolution for a CPS; CPS often deals with energy management and in our robot case with a limited power supply. To extract energy evolution scenarios a domain analysis is conducted.

4.1 Domain Analysis

Before energy scenarios can be considered the domain must be analysed. The domain can be divided into two parts namely the system, in this case the robot, and the world surrounding the system called the environment. The robot consists of various hardware parts like sensors, actuators (motors) and a power supply. Some level of abstraction is required here since not every part of the robot is relevant. For example the robot is built from various lego blocks; however it is not necessary to explicitly analyse every lego part used. The environment can be treated into endless detail; however for it to be useful to the robot again some abstractions must be applied. The robot model and the environment model are depicted in figure 6.

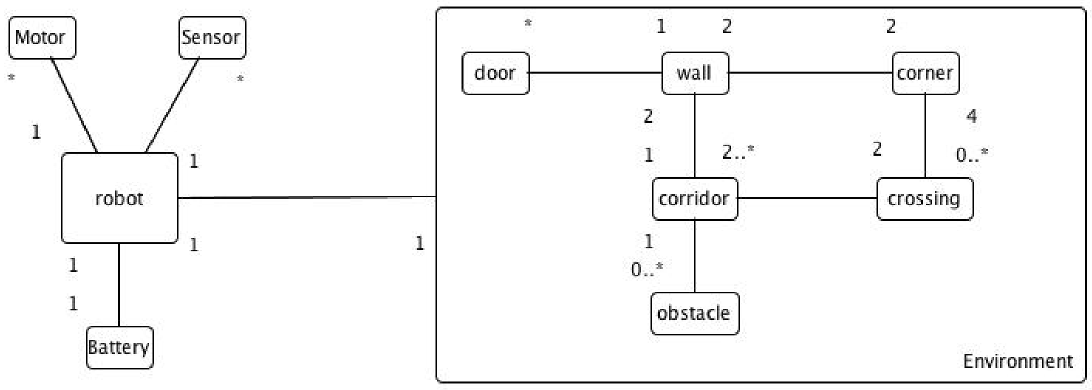


Figure 6: Domain model

The robot model exists of various sensors, motors and a power supply/battery; only actuators and sensor parts are shown for abstraction. The model is based on the view the robot has on the environment; cardinalities are given to clarify the relation between the components. The robot follows the wall (the robots rides along side the wall) and can come across doors and obstacles; once the robot finds a corner it reached the end of a corridor and possibly arrived at a crossing. The environment model can be seen as a graph with nodes and edges. Nodes are abstracted as waypoints which can be doors, obstacles and crossings; edges are the walls which connect the various waypoints. This model provides sufficient detail to extract energy evolution scenarios.

4.2 Scenarios

The most simple scenarios are based on the robot itself; the robot consumes energy and it has sensors which can measure current and voltage. Measuring in itself is not a scenario; therefore an action or adaptation must be defined if the measured value is below or above a certain threshold.

Voltage in itself is no indication of the amount of energy left in a battery. However below a certain threshold it can be assumed the battery is almost empty. If this happens while the robot is away from its base-station (it is patrolling) there is a problem. If the robot runs out of energy during a patrolling cycle it will be stuck somewhere on the route without any options. At the base station a charger can be placed so the robot can recharge itself. In any case it is vital that the robot reaches the base station before it runs out of energy. The following two scenarios should make this possible:

V1: If the voltage reaches the critical point, abort the current task and return to the base station taking the shortest path.

V2: If the voltage reaches the critical point, only the behaviours responsible for moving must kept activated.

Both scenarios complement each other and can coexist; V2 increases the chance of success of scenario V1. Note that it is not certain that the robot will reach the base station. There is no historic energy consumption information, only a voltage threshold. If for some reasons the robot consumes more energy than assumed, the voltage may drop sooner than expected. Even if the robot notices in time the voltage drops below the threshold, this is not enough to ensure the robot reaches the base station.

Another scenario can be extracted by monitoring the current drawn from the power supply. If the drawn current is too high it could fatally damage the system. Although it must be noted this functionality is usually implemented by hardware we will still consider it. A current threshold could be defined just below the critical value to shutdown the system correctly instead of waiting for the hardware to shutdown the power supply. With this knowledge we can define another evolution scenario:

A1: If the current reaches the threshold value (a value just below the critical value), shutdown the system to prevent information loss.

All of the scenarios so far are based on observing sensor values and taking action if a value becomes critical. Scenario V1 states that the robot should be able to reach the base-station at any time, however making this assumption based on voltage reading is far from reliable. Voltage by itself does not provide a reliable way to estimate how much energy is left in the power supply. For this reason energy data must be calculated and stored for future use.

An example environment with nodes and edges is shown in figure 7. It can be seen energy is used to perform an action (activity energy) when arrived at a node. Energy is also used to travel from one node to another (move energy). The left-hand side of the figure shows a graph model of the environment, whereas the right-hand side of the figure depicts the graphs nodes and edges within the actual environment.

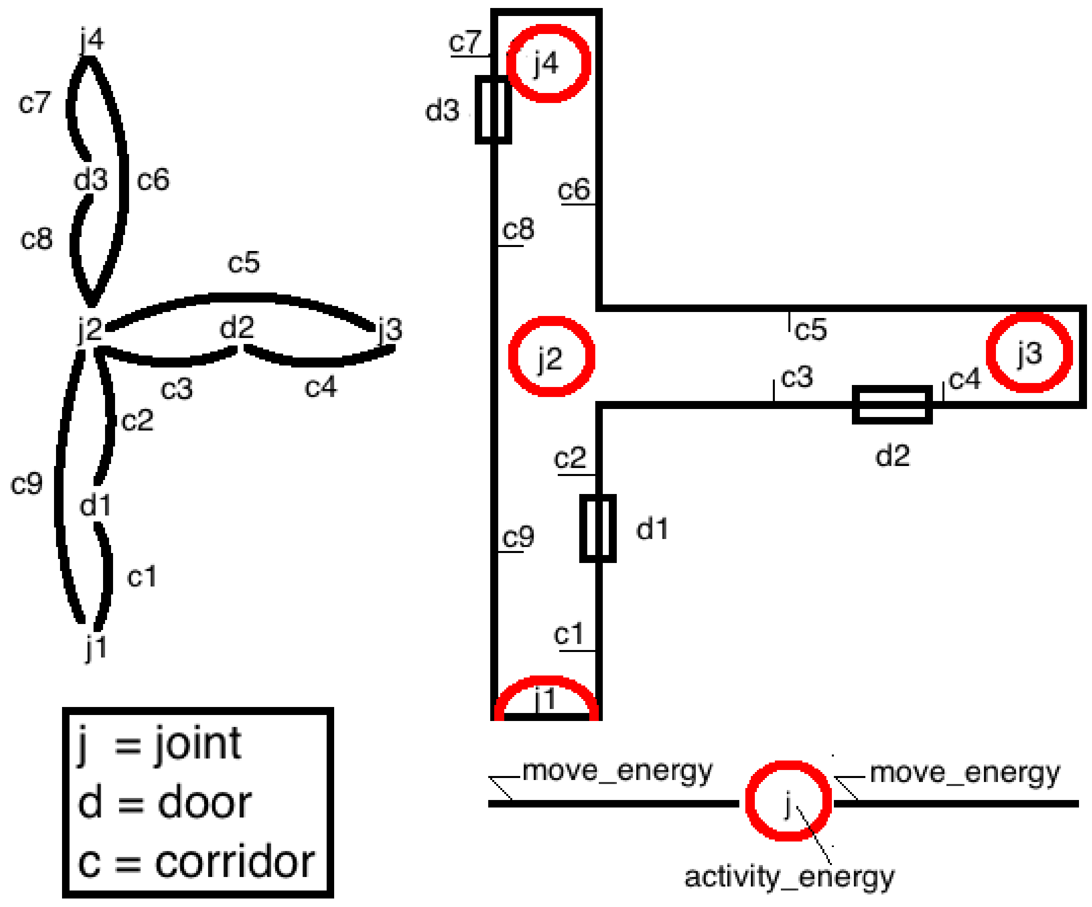


Figure 7: Environment example

To estimate if the robot can return to the base station for every node and edge the energy consumption must be known. Energy consumption can be obtained by performing a exploration tour through the environment. During this tour energy consumption will be stored for each node and edge; this historic information should be used when planning a route through the environment. In this case, if the robot runs low on energy it can compare the energy left in the battery with the energy needed to travel the most energy efficient path back to the base station. If this check is performed at every new waypoint where the robot arrives, this will be far more reliable then only checking the voltage level.

To increase the chance of the robot to arrive at the base station before it runs out of energy the following scenarios should be added to the list of evolution scenarios:

- Power

  P1: The system shall calculate the power for the entire system. (One current meter and voltage meter attached to the battery supply.)

- Energy

  E1: Calculate energy consumption between two points in time.

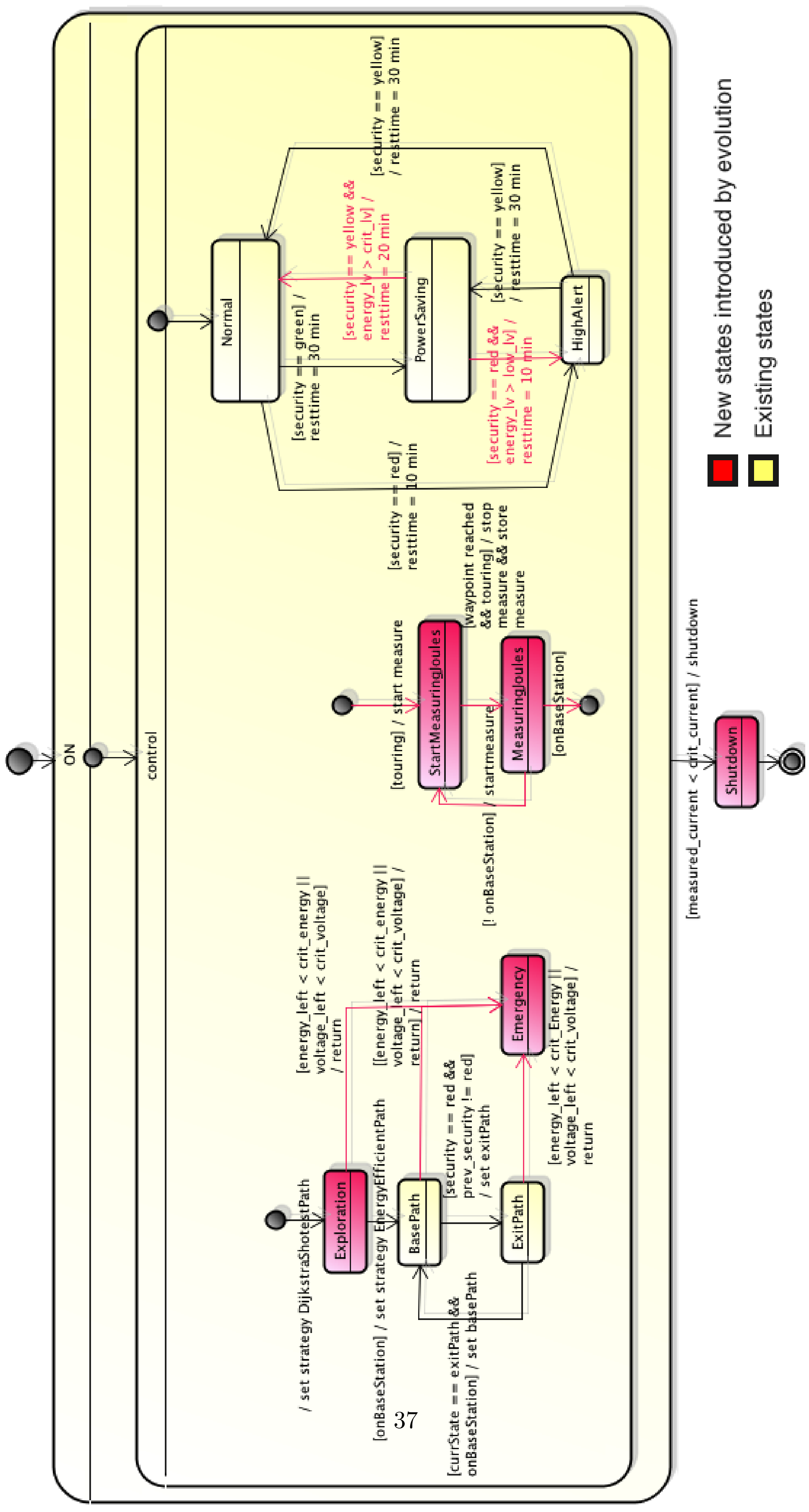
  E2: The system can store energy measurements for later use.

  E3: The system is able to travel between two waypoints using a path that consumes minimal energy. Energy management is based on historic information.

  E4: The system is able to return to the base station at all time before it runs out of energy.

Where the first scenarios were almost entirely based on the system (the robot) there was no feedback from the environment model; the latter one introduces historic information stored within the environment model which can be used to optimise energy management.

State space evolution All new states and events introduced by the different evolution scenarios affect the control system's state space. The impact on the control system's state is illustrated in figure 8, news states and events are shown in red. A new shutdown state is introduced by scenario A1 to stop the system in case of an emergency; this state is triggered by a critical current event. We see two new states in the control loop which determines the system mode; the exploration state is introduced by E3 and a emergency state is introduced by scenarios V1 and E4. The newly introduced events are critical voltage and critical energy respectively. These events form new transitions in the rest-time process and affect existing transitions in the system-mode process.



ON
control
Normal
PowerSaving
HighAlert
[security == green] / resttime = 30 min
[security == yellow] / resttime = 30 min
[security == yellow && energy_lv > crit_lv] / resttime = 20 min
[security == yellow] / resttime = 30 min
[security == red && energy_lv > low_lv] / resttime = 10 min
[security == red] / resttime = 10 min
StartMeasuringJoules
MeasuringJoules
[touring] / start measure
[waypoint reached && touring] / stop measure && store measure
[! onBaseStation] / startmeasure
[onBaseStation]
Exploration
BasePath
ExitPath
Emergency
/ set strategy DijkstraShotestPath
[onBaseStation] / set strategy EnergyEfficientPath
[security == red && prev_security != red] / set exitPath
[currState == exitPath && onBaseStation] / set basePath
[energy_left < crit_energy || voltage_left < crit_voltage] / return
[[energy_left < crit_energy || voltage_left < crit_voltage] / return] / return
[energy_left < crit_Energy || voltage_left < crit_voltage] / return
Shutdown
[measured_current < crit_current] / shutdown
New states introduced by evolution
Existing states

Figure 8: Initial state space

An entire new process is added to measure energy and can be seen in the middle of figure 8. This process measures energy and stores energy usage concerning waypoints and edges in the environment model. This new process is introduced by scenarios E1 and E2. Two new events are introduced: one to start measuring and one to stop measuring.

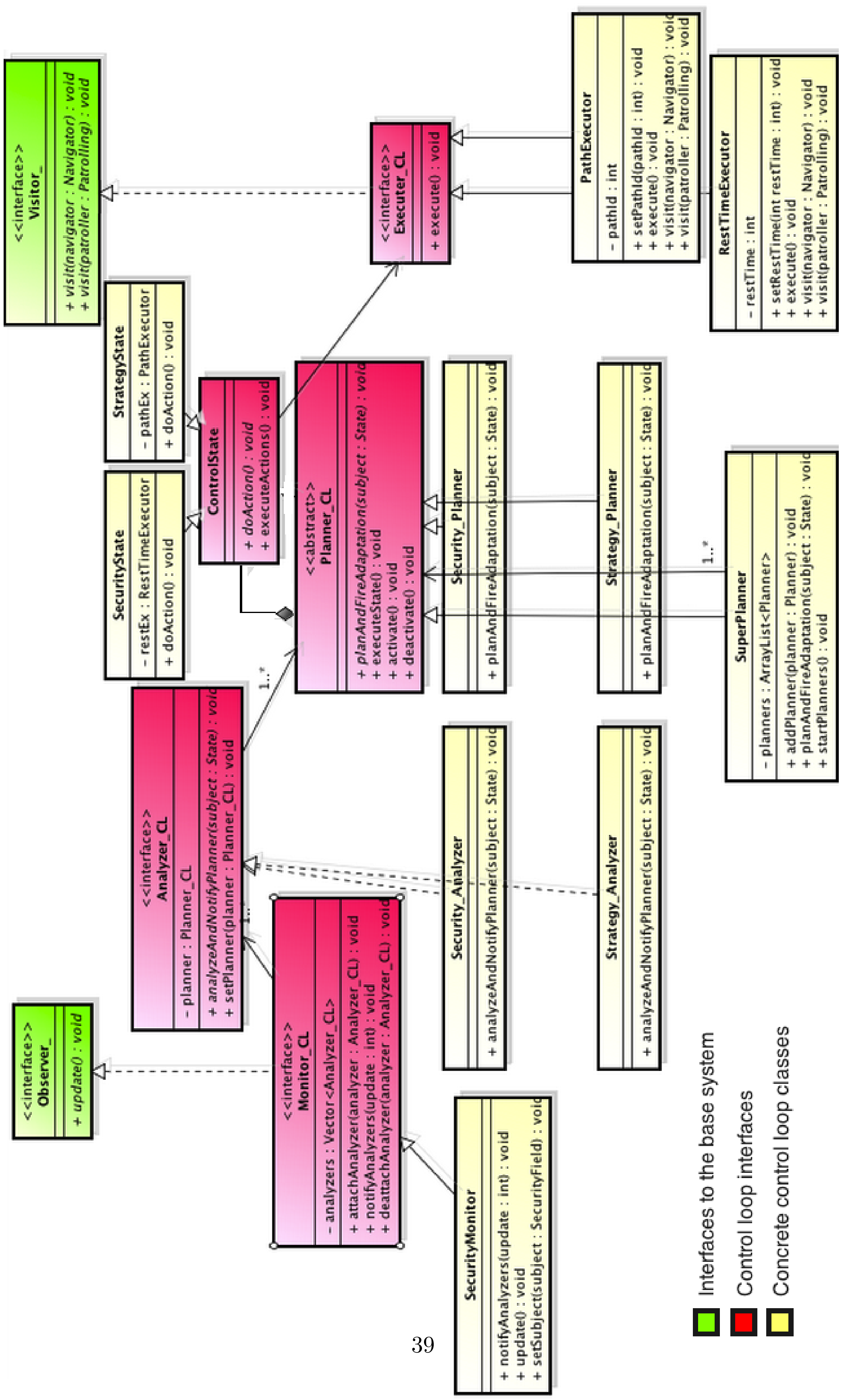
4.3 Evolution Impact on OO Implementation

In this section we will discuss the implementation for the initial base-scenario as well as the impact of the energy evolution scenarios on the initial design. The main focus is on the control part of the system and the interaction of the control system with the base system. The control-part of the system is implemented conform to the MAPE-K terminology. Challenges here are how to observe the base system's properties of interest and how to apply adaptations to the base system without tangling the control loop architecture and the base system. Rewriting part of the base system to extend it with new functionality is deemed undesired. This section will assume the initial design of the base system and describe the control system architecture before and after evolution as well as the evolution impact on the base system.

4.3.1 Initial Control System Architecture

The observation interface of the control system architecture is implemented using the Observer pattern [19]. Observable subjects implement the subject interface of the Observer pattern, allowing them to be observed by multiple observers. The observers are automatically notified once the subject is updated by the base system. The observer must be registered to the subjects before they receive updates. In the control system architecture the monitor components are the observers; concrete monitors implement the observer interface.

To apply adaptations to the base system the Visitor pattern [19] is adopted. Once visitable components implement the visitable interface, they can be visited by external components. After a visitable class accepts the visitor, the visitor can apply adaptations to the visitable class. Concrete visitors can define new functionality without rewriting the visitable class. This mechanism allows the extension of existing classes in a modular and decoupled way. The Visitor pattern is used to let the executor components of the control loop apply adaptations to the base-system. Executors implement the visitor interface which defines abstract methods for all visitable components.



<<interface>> Visitor_
+ visit(navigator : Navigator) : void
+ visit(patroller : Patrolling) : void
<<interface>> Executer_CL
+ execute() : void
PathExecutor
- pathId : int
+ setPathId(pathId : int) : void
+ execute() : void
+ visit(navigator : Navigator) : void
+ visit(patroller : Patrolling) : void
RestTimeExecutor
- restTime : int
+ setRestTime(int restTime : int) : void
+ execute() : void
+ visit(navigator : Navigator) : void
+ visit(patroller : Patrolling) : void
StrategyState
- pathEx : PathExecutor
+ doAction() : void
SecurityState
- restEx : RestTimeExecutor
+ doAction() : void
ControlState
+ doAction() : void
+ executeActions() : void
<<abstract>> Planner_CL
+ planAndFireAdaptation(subject : State) : void
+ executeState() : void
+ activate() : void
+ deactivate() : void
Security_Planner
+ planAndFireAdaptation(subject : State) : void
Strategy_Planner
+ planAndFireAdaptation(subject : State) : void
SuperPlanner
- planners : ArrayList<Planner>
+ addPlanner(planner : Planner) : void
+ planAndFireAdaptation(subject : State) : void
+ startPlanners() : void
1..*
<<interface>> Analyzer_CL
- planner : Planner_CL
+ analyzeAndNotifyPlanner(subject : State) : void
+ setPlanner(planner : Planner_CL) : void
1..*
Security_Analyzer
+ analyzeAndNotifyPlanner(subject : State) : void
Strategy_Analyzer
+ analyzeAndNotifyPlanner(subject : State) : void
<<interface>> Observer_
+ update() : void
<<interface>> Monitor_CL
- analyzers : Vector<Analyzer_CL>
+ attachAnalyzer(analyzer : Analyzer_CL) : void
+ notifyAnalyzers(update : int) : void
+ deattachAnalyzer(analyzer : Analyzer_CL) : void
SecurityMonitor
+ notifyAnalyzers(update : int) : void
+ update() : void
+ setSubject(subject : SecurityField) : void
1..
🟩 Interfaces to the base system
🟥 Control loop interfaces
🟨 Concrete control loop classes

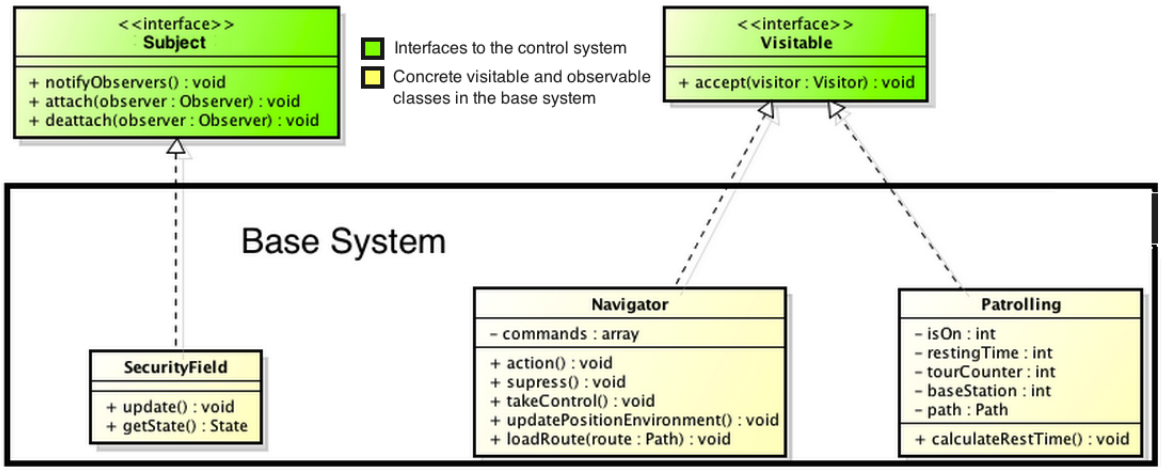
Figure 9: Base scenario control loop architecture

The complete control system implementation for the base scenario (without energy) is depicted in figure 9. The class diagram shows red classes which are the interfaces/abstractions of the four control loop components. Note that the controlState component is not a control loop class, the planner class contains states and depending on the state different adaptations can be applied. Green classes represent the Observer and Visitor pattern interfaces; they form the coupling between the control system and base system. All concrete control loop classes introduced by the base-scenario are coloured yellow.

4.3.2 Base-scenario Implementation

Two control loops can be distinguished in figure 9, namely the security-loop and system-mode-loop. Both loops share the mutual security-state monitor; the system-mode-loop adapts routing strategies according to this information, whereas the security-loop adjusts the resting time to the security state of the system.

The observable and visitable components of the base system are depicted in figure 10. These are the components which supply information (observable) to the control loop and the components which allow the application of adaptations (visitable) to the base system.



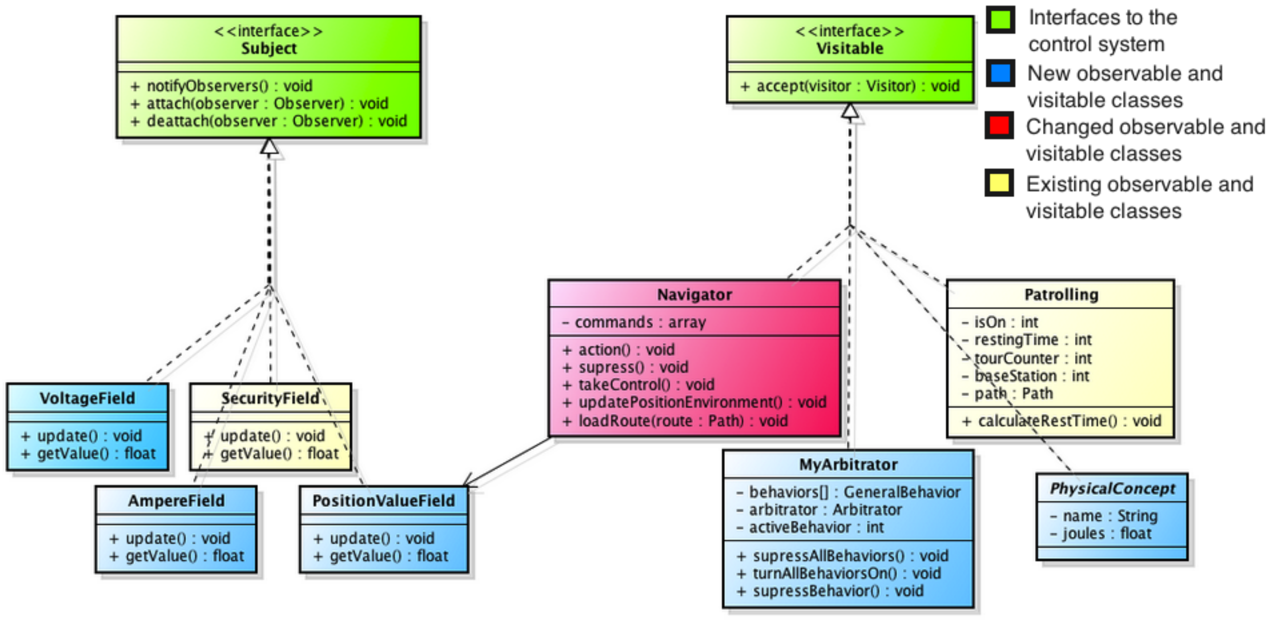
<<interface>>
Subject
+ notifyObservers() : void
+ attach(observer : Observer) : void
+ deattach(observer : Observer) : void
Interfaces to the control system
Concrete visitable and observable classes in the base system
<<interface>>
Visitable
+ accept(visitor : Visitor) : void
Base System
SecurityField
+ update() : void
+ getState() : State
Navigator
- commands : array
+ action() : void
+ supress() : void
+ takeControl() : void
+ updatePositionEnvironment() : void
+ loadRoute(route : Path) : void
Patrolling
- isOn : int
- restingTime : int
- tourCounter : int
- baseStation : int
- path : Path
+ calculateRestTime() : void

Figure 10: Coupling control system architecture to the base system.

Initially only the security state is observed; depending on the state adaptations must be applied to the navigation and patrolling components. The current route of the robot can be adjusted in the navigator, whereas the resting time (time between two tours) can be adjusted through the patrolling class.

4.3.3 Energy-awareness Evolution Impact

The Observer and Visitor pattern ensure a high degree of modularity and decoupling. The design choices where made to maximise reuse, decoupling and modularity. To test the system's evolvability energy-awareness functionality is introduced. By introducing energy-awareness functionality the number of observable and visitable components is increased. Figure 11 shows all new observable and visitable classes. Blue classes are newly introduced classes, red classes need modification to satisfy the new requirements.

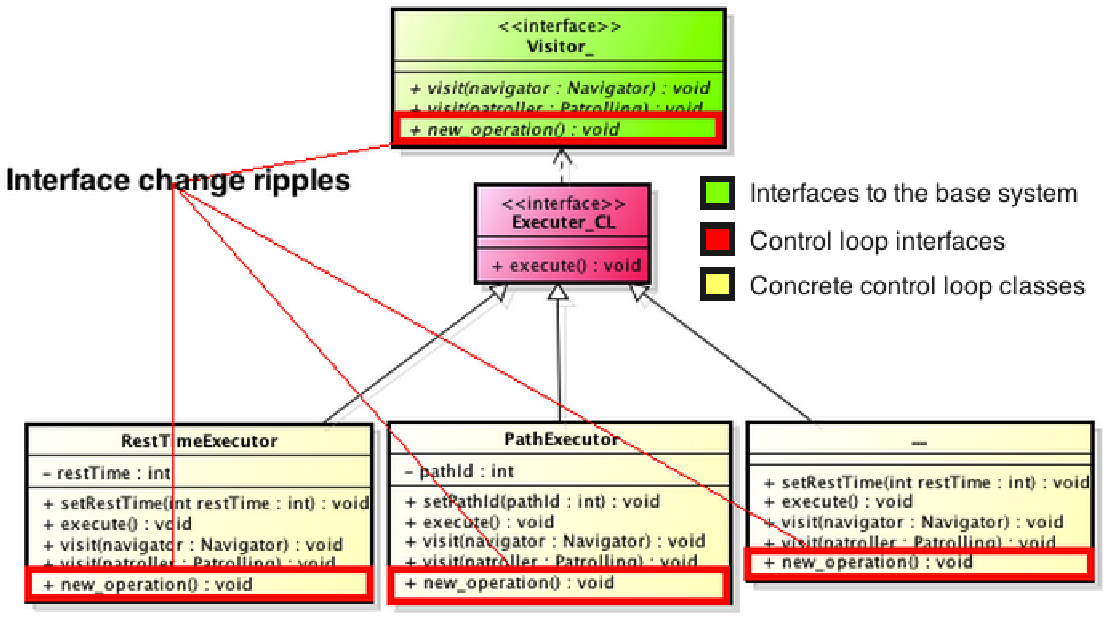


<<interface>>
Subject
+ notifyObservers() : void
+ attach(observer : Observer) : void
+ deattach(observer : Observer) : void
<<interface>>
Visitable
+ accept(visitor : Visitor) : void
🟩 Interfaces to the control system
🟦 New observable and visitable classes
🟥 Changed observable and visitable classes
🟨 Existing observable and visitable classes
Navigator
- commands : array
+ action() : void
+ supress() : void
+ takeControl() : void
+ updatePositionEnvironment() : void
+ loadRoute(route : Path) : void
Patrolling
- isOn : int
- restingTime : int
- tourCounter : int
- baseStation : int
- path : Path
+ calculateRestTime() : void
VoltageField
+ update() : void
+ getValue() : float
SecurityField
+ update() : void
+ getValue() : float
AmpereField
+ update() : void
+ getValue() : float
PositionValueField
+ update() : void
+ getValue() : float
MyArbitrator
- behaviors[] : GeneralBehavior
- arbitrator : Arbitrator
- activeBehavior : int
+ supressAllBehaviors() : void
+ turnAllBehaviorsOn() : void
+ supressBehavior() : void
PhysicalConcept
- name : String
- joules : float

Figure 11: Modification to the base system

Figure 11 shows the voltage and ampere fields (both data fields in the battery class modelled as a separate class) as two of the new observable classes (blue). These classes are introduced because before energy can be considered we need to measure voltage and current. With voltage and current known at a given moment in time, power (formula 2) can be calculated for that moment in time. Evolution scenario's V1,V2, E3 and E4 adapt the behaviour of the robot. Because of this the navigator will need additional adaptations; new visitors must be declared to add additional adaptation functionality. Recall that the robot should be able to notice when it is running out of energy; the robot should return to the base station as soon as it is running low on energy. A new route must be calculated to navigate from the current position back to the base-station.

These newly introduced observable classes affect the monitor interface; new event handlers must be created to handle each new observable class. This will not only affect the monitor but also the analyser and planner components of the control loop architecture. Each of these classes must handle the new event, for which they need a handler. The same applies to new visitable components, which need new adaptation handlers. Figure 12 illustrates the ripples caused by evolution for the Visitor pattern.



<<interface>>
Visitor_
+ visit(navigator : Navigator) : void
+ visit(patroller : Patrolling) : void
+ new_operation() : void
Interface change ripples
<<interface>>
Executer_CL
+ execute() : void
Interfaces to the base system
Control loop interfaces
Concrete control loop classes
RestTimeExecutor
- restTime : int
+ setRestTime(int restTime : int) : void
+ execute() : void
+ visit(navigator : Navigator) : void
+ visit(patroller : Patrolling) : void
+ new_operation() : void
PathExecutor
- pathId : int
+ setPathId(pathId : int) : void
+ execute() : void
+ visit(navigator : Navigator) : void
+ visit(patroller : Patrolling) : void
+ new_operation() : void
...
+ setRestTime(int restTime : int) : void
+ execute() : void
+ visit(navigator : Navigator) : void
+ visit(patroller : Patrolling) : void
+ new_operation() : void

Figure 12: Ripples due to interface change visitor example

Each extension of the visitor interface by a new visitable class results in modifications of all existing concrete visitors. In this case the visitors are the concrete executor components of the control loop architecture. All concrete executors must make the new method to visit the new visitable class concrete. This means all concrete executor classes suffer from the introduction of newly introduced visitable classes.

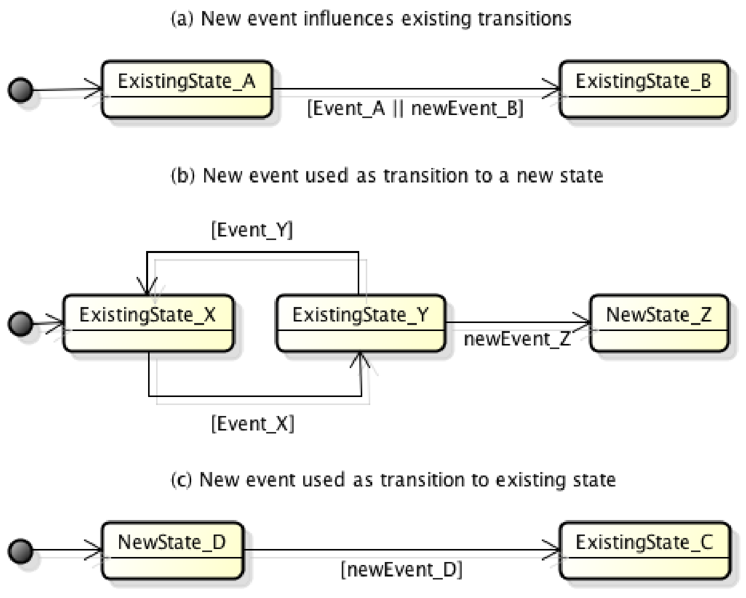


Figure 13: Scenarios for introducing new events.

All concrete monitor, analyser and planner components suffer from the introduction of newly introduced observable components. They must implement the new event handlers defined by the respective interfaces. The new event handlers influence the planner class the most, the state space must be extended by new events and actions (adaptations). The scenarios to introduce new events in this case study are given in figure 13. New events can affect existing transitions (figure 13(a)) but they also can form new transitions to new states (figure 13(b)). Another option is a transition from a new state to an existing state (figure 13(c)). A final possibility is that a new event forms a transition from one existing state to another existing state. This option is not shown in figure 13 because it is unlikely to happen; only if the initial state space was incomplete or badly modelled this option is possible. We do not consider this scenario as evolution impact but as a result of poor design of the state space.

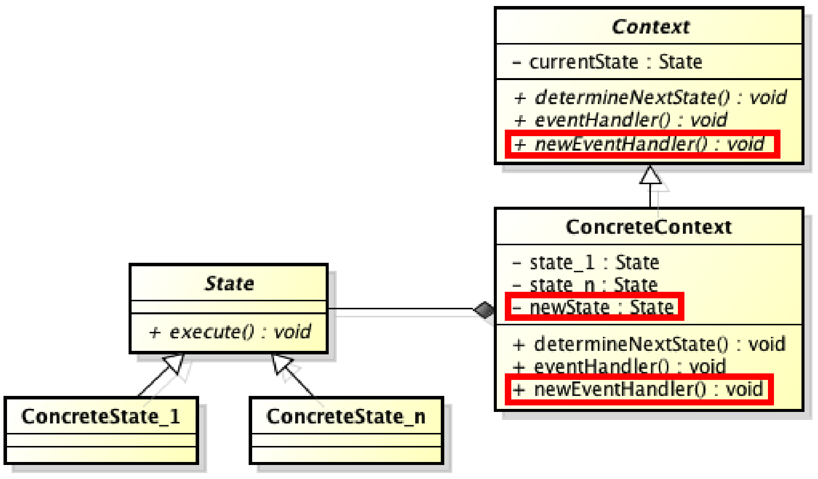
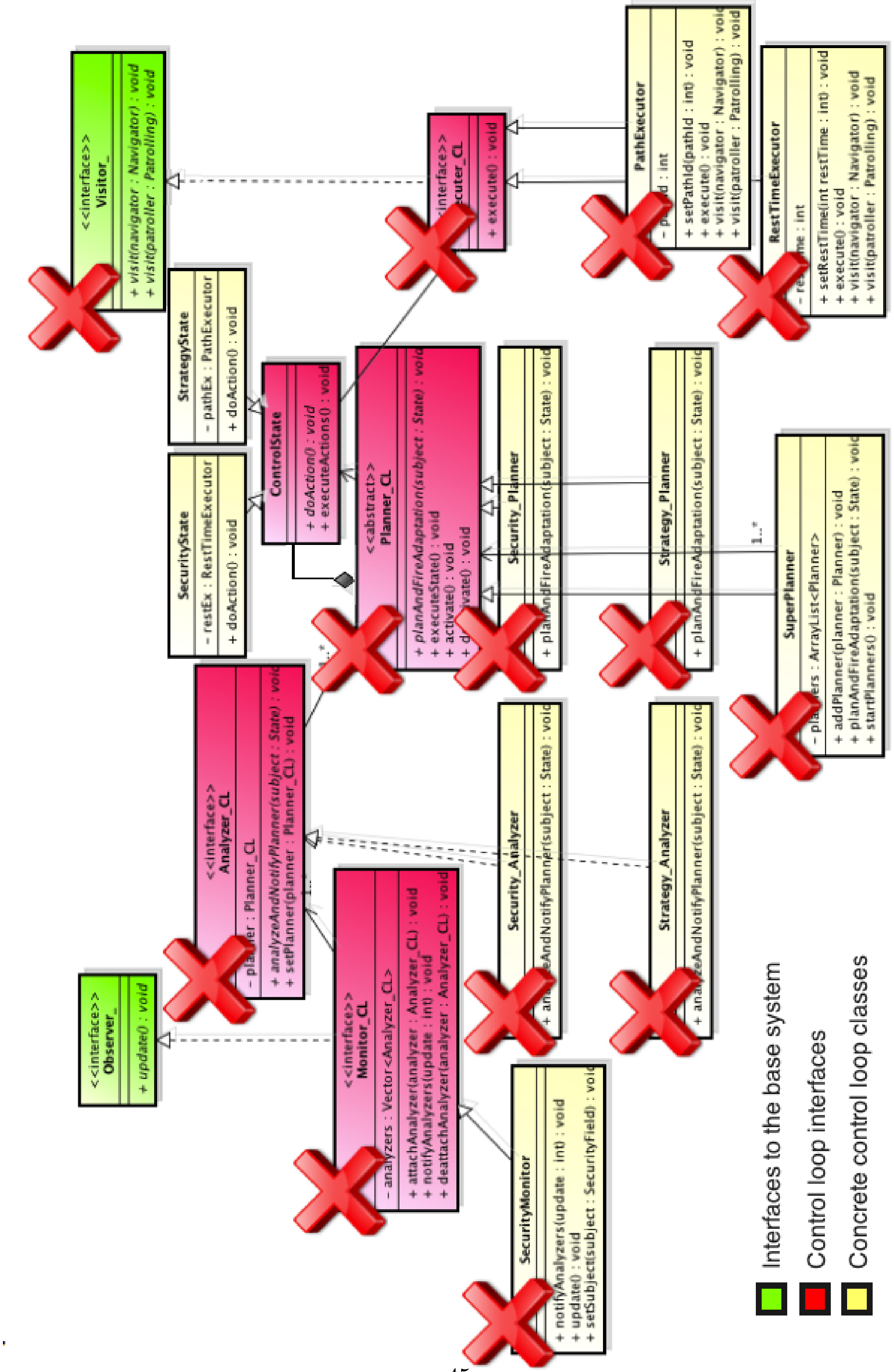


Figure 14: Introduction of new states and events.

State space evolution of the planner leads to rewriting the state space context. The generic impact of state space evolution is shown in figure 14. New state fields and event handlers must be introduced to existing contexts. New event handlers affect the context interface. In our case the planner class is the context; therefore new event handlers must be added to the planner interface. This means that every concrete planner must make the new event handler concrete.

All MAPE-K components suffer from evolution; the Monitor, Analyser and Planner components suffer from the introduction of new events handlers, whereas the Executor class suffers from the introduction of new adaptation handlers. If we look at the control loop design as a whole as depicted in figure 15 it can be seen that it almost breaks down completely after energy is introduced.



<<interface>> Visitor_
+ visit(navigator : Navigator) : void
+ visit(patroller : Patrolling) : void
<<interface>> Executer_CL
+ execute() : void
PathExecutor
- pathId : int
+ setPathId(pathId : int) : void
+ execute() : void
+ visit(navigator : Navigator) : void
+ visit(patroller : Patrolling) : void
RestTimeExecutor
- restTime : int
+ setRestTime(int restTime : int) : void
+ execute() : void
+ visit(navigator : Navigator) : void
+ visit(patroller : Patrolling) : void
StrategyState
- pathEx : PathExecutor
+ doAction() : void
SecurityState
- restEx : RestTimeExecutor
+ doAction() : void
ControlState
+ doAction() : void
+ executeActions() : void
<<abstract>> Planner_CL
+ planAndFireAdaptation(subject : State) : void
+ executeState() : void
+ activate() : void
+ deactivate() : void
Security_Planner
+ planAndFireAdaptation(subject : State) : void
Strategy_Planner
+ planAndFireAdaptation(subject : State) : void
SuperPlanner
- planners : ArrayList<Planner>
+ addPlanner(planner : Planner) : void
+ planAndFireAdaptation(subject : State) : void
+ startPlanners() : void
1..*
<<interface>> Analyzer_CL
- planner : Planner_CL
+ analyzeAndNotifyPlanner(subject : State) : void
+ setPlanner(planner : Planner_CL) : void
Security_Analyzer
+ analyzeAndNotifyPlanner(subject : State) : void
Strategy_Analyzer
+ analyzeAndNotifyPlanner(subject : State) : void
<<interface>> Observer_
+ update() : void
<<interface>> Monitor_CL
- analyzers : Vector<Analyzer_CL>
+ attachAnalyzer(analyzer : Analyzer_CL) : void
+ notifyAnalyzers(update : int) : void
+ deattachAnalyzer(analyzer : Analyzer_CL) : void
SecurityMonitor
+ notifyAnalyzers(update : int) : void
+ update() : void
+ setSubject(subject : SecurityField) : void
🟩 Interfaces to the base system
🟥 Control loop interfaces
🟨 Concrete control loop classes

Figure 15: Ripples due to interface changes after introduction of energy evolution scenarios

As can be seen in figure 15, all control loop components are affected when energy-awareness functionality is introduced. Interface changes force concrete classes to make new events or adaptation handlers concrete.

Conclusion The main problems the OO design suffers from are all related to the introduction of new events. The base system must be adapted to be able to notify the event to the control system. The control system interface needs modification with each newly introduced event. New events lead to state space evolution of the planner class of the control loop architecture. State space evolution results in rewriting the state space context. New states apply new forms of adaptation through the Visitor pattern. New visitable base system classes must accept visitors and the visitor interface must be extended with a handler for each of these classes.

4.4 Evolution Impact on AO Implementation

AO provides solutions for many problems in the OO implementation. The AO join-point model and point-cut mechanism allow to define new functionality for existing classes in the form of advices. It also provides mechanisms to implement interfaces and additional methods by the means of aspects to extend existing classes. This section will focus on how AO can solve the problems present in the OO implementation.

4.4.1 AO Improvements Initial Control System Design

The AO point-cut mechanism can be used to create a looser coupling between the base system and control system. All problems encountered in the OO implementation can be traced back to the control system coupling and interaction with the base system. Recall that the Observer and Visitor patterns did not cope well with new events and adaptations introduced by evolution scenarios. To cope with evolution, interfaces had to be extended, existing concrete classes implementing these interfaces needed to be updated to handle the new interface methods. AO allows the extension of existing classes and interfaces by means of aspects. Research is conducted on how design pattern modularity can be improved by adopting AO techniques [19, 35]. Sant'Anna, Garcia, Kulesza, Lucena and Von. Staa [35] conducted a research which provides a comparison between an OO and an AO version of most of the known design patterns. Recall that evolution ripples appeared with the introduction of new events (Observer) and new adaptations (Visitor). Also the control loop interfaces were affected by the introduction of new events. It must be noted that the AO implementation does not run on the robot and is simulated on a standard java platform on a general purpose computer.

Observer Pattern Where the OO implementation explicitly must update the data field we want to observe, in the AO implementation a point-cut can be defined to intercept the update method. Once the update method is intercepted it can access the changed data field. The point-cut and advise in listing 1 illustrate how an update method is intercepted and passed down to the concrete observers which are the monitor components of the control loop. The OO Observer pattern can be replaced by an AO implementation of the pattern. The AO implementation is taken from a study conducted by Hannamann and Kiczales [19]. In the AO implementation of the pattern, observable components in the base-system no longer need to implement an observable/subject interface. This solves the problem of unwanted modifications to the base-system code caused by the introduction of new events. This results in a control system which is completely decoupled from the base system.

The AO Observer pattern is implemented as shown in listing 1. The aspect declares two interfaces namely Subject and Observer. These interfaces must be implemented by the respective concrete subjects and concrete observers. A abstract point-cut and method are declared that must be made concrete by a concrete observer aspect. The subjectChange point-cut can handle all concrete subjects in a uniform way. The updateObserver point-cut must have a concrete advice declared in the concrete observer aspects. This is due to the fact that the concrete subject update data may be in different formats; each type must be passed to an analyser of the same type.

Listing 1: Abstract Observer pattern aspect [35]

```
protected interface Subject  { }
protected interface Observer { }

protected abstract pointcut subjectChange(Subject s);
after(Subject subject): subjectChange(subject) {
                Iterator iter = getObservers(subject).iterator();
        while ( iter.hasNext() ) {
                updateObserver(subject, ((Monitor)iter.next()));
                }
}

protected abstract void updateObserver(Subject subject, Monitor observer);
```

A concrete observer aspect is shown in listing 2. The aspect declares interfaces for the concrete subject and concrete observer classes. The subjectChange point-cut is made concrete and declared in the update-method of the concrete subject. The updateObserver method must be made concrete to notify components interested in the new data. In our case the observer fulfils the role of the monitor component of the control loop architecture, so the monitor will notify registered analysers who can process the data.

Listing 2: Concrete observer aspect [35]

```
declare parents: ConcreteSubject implements Subject;
declare parents: ConcreteObserver implements Observer;

protected pointcut subjectChange(Subject subject):
        call( [returntype] [methodToInterceptConcreteSubject] ) && target(subject);
protected void updateObserver(Subject subject, Monitor observer) {
        observer.notifyAnalyzers(subject.getUpdate());
}
```

The aspect shown in listing 2, supported with aspects to extend the control loop interfaces and the concrete control loop components, solves the problems present in the OO Observer pattern. Listing 3 illustrates how all control loop components can be extended without modifying the existing class.

Listing 3: Visitor pattern extension AO

```
public abstract class Monitor{
        public abstract void notifyAnalyzers(int update);
}

public class ConcreteMonitor extends Monitor{
        public void notifyAnalyzers(int update){
                //code
        }
}

public abstract aspect MonitorAspect {
        public abstract void Monitor.notifyAnalyzers(New_Type update);
}

public aspect ConcreteMonitorAspect extends MonitorAspect{
        public void SecurityMonitor.notifyAnalyzers(New_Type update) {
                // code
        }
}
```

The monitor component is used as example; it defines an event handler for a specific data type. After evolution new events with different data types are introduced. New event handlers are added to the existing object by the means of an aspect. The abstract class (in this example) or interface is extended by an abstract aspect defining abstract event handlers. These new abstract event handlers must be made concrete by a concrete aspect. The concrete aspect defines the new concrete event handlers for the concrete monitors.

Visitor Pattern The other connection to the base system, the Visitor pattern, whose purpose is to add new functionality to base system components, can also benefit from AO mechanisms. In AO programming it is possible to declare a parent class for existing classes. This mechanism can also be used to let a class implement an interface. Additionally it can be used to implement the concrete methods defined by this interface for the respective class. This way an existing class can be enriched without modifying the existing code of the base system. Also visitable base system components do not need to implement the visitable interface anymore. A visitable component had to implement the visitable interface to give visitors permission to visit it and apply adaptations. An aspect can declare a parent for a concrete visitable class and therefore let the class implement the visitable interface. The concrete accept method defined by the visitable interface can be made concrete in this aspect.

The AO implementation of the Visitor pattern uses aspects to solve evolution problems present in the OO version of the pattern. The original OO Visitor pattern is enriched by extending the visitor interface as in listing 4.

Listing 4: Visitor pattern extension AO

```
public interface Visitor {
        public abstract void visit(Type_1 visitable);
        ...
        public abstract void visit(Type_n visitable);
}

public aspect VisitorAspect {
        public abstract void Visitor.visit(NewType_1 visitable);
        ...
        public abstract void Visitor.visit(NewType_n visitable);
}
```

The existing interface is extended by an aspect which declares new abstract methods for the interface. This allows the programmer to extend an interface without rewriting it. Any concrete class implementing the interface must still be extended by the new methods declared by the aspect. The aspect code to declare new methods for a concrete existing visitor class is given in listing 5.

Listing 5: Visitor pattern extension AO

```
public aspect ConcreteVisitorAspect {

        public void ConcreteVisitor.visit(NewType_1 visitable) {
                // custom code
        }

        ...

        public void ConcreteVisitor.visit(NewType_n visitable) {
                // custom code
        }
}
```

The aspect declares a new concrete method for every visitable component declared by the VisitorAspect from listing 4. This mechanism allows the extension of existing visitor classes by aspects without rewriting.

Another problem present in the OO version of the pattern, were the ripples that occurred when new visitable components were introduced by evolution. In OO, visitable classes (base system component) had to implement the visitable interface and they had to make the accept method concrete, resulting in unwanted modifications of the base system. Again an aspect can be used to declare a parent class for a concrete visitable component. The same aspect can make the accept method concrete, illustrated in listing 6.

Listing 6: Visitor pattern extension AO

```
public interface Visitable  {
        void accept(Visitor visitor);
}

public aspect VisitableAspect {
          public void MyVisitableClass.accept(Visitor visitor) {
                  visitor.visit(this);
          }
          declare parents: MyVisitableClass implements Visitable;
}
```

The visitable interface remains the same as in the OO implementation. The declaration of visitable classes is now handled in the aspect code instead of in the base system code. The visitable aspect declares a parent class, the Visitable interface; the accept method declared by the Visitable interface is made concrete. This AO implementation of the visitor pattern decouples the visitor pattern from the base system. New visitable components can be made visitable by aspect code. Also the abstract visit methods can be added to the existing interface within aspect code. Evolution of the pattern interfaces is solved by creating aspects defining evolution functionality.

State space Evolution Although AO programming allows us to extend existing classes and interfaces, this mechanism should not always be used. Reuse should be considered when applying aspects. There was one other problem present in the OO implementation regarding the state space evolution. This is best illustrated when focussing on the evolution of the state space of the planner components of the control loop. Recall there was a problem when introducing new states and events to an existing context.

AO programming allows the programmer to declare an around advice to override the execution of the method that determines the next state. For this to work, existing states should be known, or, better the context should be known. One cannot extend an existing context if this context is not known. AO programming provides some mechanisms that can help to solve this problem. In AspectJ aspects can be declared privileged; if an aspect is declared privileged it can access private and protected members of classes, for which it defines new functionality (methods or advices). However, reusing this solution for multiple contexts is impossible. Let us illustrate this problem in detail by showing some code fragments.

In listing 7 an abstract and concrete context are declared in OO; in addition an abstract aspect is declared to enrich the abstract context. The AO aspect adds new states and event handlers to the abstract context class.

Listing 7: Context code

```
public abstract class Context{
        protected State currentState;
        public abstract void HandleEvent_X();
        public abstract void determineNextState();
}

public class ConcreteContext extends Context{
        private State state_1;
        ...
        private State state_n;
        public void HandleEvent_X(){..}
        public void determineNextState(){..}
}

public abstract aspect AbstractContextAspect {
        public abstract void HandleNewEvent_X();

        public abstract pointcut stateChange(Context c);
        public abstract pointcut stateInit(Context c);
}
```

The OO context's abstract methods are made concrete in the ConcreteContext class. States are context dependent and therefore declared in the ConcreteContext class. The abstract context class only defines a currentState field to keep track of the current system state; every other functionality is encoded in the ConcreteContext class. The abstract aspect defines a new abstract event handler and two point-cuts, one to override initialisation of the initial state (stateInit) and another to override the state-transitions (StateChange). Note that in the abstract aspect the abstract context object is used as parameter for both abstract point-cuts. This is necessary since we want to reuse the point-cuts; concrete aspects should make these point-cuts concrete.

However, when concrete advices are defined we need to handle every context in an individual way. Every context has its own unique states and transitions. The goal is to extend the state space of the concrete contexts; existing states and events can be accessed if the concrete aspect is declared privileged. The problem is that concrete context states cannot be accessed if the context is passed like a general context parameter to the advices. Of course it is possible to perform a cast, but we already deemed casting as unsafe and undesired in the OO implementation. However, if we consider casting applied in a generic way as encoded in listing 8, it is potentially more reliable.

Listing 8: Generic casting example

```
advice() : pointcut() {
        Class c = Class.forName(returnConcreteClass());
        AbstractClass ac = (AbstractClass)c.newInstance();
        //advice code
}

public class ConcreteContext{
        public String returnConcreteClass() {
                return "ThisIsMyConcreteClass";
        }
}
```

As can be seen it is possible to statically look up a class and use reflection to make the explicit cast. Before doing anything concrete in the advice, a method is called to request the string identifier for the concrete context to apply the cast. After the cast it is possible to access private and protected states of the context. At first this dynamic casting solution does look quite dynamic and casting errors seem unlikely. However, if other non-context classes use a similar construct, they can be mistaken for a context class. If this happens the cast will still fail at runtime since there is no guarantee the program is dealing with a context.

Listing 9 illustrates the complete aspect code to extend a existing state space.

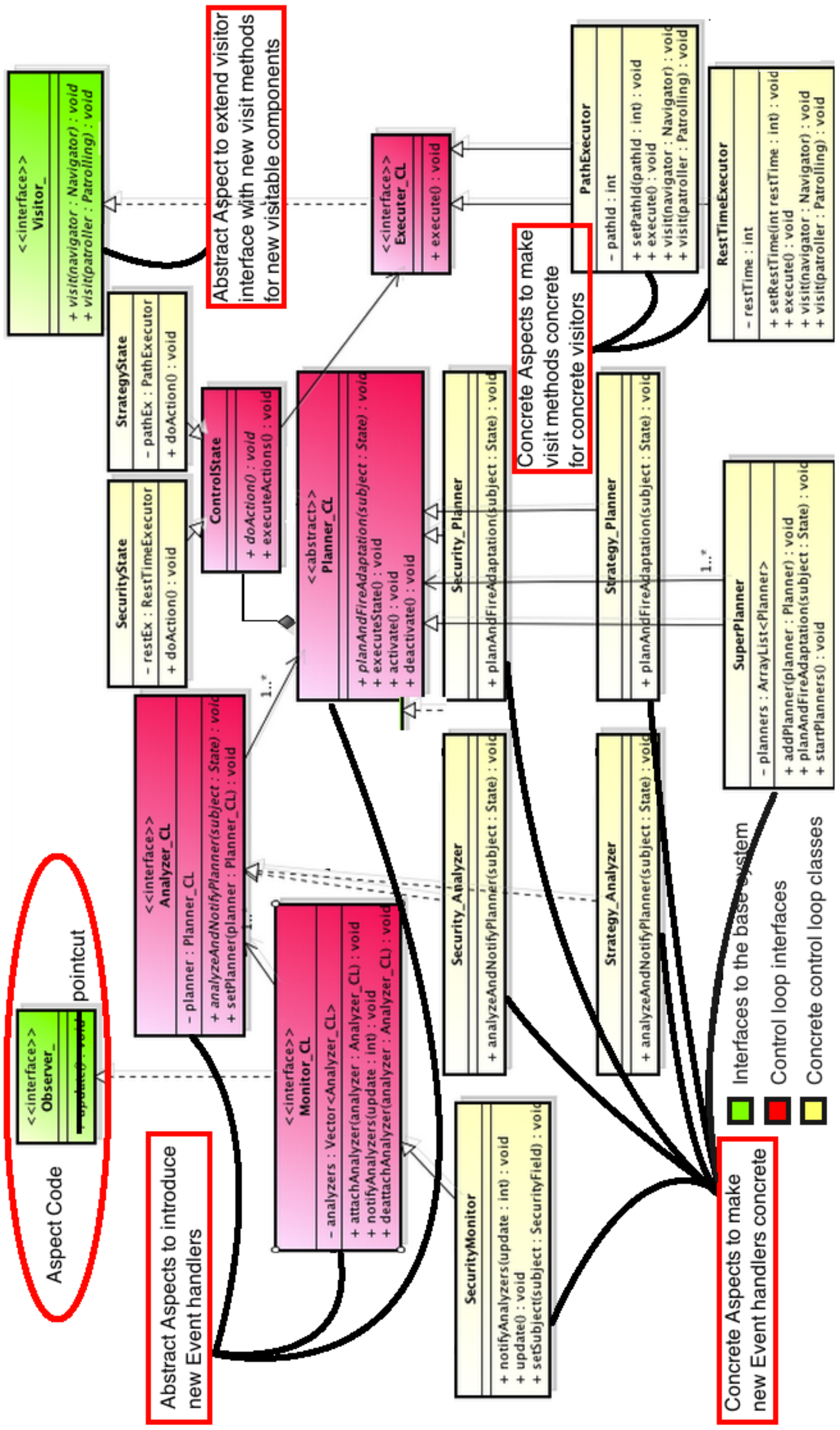
Listing 9: A concrete context aspect extending AbstractContextAspect

```
priviliged aspect ConcreteContextAspect {
        public void HandleNewEvent_X()

        public pointcut stateChange(Context c):
                call(void ConcreteContext.determineNextState(..)) && target(c);

        void around(Context c): stateChange(c) {
                ConcreteContext cc = (ConcreteContext) c;
                // transition code
                ... // set state
        }

        public pointcut stateInit(Context c):
                initialization( ConcreteContext.new(..))&& target(c);

        after(Context c): stateInit(c) {
                ConcreteContext cc = (ConcreteContext) c;
                // set initial state
                cc.currentState = cc.state_1;
        }
}
```

As can be seen, in every advise a cast must be performed before one can access existing (private) states and event information from the concrete contexts. Aside from potential casting errors, the mechanism can still work if we do not take reuse of code into account. However, one of the goals of this study is to maximise reuse of code. Even if we do not consider reuse, new future evolution scenarios may introduce other new states which force the change or declaration of a new advice to override the advices defined by the concrete context aspect. New evolution will lead to stacking advices on top of each other to cope with evolution.

4.4.2 Base-scenario Implementation

The AO improvements of the Observer and Visitor pattern increased the modularity and decoupling of the design. The application of aspect code in the existing control loop architecture is depicted in figure 16. The observer and visitor interfaces are coloured green to indicate the coupling to the base system. Control loop interfaces are coloured red to point out the abstract control loop architecture components. Aspect code modifications are shown in red squares and circles. We will point out the modifications one by one starting from the left-hand side. The Observer pattern is extended with a point-cut to intercept the update methods of the respective subject classes in the base system. For this reason the base system no longer suffers from the introduction of new observable subjects. Newly introduced event handlers can be added by means of aspect extensions of the control loop interfaces. Figure 16 (middle) shows the extension of the monitor, analyser and planner component interfaces; as well as the aspect extension of the respective concrete components. To extend the respective component interfaces with new event handlers, abstract aspects are defined; concrete aspects can make the newly introduced event handler concrete for every concrete monitor, analyser and planner.



Abstract Aspect to extend visitor interface with new visit methods for new visitable components
<<interface>> Visitor_
+ visit(navigator : Navigator) : void
+ visit(patroller : Patrolling) : void
<<interface>> Executer_CL
+ execute() : void
PathExecutor
- pathId : int
+ setPathId(pathId : int) : void
+ execute() : void
+ visit(navigator : Navigator) : void
+ visit(patroller : Patrolling) : void
RestTimeExecutor
- restTime : int
+ setRestTime(int restTime : int) : void
+ execute() : void
+ visit(navigator : Navigator) : void
+ visit(patroller : Patrolling) : void
Concrete Aspects to make visit methods concrete for concrete visitors
StrategyState
- pathEx : PathExecutor
+ doAction() : void
SecurityState
- restEx : RestTimeExecutor
+ doAction() : void
ControlState
+ doAction() : void
+ executeActions() : void
<<abstract>> Planner_CL
+ planAndFireAdaptation(subject : State) : void
+ executeState() : void
+ activate() : void
+ deactivate() : void
Security_Planner
+ planAndFireAdaptation(subject : State) : void
Strategy_Planner
+ planAndFireAdaptation(subject : State) : void
SuperPlanner
- planners : ArrayList<Planner>
+ addPlanner(planner : Planner) : void
+ planAndFireAdaptation(subject : State) : void
+ startPlanners() : void
1..*
<<interface>> Analyzer_CL
- planner : Planner_CL
+ analyzeAndNotifyPlanner(subject : State) : void
+ setPlanner(planner : Planner_CL) : void
Security_Analyzer
+ analyzeAndNotifyPlanner(subject : State) : void
Strategy_Analyzer
+ analyzeAndNotifyPlanner(subject : State) : void
<<interface>> Monitor_CL
- analyzers : Vector<Analyzer_CL>
+ attachAnalyzer(analyzer : Analyzer_CL) : void
+ notifyAnalyzers(update : int) : void
+ deattachAnalyzer(analyzer : Analyzer_CL) : void
SecurityMonitor
+ notifyAnalyzers(update : int) : void
+ update() : void
+ setSubject(subject : SecurityField) : void
<<interface>> Observer_
+ ~~update() : void~~
pointcut
Aspect Code
Abstract Aspects to introduce new Event handlers
Concrete Aspects to make new Event handlers concrete
🟩 Interfaces to the base system
🟥 Control loop interfaces
🟨 Concrete control loop classes

Figure 16: Base scenario control loop architecture AO

The right-hand side of figure 16 shows the application of aspect code in the Visitor pattern. New visitable components can be added to the visitor interface in the same way the control loop interfaces are extended. An abstract aspect is defined to extend the visitor interface; for each existing concrete visitor a concrete aspect can be defined to make the newly introduced visitor methods concrete. The coupling to the base system which is the visitable interface can be extended in a similar way. The visitable interface does not need modification, however the accept method needs to be made concrete for all concrete visitable components. A concrete aspect can be defined to let a new visitable component implement the visitable interface; then the accept method defined by the visitable interface is made concrete in the same aspect. This solution again result in no evolution impact to the base system at all.

4.4.3 Energy-awareness Evolution Impact

AO solved quite some evolution problems present in the OO implementation. The base-system is completely unaffected when new components must be made visitable or observable. Also the Monitor, Analyser and Planner components do not suffer from the introduction of new events anymore, they can be extended by aspects defining new event handlers.

Compared to OO many problems are solved and a higher degree of decoupling between the base system and control system is achieved. The problems related to the extension of control loop interfaces with new events and adaptations are completely solved; aspects add new functionality to existing components. The only problem remaining in the AO implementation is state space evolution. Aspects cannot prevent rewriting existing state space contexts. The main reason for this is that the introduction of new states leads to rewriting the state space context. Even if aspects are used to override state transition with the point-cut and advice mechanism, future evolution still leads to rewriting the advice. The advices are stacked on top of each other, old advices become obsolete and overridden by new advices. New advices should take into account every earlier advise the state space is scattered over; this will lead to rewriting the state space in the new most recent advice. Which is essentially the same outcome as we had in the OO implementation. Other techniques should be considered to find a solution for the state space evolution problem; event-based modularisation techniques possibly can provide a solution.

5 Experiments

To be able to determine which of the implementations performs better regarding decoupling, component extension and modularity, we conducted a series of measurements. Various metrics are used to evaluate the modularity and growth. To evaluate the impact by scenario, *SAAM* [22] is applied. This scenario-based evaluation technique illustrates impact by scenario, for every evolution scenario minor metrics (number of LOC, components and operations) are measured to give insight in the impact of every separate scenario on the implementation. The OO and AO implementation are compared on concern diffusion metric [21]. Concern diffusion metrics can be compared if components and operations related to a concern in both implementations are calculated. The number of components (3) can be calculated by adding up all classes and aspects. We use # symbol to indicate 'number of'.

$$\#Components = \#Classes + \#Aspects \tag{3}$$

The number of operations (4) can be calculated by adding up all methods and advices.

$$\#Operations = 1 * \#Methods + \#Advices \tag{4}$$

This chapter will describe the evaluation method, e.g. which metrics and tools are used to evaluate the OO and AO implementations. The results are presented separately for the OO and AO implementations before the results are evaluated and compared to each other. Finally an interpretation and a discussion about the results are given.

5.1 Quantitative Evaluation of Evolution Impact

To evaluate the OO and AO implementation we will perform a quantitative evaluation of evolution impact. We will give an overview of the metrics used for evaluation as well as of the tools we used to extract them.

Tools and Metrics General metrics are extracted by a tool called *Metrics 1.3.6* [36]. This tool can be added as a plugin to eclipse; once activated every time a project is build, metrics will be calculated for the project. The tool provides a wide range of metrics, however not all metrics are suitable to determine evolution impact. The following metrics are selected to show evolution impact:

- Components
  - #classes*
  - #aspects*
- Operations
  - #methods*
  - #advices (counted as methods)*

- #LOC (excluding comment and blank lines)*

A star indicates that the metric can be extracted by the tool; with these metrics some additional metrics can be calculated. AO code influences how the metrics are counted. If the tool deals with aspect code aspects are counted as classes and advices are counted as methods.

SAAM General metrics do not measure scenario-based evolution impact; to evaluate the effect of evolution between two subsequent software versions *SAAM* is applied. A quantitative evaluation of scenario-based evolution is achieved by applying various minor metrics (listed below). Metrics are gathered in various subcategories: classes, aspects, methods and lines of code:

- Classes
  - #new
  - #changed
  - #deleted
- Aspects
  - #new
  - #changed
  - #deleted
- Methods
  - #new
  - #new methods existing classes
  - #changed
  - #deleted
- LOC (excluding comment and blank lines)
  - #new
  - #deleted

A component or operation is changed if it is modified in a newer software version. Note that there is no metric for changed LOC; to detect if a line of code is changed is very hard to determine. There exist algorithms [37] that can make a fair guess if a line of code is changed but there is no absolute certainty. A new line is a line of code present in the new version and not in the old version. Some new lines can be changed lines of code but it is impossible to detect which lines are new and which lines are changed. A deleted line of code is a line of code present in the old version but not in the new. A deleted line could be changed into a new line of code but again in this case the line should be categorised as a changed line of code. However one cannot say with certainty if a line is changed or not. Therefore lines are categorised into new and deleted lines of code. The sum of all new LOC and deleted LOC is the number of different LOC between two subsequent software versions.

RippleTool To extract these metrics our tool called *RippleTool* [38] is used. This tool extracts the different metrics between two subsequent source code versions. Conventional version management tools like *Git* [39] only provide metrics related to LOC (including comment and white spaces). There exist additional plugins like *GitStats* [40] which can generate metrics related to author and commented LOC. These metrics do not give insight in the changes between two versions of source code. The *Metrics 1.3.6* [36] includes empty lines and comment into the LOC count; RippleTool extracts new and deleted LOC metrics which do not include comment and empty lines.

RippleTool takes as input two different source code versions and outputs a log containing the modifications. The same metrics can be extracted from aspect files without problems. However, aspects are not counted as classes but as separate entities. If the project contains aspects, the tool will calculate separate metrics for the aspect and object part.

The metrics extracted with RippleTool can be used to calculate *concern diffusion metrics* [21]. Concern diffusion metrics express the diffusion of a concern over components (CDC) or operations (CDO). All changes between the version before evolution and the version after evolution are caused by the energy evolution functionality. The number of components can be calculated with formula (3); this must be done manually and is not done by the tool. Also the number of operations must be calculated manually and can be done by formula (4). A simple addition of changed and new components in the evolved software version can be made to calculate the *CDC* metrics:

$$\#ConcernDiffusionoverComponents(CDC) = \\ \#New\_Components + \#Changed\_Components \quad (5)$$

The *CDO* metric can be calculated by adding up all new and changed operations in the evolved software version:

$$\#ConcernDiffusionoverOperations(CDO) = \\ \#New\_Operations + \#Changed\_Operations \quad (6)$$

Together these metrics will give a good overview what the effects of the different evolution scenarios are. Which components are affected by which scenarios and especially where and if ripples appear in the design.

5.2 Results

This chapter will present the results separately for the OO and AO implementation for both implementations; a short explanation is given how the results relate to the problem parts of the designs.

5.2.1 The Impact of Energy-awareness Functionality to the OO Design

The general metrics of the design are given in table 1. Metrics are calculated before and after evolution; the base system and control system metrics are given separately .

	# of classes/ components	# of interfaces	# of methods/ operations	# of interface methods	LOC
Base System Before Evolution	36	3	255	4	1841
Control System Before Evolution	22	1	61	1	447
Base System After Evolution	36	3	257	6	1857
Control System After Evolution	42	2	171	2	1133
Total Before Evolution	58	4	316	5	2288
Total After Evolution	78	5	428	8	2990
Difference	+20	+1	+112	+3	+702

Table 1: OO metrics before and after evolution.

From these general metrics we can conclude that the program evolved and grew in size. Table 2 illustrates what impact the various separate scenarios had on the code and how they affected the program structure. To be able to calculate these metrics the code version before evolution is compared to the code version after evolution. Note: with "*new operations*" new operations in existing components are meant, new operations in new components are not counted.

Scenarios	#changed classes	#changed operations	#new operations	#new states	#new events
V1	11	3	11	1	1
V2	14	3	14	1	1
All voltage scenarios	14	3	14	1	1
A1	11	3	11	1	1
All current scenarios	11	3	11	1	1
P1	0	0	0	0	0
All power scenarios	0	0	0	0	0
E1	0	0	0	0	0
E2	15	4	15	2	2
E3	16	4	16	3	2
E4	16	4	16	4	3
All energy scenarios	16	4	16	4	3
All scenarios	16	4	28	5	5

Table 2: OO metrics by evolution scenario.

Naturally there are also general metrics to illustrate the difference between an old and a newer software version. Especially how many new code is added and deleted(LOC). Metrics for components (classes) and operations (methods) are also given. Note that the sum of deleted and new lines of code is different from the growth in LOC indicated by table 1. The reason for this is that the results of table 1 are extracted with the *Metrics 1.3.6* [36] tool which does count comment and empty lines also as LOC, whereas the metrics from table 3 are extracted with *RippleTool* [38], which does not count comment and empty lines as LOC.

Metric	#
New_Classes/Components	19
Changed_Classes/Components	17
New_Interfaces	1
New_Methods/Operations	112
Changed_Methods/Operations	2
New_Interface Methods	3
New_Methods/Operations existing component	28
New_Interface Methods existing interfaces	2
Changed_Methods/Operations	5
New/Changed LOC	706
Deleted LOC	27

Table 3: OO implementation change metrics (difference between before and after evolution)

With this information it becomes possible to calculate the concern diffusion metrics for the energy-awareness concern. The concern diffusion metric results can be seen in table 4. Without comparisons the result of this metric do not tell us much.

Metric	#
CDC	36
COC	130

Table 4: OO concern diffusion metrics for the energy-awareness concern

The metrics show only that a small portion of the new methods affects existing components, indicating a lot of new functionality could be added in a modular and decoupled way. Some LOC disappeared due to rewriting of the state space context in the planner components of the control system.

If these results are analysed in more detail, we find that most scenario's affect the control system architecture. Table 5 lists the components and which scenarios affect them. The components which are listed must be modified to implement the listed scenarios. Scenarios add new methods or change existing methods to existing classes.

Module	Scenarios
Monitor interface + concrete Monitors	V1, V2, A1, E2, E3, E4
Analyzer interface + concrete Analyzers	V1, V2, A1, E2, E3, E4
Planner interface + concrete Planners	V1, V2, A1, E2, E3, E4
Visitor interface + concrete Executors	V2, E2, E3, E4
MyArbitrator	V2
SuperPlanner (concrete Planner)	A1
PhysicalConcept	E2, E3, E4
Navigator	E3, E4

Table 5: Impact scenarios on components

After looking at the results, the question rises if these problems could have been avoided, or if different design choices would have resulted in a different outcome. Therefore the control loop architecture should be examined. Especially the problems related to both interfaces to the base system; the observer and visitor pattern. Both the introduction of new events and adaptations boil down to the introduction of new object types; new handlers must be created to handle the new data types. Given the assumption to maximise reuse within the design, there is one other option to resort to: casting. Casting gives the programmer the option to handle data in a uniform way regardless the format. Before applying operations on the data, a cast should be made to cast the data back to the original format. Since casting is static, casting might fail on runtime if the object is casted to a type which is not the object type or a supertype of the object to be cast. Methods with a general parameter are more flexible but eventually the data must be extracted to analyse them. If we take the control system architecture as example, it is possible to generalise the monitor data. It is not until the data arrives at the analyser that a cast is required. If this principle is applied only the planner component will deal with the introduction of new events. However, as said, casting is deemed unsafe and encourages runtime errors. Handling data in uniform way neglects all the features the object model gives us; in a sense by using such a mechanism the compiler is extended and told forcefully what to do. One could go even further and store every object in a bitmap, and write a custom interpreter to be as flexible as possible completely ignoring the object model. Considering this, the conclusion would be OO cannot solve the problems introduced by energy evolution scenarios. Other options should be considered to intrude the energy aspect to an existing software program. Energy crosscuts the entire control system architecture and is tangled in most of its components after evolution.

5.2.2 The Impact of Energy-awareness Functionality to the AO Design

The general metrics of the AO implementation are given in table 6. Metrics are calculated before and after evolution; base program and control system metrics are separated.

	# of operations	# of interfaces	# of methods	# of interface methods	LOC
Base System Before Evolution	38	0	264	0	1867
Control System Before Evolution	26	5	78	4	533
Base System After Evolution	38	0	264	0	1867
Control System After Evolution	65	6	207	5	1343
Total Before Evolution	64	5	342	4	2400
Total After Evolution	103	6	471	5	3210
Difference	+39	+1	+129	+1	+810

Table 6: AO metrics before and after evolution.

As with the OO implementation, these general metrics give no information regarding ripples or changed components or methods. However if we look in detail to the base system before and after evolution, it can be seen it has not changed. This is a first indication that evolution impact is reduced. Again it can be concluded that the program evolved and grew in size.

Let us have a look at what impact the various separate scenarios had on the code. Table 7 illustrates the impact of various scenarios on the code. To be able to calculate these metrics the code version before evolution is compared with the code version after evolution.

Scenarios	#changed components	#changed operations	#new operations	#new states	#new events
V1	2	2	0	1	1
V2	2	2	0	1	1
All voltage scenarios	2	2	0	1	1
A1	2	2	0	1	1
All current scenarios	2	2	0	1	1
P1	0	0	0	0	0
All power scenarios	0	0	0	0	0
E1	0	0	0	0	0
E2	2	2	0	2	2
E3	2	2	0	3	2
E4	2	2	0	4	3
All energy scenarios	2	2	0	4	3
All scenarios	4	4	0	5	5

Table 7: AO metrics by evolution scenario.

Naturally there are also general metrics to illustrate the difference between the old and newer version of the software. Especially how many new code is added and deleted (LOC). Metrics for components (classes and aspects) and operations (methods and advices) are given in table 8. Again, the results of table 6 are extracted with the *Metrics 1.3.6* [36] tool which does count comment and empty lines also as LOC, whereas the metrics from table 8 are extracted with *RippleTool* [38], which does not count comment and empty lines as LOC.

Metric	#
New_Classes	19
New_Aspects	20
New_Components_Total (OO+AO)	39
Changed_Components_Total (OO+AO)	4
New_Methods_OO	77
New_Methods existing components_Total (OO+AO)	0
New_Point-cuts_AO	6
New_Methods/Advices_AO	24
New_Operations_Total (OO+AO)	101
Changed_Operations_Total (OO+AO)	4
New LOC_OO	531
New LOC_AO	172
New LOC_Total (OO+AO)	703
Deleted LOC_OO	20
Deleted LOC_AO	0
Deleted LOC_Total (OO+AO)	20

Table 8: AO implementation metrics.

With these information we can calculate the concern diffusion metrics for the energy-awareness concern as illustrated in table 9. Like we said before, these metrics do not say much unless compared with another implementation.

Metric	#
CDC	43
COC	105

Table 9: AO concern diffusion metrics for the energy-awareness concern

The high amount of new aspects can be explained by the fact that every concrete control system component has its own concrete aspect module. Aspects extend the existing OO object with new functionality like new event or adaptation handlers. Some lines of code disappeared; the reason for this is that the state space context in the planner components of the control system architecture had to be rewritten.

5.3 Evaluation

The metrics have already been given for the OO and AO implementation. However, without comparison metrics are useless. Although one can come up with the best possible design/implementation, it is not proven until measurements are extracted and compared to measurement data of other implementations.

5.3.1 Comparison

This section contains a comparison of both implementation before and after evolution. The implementations are compared on general metrics, concern diffusion metrics and finally a SAAM analysis.

Before Evolution Table 10(left) shows all generic metrics results for both the OO and AO implementation. Base system and control system code are distinguished for components, operations and LOC.

Metric	Before Evolution		After Evolution	
	OO	AO	OO	AO
COMPONENTS Base System	36	38	36	38
COMPONENTS Control System	22	26	42	65
COMPONENTS Total	58	64	78	103
OPERATIONS Base System	255	264	257	264
OPERATIONS Control System	61	78	171	207
OPERATIONS Total	316	342	428	471
LOC Base System	1841	1867	1857	1867
LOC Control System	447	533	1133	1343
LOC Total	2288	2400	2990	3210

Table 10: OO and AO implementation compared before and after evolution

We see that the AO implementation has slightly more components before evolution, caused by the fact that aspects extend existing classes. A class is no longer modified when introducing new functionality; instead new functionality is introduced in new components. We see that the number of methods is constant in the base system for the OO and AO implementation where one would think the number of methods should decrease in the AO implementation due to the removed components. This is caused by the fact that the AO implementation did not run on the robot; it needs some simulation code to run on a standard java platform. Table 11 shows the simulation code metrics. To get the results without the simulation code, the general metrics of the base system before and after evolution must be subtracted from the simulation code results.

Metric	**#**
Classes	1
Methods	15
LOC	79

Table 11: AO simulation code metrics

The AO point-cut mechanism increases the decoupling between the control system and the base system. Components can be observed by intercepting the execution flow of the update methods; a return value can simply be intercepted. The LOC are more or less the same for both implementations; the only difference lies in the control system code; the aspect version contains slightly more code due to the aspect code being more elaborate.

After Evolution Table 10(right) shows all generic metrics results for both the OO and AO implementation after evolution. It can be seen that the amount of components and operations in the control system increased drastically after evolution. Especially in the AO implementation there are a lot of new components; mainly caused by the fact that the AO implementation hardly contains rewritten components after evolution. Existing components are extended with aspects to encode new functionality. Each aspect counts as a new component resulting in a high amount of components for the AO control system implementation. The results show clearly that the base system is not affected by evolution in the AO implementation. The general conclusion is the AO implementation needs more code to implement the functionality introduced by evolution.

If we compare the energy-awareness concern diffusion metrics (table 12), we see that the AO implementation needs more components and slightly less operations to implement the functionality introduced by evolution scenarios.

Metric	*OO*	*AO*
CDC	36	43
CDC	130	105

Table 12: OO and AO implementation diffusion metrics compared

SAAM From the general metrics we concluded that the AO implementation needs more code to implement the energy functionality. However, how this code distributed over the components, and in what way it is introduced cannot be concluded from these metrics. They do not show if energy functionality affects existing components or if it can be introduced by new components.

Table 13 shows the results of the SAAM analysis for both the OO and the AO implementation. It can be seen that the AO implementation hardly contains changed components and operations. Only the introduction of new states leads to rewriting components and operations; this is the case for all scenarios except P1. The states are introduced in both the initial control loops and the super control loop resulting in three changed components. The respective state space contexts are rewritten to introduce the new states.

Scenarios	#changed classes		#changed operations		#new operations		#new states		#new events	
	OO	**AO**	OO	**AO**	OO	**AO**	OO	**AO**	OO	**AO**
V1	11	**2**	3	**2**	11	**0**	1	**1**	1	**1**
V2	14	**2**	3	**2**	14	**0**	1	**1**	1	**1**
All voltage scenarios	14	**2**	3	**2**	14	**0**	1	**1**	1	**1**
A1	11	**2**	3	**2**	11	**0**	1	**1**	1	**1**
All current scenarios	11	**1**	3	**1**	10	**0**	1	**1**	1	**1**
P1	0	**0**	0	**0**	0	**0**	0	**0**	0	**0**
All power scenarios	0	**0**	0	**0**	0	**0**	0	**0**	0	**0**
E1	0	**0**	0	**0**	0	**0**	0	**0**	0	**0**
E2	15	**2**	4	**2**	15	**0**	2	**2**	2	**2**
E3	16	**2**	4	**2**	16	**0**	3	**3**	2	**2**
E4	16	**2**	4	**2**	16	**0**	4	**4**	3	**3**
All energy scenarios	16	**1**	4	**2**	16	**0**	4	**4**	3	**3**
All scenarios	16	**4**	4	**4**	28	**0**	5	**5**	5	**5**

Table 13: OO and AO metrics by evolution scenario.

The new operation column in table 13 shows the AO implementation does not introduce a single new operation to existing components after evolution. In terms of changed operations and components the AO implementation is superior to the OO implementation; despite the greater amount of LOC, operations and components. The AO code extends components without rewriting, creating a looser coupling between the initial code (*legacy code*) and code introduced after evolution.

5.3.2 Interpretation and Discussion

The introduction of the energy-awareness concern to the OO implementation resulted in a lot of design ripples. The introduction of new events and adaptations had a big impact on the control system architecture. The monitor, analyser and planner components needed to be extended with new event handlers. This resulted in interface changes; every concrete component implementing the interfaces also suffered since they had to make the new abstract interface methods concrete. New adaptations handlers affected the Visitor pattern as well as the State pattern. New adaptations led to new or changed system states i.e. state space evolution. AO allows the dynamic extension of interfaces; new concrete components can implement the interfaces by the declare mechanism. New concrete components can be extended by concrete aspects to implement new interface methods introduced through evolution. This solves the problems related to the introduction of new events and adaptations. However, state space evolution remains a problem.

The introduction of new events to the state context is solved by the dynamic extension and declare mechanisms. Introducing new states on the other hand is still a problem, new states lead to rewriting the existing context code. Overriding the execution of the context with a point-cut is possible but will lead to the stacking of advices on top of each other after each new evolution version of the program.

5.3.3 Conclusion

The measurement results look quite typical for an OO and an AO implementation of the same program; the AO implementation adds new functionality through new code whereas the OO implementation must implement new functionality through modifying existing components. The higher amount of code in the AO implementation can be explained by the fact the AO code contains code to intercept join-points to which the advices will be applied. In general, AO uses more code to implement new functionality; however, the newly introduced code is more decoupled from the existing implementation. Where in the OO implementation the addition of new functionality results in the rewriting of existing components and methods, AO introduces new functionality through new components, dynamically extending existing component by aspects. This information cannot be extracted from the general and concern diffusion metrics results. Only after applying the scenario-based analysis technique SAAM, we found that AO introduced new functionality in a decoupled way. Various minor metrics illustrated the impact for each separate scenario. The results showed that the AO implementation performed better than the OO implementation with respect to rewriting code. SAAM also illustrated the energy-awareness functionality crosscuts various other concern. A closer look at the new and changed components revealed that the concern was scattered throughout the implementation. From the actual measuring of energy within the base system; to the new adaptations handlers introduced to optimise energy usage. Even though AO succeeded into extracting energy-awareness functionality from the base system implementation, energy remains scattered across various separate aspects. As said, AO can not solve the state space evolution problem. Other approaches, like context oriented programming and event-based modularisation techniques, should be considered when attempting to solve this problem. Especially event-based modularisation may provide a solution for this problem, since events form the coupling between states and have a big impact on the state space context.

6 Beyond Objects

An object oriented or aspect oriented approach are not the only options to implement control systems. As presented in the background chapter, process control is closely related; Shaw [41] presents a software design paradigm based on process control. She argues that: "Unlike object oriented or functional design, which are characterised by the kinds of components that appear, control system designs are characterised both by the kinds of components and the special relations that must hold among the components."

Event-based modularisation techniques can provide a solution. In this chapter we will give an overview of known event-based extensions for OO programming. Additionally we will evaluate two event-based modularisation techniques (Esper [42] and EventReactor [43]) in detail. We evaluate where these techniques can help to increase the decoupling and modularisation of the control system design. Esper is an established OO extension which provides a solution for complex event processing; events can be filtered by event queries, providing a way to process large quantities of events. EventReactor is a experimental project which uses events as a basis for modularisation. EventReactor allows the declaration of event modules; event modules are decoupled modules which are triggered by input events. The modules can perform adaptation actions and produce output events to trigger other event modules.

We will give a brief overview of existing event-based modularisation techniques for various languages. We will discuss Esper and EvenReactor in detail and point out where our design can benefit from these techniques.

6.1 Existing Event-Based Modularisation Techniques

Event-based modularisation [44] techniques especially focus on the interaction and relation between components through events. The state space evolution problem we identified could possibly be solved by applying event-based modularisation techniques. There exist established event-based extensions for OO programming; an overview is given Malakuti and Aksit [44]. They give the event-delegate mechanism of the C# programming language as example. Escala an extension to the Scala programming language, Scala is both functional and object oriented, is given as another example. They also address the Ptolemy programming language which is used to implement crosscutting concerns in Java programs. Although all these mechanisms/languages allow the use of events in OO programming languages they have shortcomings. The first two techniques suffer from the same problems as the AO implementation. One of these problems is the lack of DSL support to implement the reactive (control) system. The implementation of constraints among multiple reactive parts may scatter across and tangle with the reactive system, which may introduce reuse anomalies. The event selection predicates/handlers have limited expressive power to select the event calls of interest. This forces the programmer to define complex selection semantics in the reactive part, which increases the complexity of implementation and reduces its reusability. Ptolemy suffers from limited expressive power to select the events of interest, queries can only be expressed over event types. Another potential problem arises if complex binding semantics are needed. These must be expressed as a part of the handler method which reduces the reusability of these methods. In Ptolemy handler methods and point-cut expressions are defined within one class. The reusability of handlers for different events is reduced by such a tight coupling of handler methods to the specification of events of interest. Another problem is that the necessary interaction constraints must be programmed as part of handler methods. In the case different constraints are needed due to evolution requirements this may reduce the reusability of handler methods.

These techniques do not solve the evolution ripples present in the control system. The Observer and Visitor pattern still suffer from the introduction of new event types and adaptation types.

6.2 Esper

Esper [42] is an OO extension designed for complex event processing (CEP). This includes event series analysis, and is available for Java and .NET.

6.2.1 Overview

Esper works more or less like a database turned upside down. Queries can be defined before the data is present. Like a filter, data that meets the requirements specified by the query is filtered out and can be processed in more detail, saving computation/processing time. A normal database on the other hand stores data; queries can be executed on the database to acquire a set of data, whose members share one or more specific properties.

Esper is designed for applications that must process events (messages) in real-time or near real-time. According to the Esper manual [42] Esper is especially suited for application that deal with high quantities of events:

– This is sometimes referred to as complex event processing (CEP) and event series analysis. Key considerations for these types of applications are throughput, latency and the complexity of the logic required.

- High throughput - applications that process large volumes of messages (between 1,000 to 100k messages per second)
- Low latency - applications that react in real-time to conditions that occur (from a few milliseconds to a few seconds)
- Complex computations - applications that detect patterns among events (event correlation), filter events, aggregate time or length windows of events, join event series, trigger based on absence of events etc.

The Esper engine was designed to make it easier to build and extend CEP applications.–

[Esper Reference V5.0.0]

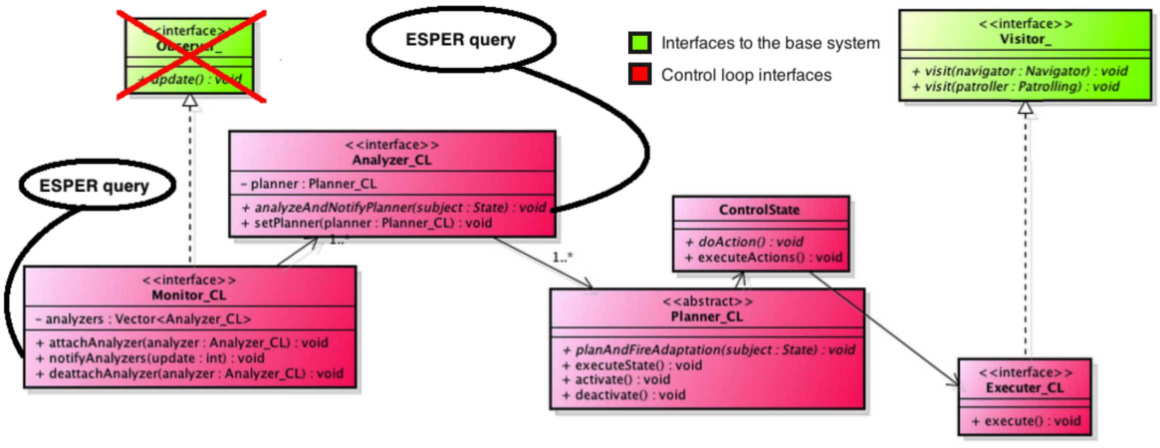
6.2.2 Applicability Case Study

Esper can be used to make the Observer pattern obsolete. Events can simply be fed to the Esper engine and queries can be defined to select them where needed. Components in the control loop architecture where Esper can be applicable are the monitor and possibly the analyser components. The monitor must define an Esper query to select all relevant events. These events should be passed down to the appropriate handlers; the handlers pass the events to the appropriate analysers. The analyser evaluates the events and determines if they require further action. Esper can be applicable in the analyser component if large sets of events are passed down from the monitor component. The analyser could define a more specific query to filter out events that require adaptation. However, robot control systems have quite a different event / latency profile then Esper is made for, for the number of events that need to be analysed is too limited. Even under extreme circumstances 50 events per second would be considered as a lot. Assuming the system processes incoming events four or five times a second the set of events selected by the monitor will be smaller than five and probably even one in some cases.

Of course it is possible to make the monitor interval longer but by doing so adaptations will be applied too late. Events are only useful for a certain amount of time; for our patrol robot an interval of a second would already be too long. Although Esper can be applied it must be noted that Esper is designed to process high quantities of events. The event quantities we deal with in our case study do not even come near the quantities (1K to 100K per second) Esper is designed for.

Although Esper can make the observer interface components obsolete, ripples caused by the introduction of new events will still be present in the control loop architecture. The main problems were caused by the introduction of new events to the monitor, analyser and planner components. These components need new methods/handlers to handle new events.

Figure 17 illustrates where Esper can be applied in the existing architecture. We see the observer interface becomes obsolete and the Esper queries are defined in the monitor and analyser components.



<<interface>>
Observer_
+ update() : void
ESPER query
🟩 Interfaces to the base system
🟥 Control loop interfaces
<<interface>>
Visitor_
+ visit(navigator : Navigator) : void
+ visit(patroller : Patrolling) : void
<<interface>>
Analyzer_CL
- planner : Planner_CL
+ analyzeAndNotifyPlanner(subject : State) : void
+ setPlanner(planner : Planner_CL) : void
ESPER query
ControlState
+ doAction() : void
+ executeActions() : void
1..*
1..*
<<interface>>
Monitor_CL
- analyzers : Vector<Analyzer_CL>
+ attachAnalyzer(analyzer : Analyzer_CL) : void
+ notifyAnalyzers(update : int) : void
+ deattachAnalyzer(analyzer : Analyzer_CL) : void
<<abstract>>
Planner_CL
+ planAndFireAdaptation(subject : State) : void
+ executeState() : void
+ activate() : void
+ deactivate() : void
<<interface>>
Executer_CL
+ execute() : void

Figure 17: Security planner state space

Esper can be used to spot trends and predict emerging behaviour to some extent. A sequence of events may be used to extract information that can be used to detect and fire new events. How fast energy is drained from the battery, for example. If the robot uses a lot of energy in a short window of time, this could mean that something is wrong with the system. This phenomenon can be detected by observing multiple events of the same type for a certain amount of time. If a trend is detected, an event could be fired to notify the system something is happening. However, this does not necessarily mean something is off; it could also mean the robot just arrived on a place in the environment where there are a lot of obstacles, corners or doors to check. Nevertheless this mechanism can be used for future prediction. For this information to be useful we should define complex energy monitoring algorithms which excel in power saving and optimisation. Unfortunately these algorithms are outside the scope of this research.

Aside from applicability, the problem that events must be published from within the base system remains. The introduction of each new event still requires a modification of the base program. Each event has to be fired from within a base system class. The method that determines if the event must be fired must be modified; this is illustrated in figure 18. Since Esper makes the Observer pattern obsolete the base system does not longer need a subject interface for observable classes (classes that fire events).

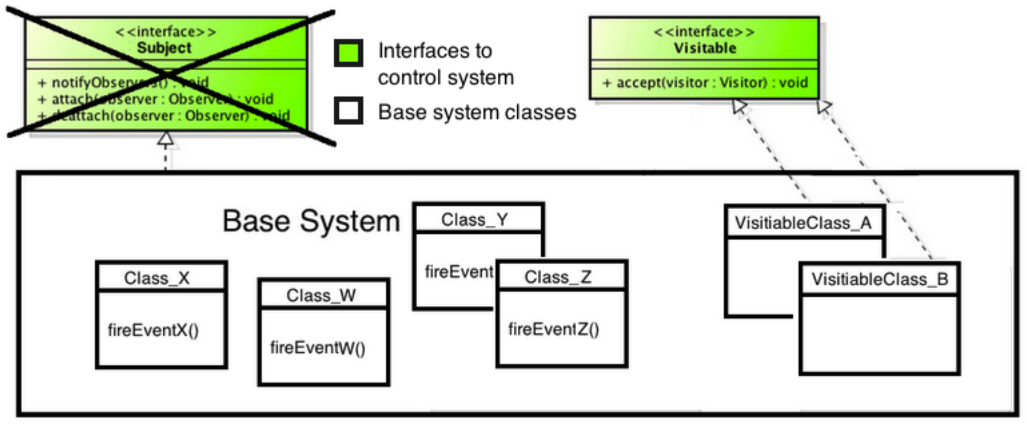


Figure 18: Security planner state space

6.2.3 Conclusion

Esper is useful for applications that deal with high quantities of events. SQL-like queries can be defined to filter out relevant events that need to be processed in more detail. Our patrolbot case study does not deal with high quantities of events. Although Esper can be applied, it will not solve the evolution problems present in the OO and the AO implementation. The problems related to the introduction of new events were:

- Ripples caused by introduction of new events (new data types need new handlers)
- Base system code must be changed to fire new events. Events cannot be extracted from the base system with a point-cut-like mechanism as is possible in AO; they need to be fired from within the base system code.

The overall conclusion is that Esper cannot help us to solve any of the evolution problems we encountered in the OO implementation. If we decide to focus on future research about complex energy optimisation algorithms, Esper should be re-evaluated. Especially applications that deal with high quantities of events can profit from Espers event processing mechanism. However, in our case study the amount of events that must be processed is too few in number to fully utilise Espers event processing power.

6.3 EventReactor

Malakuti and Wilke [45] designed the GreenDev framework to extend existing applications with energy-awareness functionality through event modules.

To effectively extend legacy applications with energy-awareness functionality, dedicated modularisation mechanisms are required. The GreenDev framework integrates energy testing and event-based modularisation for this matter. Energy testing helps identifying the energy-related interfaces of applications to the energy-awareness functionality, where event-based modularisation helps to modularise and decouple this functionality from the base functionality of the applications.

In GreenDev there is no need to manually inspect base applications to identify energy join points and activate these join points. Instead, the specification of energy-related events, can be defined transparently from the actual implementation of the base application. Therefore, structural changes in the application, e.g. changes in method names and class hierarchies, do not influence the event module specifications. In this chapter we will evaluate the EventReactor language presented in their paper.

6.3.1 Overview

EventReactor [43] is a event-based modularisation technique; events modules are coupled by input and output events. A module can be activated or triggered by one or more events; a module can apply an operation or adaptation as response. This response or adaptation can publish a new event if needed (output event). An output event can be a trigger for another event module. An event reactor module is depicted in figure 19. It shows how reactor chains link events to reactor modules, which contain action classes to apply adaptations or fire new events. The EventReactor engine captures all events fired from within the base system. Events are published to the EventReactor engine; event modules define a Prolog query to specify the activation or selection criteria to activate the module. Events contain various distinct properties; the selector criteria can select events based on the value of such properties.

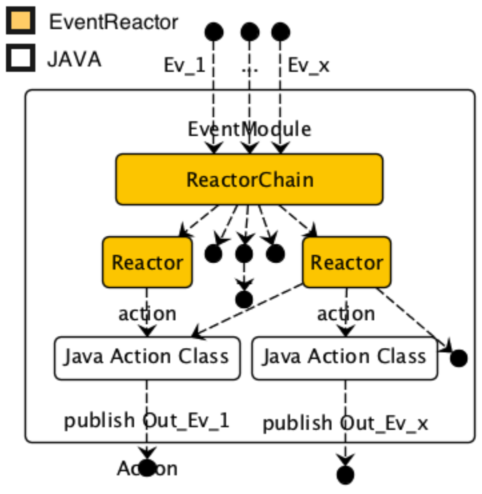


Figure 19: EventReactor overview

The use of input and output events creates a very loose coupling between event modules. If a module must be activated by a new event, the selector (input interface) query must be modified. New reactors to existing events can be added to the reactor chain; a reactor chain links a reactor to the events selected by the selector query. Reactors define actions/adaptations in the form of actions classes. Actions classes are java objects that apply adaptations at runtime to the base system.

6.3.2 Event Reactor MAPE-K Implementation

In order to implement MAPE-K in EventReactor we need various reactor chains and reactor types. Reactors are concrete instantiations of reactor-types. There are two main alternatives to implement MAPE-K loops in event reactor. All MAPE-K components can be implemented in one reactor module or all modules can be implemented in separate reactor modules.

Single Control loop Reactor Module In figure 20 a event module design is presented which implements all MAPE-K components in one single module. The module is triggered by one or more base events. Each module can fire synchronisation events to notify other control reactors. A module can define multiple action classes to apply adaptations to the base system.

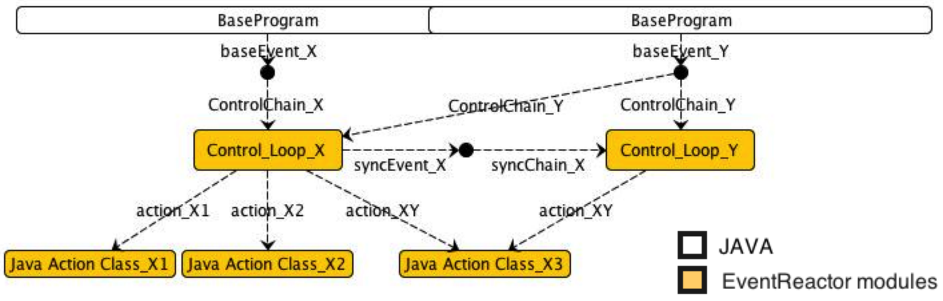


Figure 20: EventReactor MAPE-K design

When taking the reuse requirements into account, one immediately notices that this approach does not perform well concerning reusable modules. Especially the monitor and analyser functionality must be recoded in every new control loop module.

Separate Reactor Modules Control Loop components Given the event-based approach of EventReactor, separate modules for each MAPE-K component coupled by events are a more logical approach. An overview of the MAPE-K implementation in separate reactor modules is given in figure 21. Every reactor is loosely coupled to other reactors further in the chain.

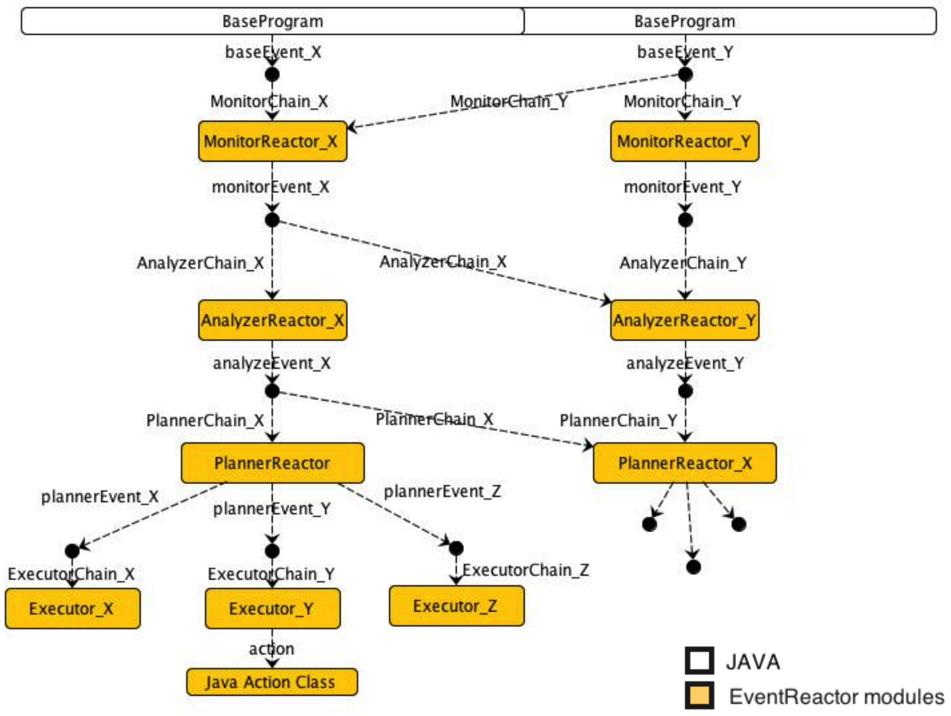


Figure 21: EventReactor MAPE-K design

Monitor (Reactor) The monitor must intercept instances of a specific type of event from the base system. The base event is linked to the monitor module by a reactor chain. Each event type needs its own monitor. The sole function of the monitor is to deliver/link the base event to the appropriate analyser. Various interested reactors (analysers) can react to the output event produced by the monitor module.

Analyser (Reactor) +Knowledge Base Analyser reactor chains link one or more monitor events to the analyser reactor. The analyser component analyse/determines if the event needs additional processing. To determine if an event needs additional processing the data-field of the event must be compared to some kind of knowledge base. This knowledge base can be hardcoded into the analyser component; if the system must be dynamically configurable, values could be stored in a configuration file and read from there. In this case the event indicates that the system needs to adapt(for example power levels become critical); the analyser component must fire a new planner output event.

Planner (Reactor) The planner component keeps track of the current system state and adapts this state depending on the events that reach this component. The events that reach/trigger this component are analyser output events. Depending on the system state the planner will fire adaptation/executor-events. Executor events indicate what kind of adaptations must be applied.

Executor (Reactor) The adaptation events fired by the planner reactors are linked to the executor reactors by executor chains. The executor reactors must have concrete action classes whose purpose is to apply adaptations in the base system. The action classes are java classes which can apply adaptations to the base system.

The separation of MAPE-K modules results in a loosely coupled control system where modules interact by events. Since modules are unaware of other modules but only of events, reuse becomes easy. If a new or existing module is interested in a certain output event produced by another module, it can be defined in the selector query present in the input interface of the respective module. Other modules further in the chain do not suffer from modifications unless the output event of the module is altered.

6.3.3 Applicability Case Study

Our case study contained various control loops which can be modelled using event modules. If we apply the second approach to our case study the control system design will contain the modules depicted in figure 22. White modules are present in the initial design which was energy-unaware. Red modules are the modules introduced by the energy evolution scenarios. Modules communicate through events and do not follow fixed interfaces.

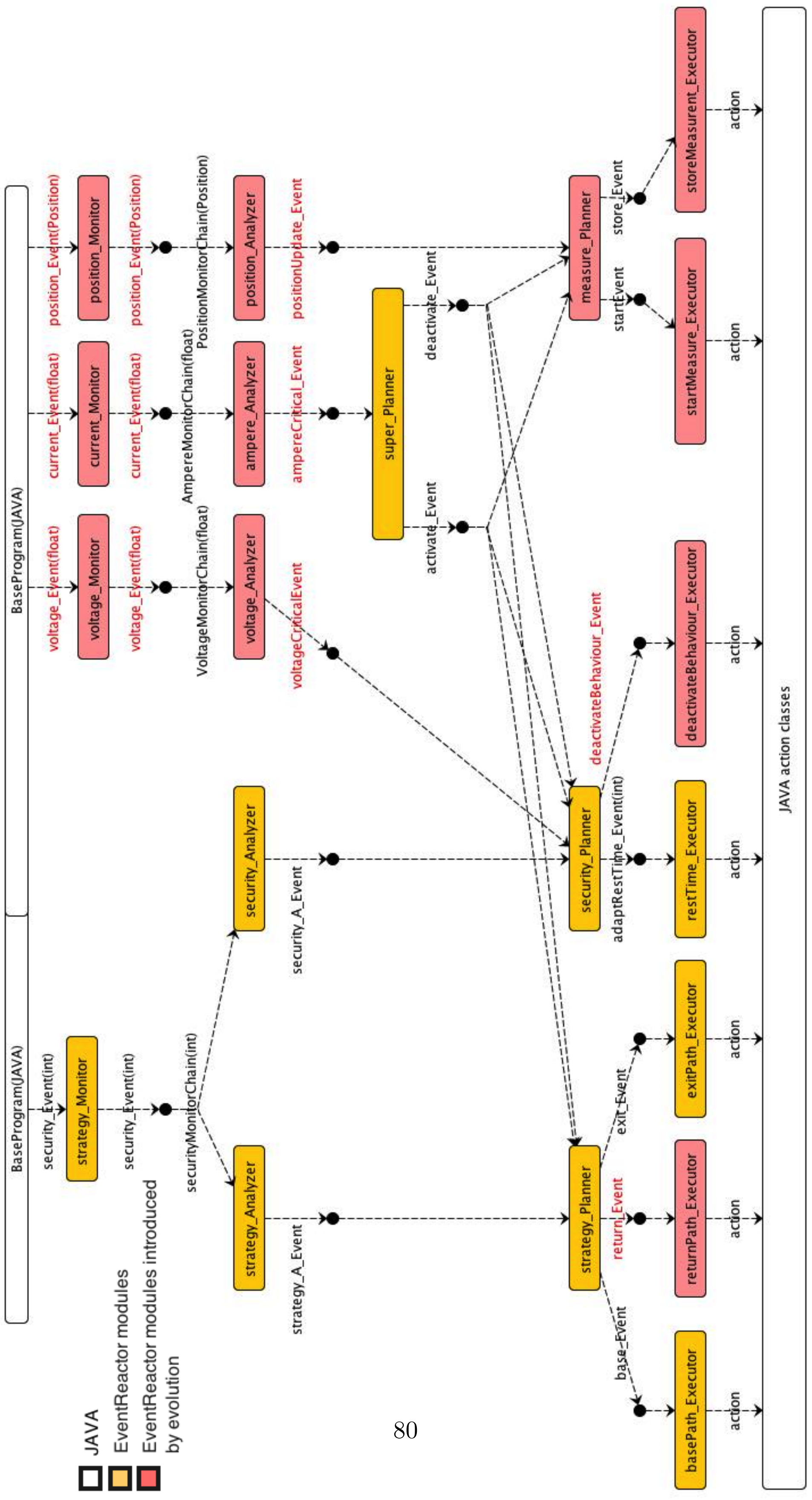


Figure 22: EventReactor design (red components and events are introduced after evolution)

It can be seen that new event modules are coupled to events. Existing modules can react to new events if their selector query is modified, as illustrated in listing 10. For each new event a selector identifier must be added, which can be used to modify the selector query. The new event can be added to the existing list of events in the selector query which triggers the respective reactor chain.

Listing 10: Triggering existing modules with new events

```
eventpackage ExistingConcern_X{
        selectors
                existingEvent = { E| isEvent(E),
                    hasAttribute(E, 'name', 'existingE')
                    hasAttribute(E, 'publishername', 'SomeExistingConcern.Module_Y')
                }
                newEvent = { E| isEvent(E),
                    hasAttribute(E, 'name', 'newE')
                    hasAttribute(E, 'publishername', 'SomeExistingConcern.Module_Z')
                }
        eventmodules
                Module_X := {existingEvent, newEvent} {"publisher"} <-
                ExistingReactorChain() ->{};
}
```

Listing 11 illustrates how existing reactor modules can be modified when existing action classes must fire new events. Naturally the action class defining the adaptation action must be modified as well. A new method which fires the new event must be added to the existing actions class.

Listing 11: Extending existing action classes with new events

```
reactortype Reactor {
        action = AdaptationClass;
        events = {Event_X, newEvent_X;}
}
```

This event-based approach results in separate modules loosely coupled to each other. New modules can easily be added without much impact.

6.3.4 Conclusion

Unfortunately EventReactor is still under development, resulting in limited implementation of all the possibilities. For this reason this research does not cover a complete event reactor implementation for the control loop architecture. However, EventReactor can solve problems that emerge when a control loop architecture evolves. Extension with new events becomes easy, new events can be defined and dynamically added to the selector query of the interested reactor-chains.

Defining additional adaptation components and the coupling of these components to the existing base system is handled by reactor modules. A new reactor can be defined and added/coupled to the list of already existing reactors. The reactor can define an action to apply the actual adaptation. However, to apply the adaptation to the base system some kind of mechanism must be used. Right now the current EventReactor implementation does not contain a point-cut-like mechanism as defined in AO programming. The execution of an existing base system method cannot be intercepted to add advice code in which an event could be fired. Since EventReactor cannot use such a mechanism, it should resort to the Visitor design pattern used in the OO implementation. The lack of a point-cut-like mechanism also affects the base system once a new event is introduced. Events must be fired from within the base system code and published to the EventReactor engine. Resulting in modifications to the base system once a new event is introduced. Due to the event-based approach state spaces are formed by various separate modules. Instead of one module defining the context of the states, each module/state defines its own coupling to other states/modules. This allows a more dynamic extension of existing state spaces.

The current EventReactor implementation is limited in its expressiveness. In order to improve the expressiveness of EventReactor regarding state space models a domain-specific language extension should be considered. This domain-specific language should allow the definition of a state space in separate super modules. Now it is not possible to define state space in modules a state space must be built out of numerous separate modules. This results in a cluttered implementation in which it is hard to tell which modules belong to the state space and which do not.

6.4 Conclusion

Event-based modularisation solutions provide a different view on modularisation. However, most solution are extensions to existing techniques. This limits the expressiveness and possibilities since most solution are bound to one language. EventReactor is the only technique we evaluated which tries to abstract from language level limitations. Despite these shortcomings event-based modularisation should be considered more thoroughly in future research regarding CPS design.

7 Conclusion

This research tries to illustrate how energy-awareness functionality affects the modularity of an existing CPS design. We prepared an initial design as best as possible for future evolution. A domain analysis was conducted to extract energy-awareness evolution scenarios. After implementing the energy-awareness functionality in OO and AO programming, we conducted a series of measurements to find out if energy-awareness functionality indeed affects the modularity of the design.

At the beginning of this research two research questions were formulated which we will try to answer here:

- What are the effects on software modularity that follow from the introduction of the energy concern to Cyber-Physical Systems?

The effects of adding energy-awareness to an existing system differ for both implementations. Where the OO implementation did not cope well with the introduction of new events and states to apply adaptations, the AO implementation dynamically extends components by new event handlers and adaptation handlers by means of aspects. Unfortunately, state space evolution remained a problem that could not be solved by aspects, given the constraints we placed on the design. These constraints were about maximising reuse; reuse is not possible without performing casts. Casts were deemed undesired because they ignore the core principle of the object model; objects which no longer have a type and cannot be used until the compiler is told how to handle these objects.

- Can the energy-awareness concern be introduced to an existing Cyber-Physical System without crosscutting existing concerns?

The measurement results supported the claim that energy-awareness functionality is a crosscutting concern. Energy-awareness functionality is present in many classes and aspects. The conclusion is that energy-awareness functionality can be decoupled from most components but it cannot be modularised into a single component.

We evaluated event-based modularisation techniques to illustrate where our CPS design can benefit from these new and sometimes experimental techniques. EventReactor looks the most promising event-based modularisation solution; despite its immaturity, EventReactor and its event module model may provide solutions for the state space evolution problem. However, at this moment EventReactors potential cannot be reflected in measurements. For example the events must be fired from within the base system, resulting in unwanted modifications to legacy code.

During this research we faced some problems related to language support to CPS. Although the robot could be programmed in JAVA, language support for the embedded JVM of the robot was limited.

For example, it was not possible to run the AO (AspectJ) implementation on the robot. AspectJ runs on all standard JAVA virtual machines [46]. However Lejos[25] runs on a modified embedded VM [26], older versions of Lejos run on a VM called tinyVM [47]which is a subset of the standard JVM. TinyVM has some limitations and can only load a limited amount of classes.

Not all linguistic JAVA statements are supported; TinyVM does not support switch statements for instance. More recent Lejos versions are based on JAVA embedded SE [26]. Unfortunately, documentation is incomplete and scattered around the fora. The fora indicate that still not all linguistic JAVA statements are supported. Without complete documentation one can assume that similar limitations as the ones for TinyVM prevent AspectJ-support for the Lejos platform.

Similar problems prevent EventReactor from running on the Lejos platform. For example, EventReactor has many third party dependencies which must be loaded before EventReactor can run.

7.1 Future Work

This research made a first attempt to illustrate the effects of introducing energy-awareness functionality to an existing CPS design, however, it is too early for explicit claims. Future research should be conducted on other types of CPS to be able to generalise our findings. The robot system is quite limited with respect to the amount of events per second; the scalability of the proposed design should be investigated. In systems that deal with a high amount of events per second, event processing techniques like Esper should be applied.

We investigated how event-based modularisation techniques can provide solutions for the problems identified in this research. We will provide some recommendations for event-based modularisation techniques and EventReactor in particular. A domain-specific language extension could improve the expressiveness regarding state space models. In the case of EventReactor state spaces must be defined in separate event modules instead of being grouped in a super module. This may result in a cluttered implementation in which it is hard to tell which modules belong to the state space and which not. In none of the evaluated techniques it was possible to extend the base system with new events without rewriting. The lack of an aspect-like point-cut mechanism to extract events from the base system is a topic which should be improved. For example, in the current EventReactor implementation events must be fired from within the base system, resulting in unwanted modifications to the base system. Despite the limitations of existing event-based modularisation techniques, event-based modularisation promises to be a useful asset for developing CPS in the future.

Another topic for future research is language support. AO and event-based modularisation techniques are optimised for usage on general purpose platforms. CPS platforms often are a subset of these platforms limiting the possibilities of using state-of-the-art techniques. When developing AO and event-based modularisation techniques this should be taken into account.

References

[1] R. De Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, T. Vogel, D. Weyns, L. Baresi, B. Becker, N. Bencomo, Y. Brun, B. Cukic, R. Desmarais, S. Dustdar, G. Engels, K. Geihs, K. M. G`oschka, A. Gorla, V. Grassi, P. Inverardi, G. Karsai, J. Kramer, A. Lopes, J. Magee, S. Malek, S. Mankovskii, R. Mirandola, J. Mylopoulos, O. Nierstrasz, M. Pezzè, C. Prehofer, W. Sch`afer, R. Schlichting, D. B. Smith, J. P. Sousa, L. Tahvildari, K. Wong, and J. Wuttke, "Software engineering for self-adaptive systems: A second research roadmap," in *Software Engineering for Self-Adaptive Systems II.* Springer, 2013, pp. 1–32.

[2] Y. Brun, G. D. M. Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw, "Engineering self-adaptive systems through feedback loops," in *Software engineering for self-adaptive systems.* Springer, 2009, pp. 48–70.

[3] Computing, Autonomic, "An architectural blueprint for autonomic computing," *IBM White Paper*, 2006.

[4] S. te Brinke, S. Malakuti, C. Bockisch, L. Bergmans, and M. Akşit, "A design method for modular energy-aware software," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing.* ACM, 2013, pp. 1180–1182.

[5] G. Salvaneschi, C. Ghezzi, and M. Pradella, "An analysis of language-level support for self-adaptive software," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 8, no. 2, p. 7, 2013.

[6] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. Cheng, "A taxonomy of compositional adaptation," *Rapport Technique numéroMSU-CSE-04-17*, 2004.

[7] J. Andersson, R. De Lemos, S. Malek, and D. Weyns, "Reflecting on self-adaptive software systems," in *Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS'09. ICSE Workshop on.* IEEE, 2009, pp. 38–47.

[8] C. Talcott, "Cyber-physical systems and events," in *Software-Intensive Systems and New Computing Paradigms.* Springer, 2008, pp. 101–115.

[9] E. A. Lee, "Cyber physical systems: Design challenges," in *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on.* IEEE, 2008, pp. 363–369.

[10] M. Eaddy, T. Zimmermann, K. D. Sherwood, V. Garg, G. C. Murphy, N. Nagappan, and A. V. Aho, "Do crosscutting concerns cause defects?" *Software Engineering, IEEE Transactions on*, vol. 34, no. 4, pp. 497–515, 2008.

[11] New Mexico Solar Energy Association, "Energy Concepts Primer." accessed: 2015-01-19. [Online]. Available: http://www.nmsea.org/Curriculum/Primer/energy_physics_primer.htm

[12] J. van Amerongen, *Dynamical systems for creative technology.* Controllab Products BV, 2010.

[13] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski, “Metrics and laws of software evolution-the nineties view,” in *Software Metrics Symposium, 1997. Proceedings., Fourth International.* IEEE, 1997, pp. 20–32.

[14] G. Xie, J. Chen, and I. Neamtiu, “Towards a better understanding of software evolution: An empirical study on open source software,” in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on Software Maintenance (ICSM 2009).* IEEE, 2009, pp. 51–60.

[15] M. Harsu, *A survey on domain engineering.* Citeseer, 2002.

[16] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, “Feature-oriented domain analysis (FODA) feasibility study,” DTIC Document, Tech. Rep., 1990.

[17] M. Simos, D. Creps, C. Klingler, L. Levine, and D. Allemang, “Organization domain modeling (ODM) guidebook version 2.0,” 1996.

[18] A. MacCormack, J. Rusnak, and C. Y. Baldwin, “The impact of component modularity on design evolution: Evidence from the software industry,” *Harvard Business School Technology & Operations Mgt. Unit Research Paper*, no. 08-038, 2007.

[19] J. Hannemann and G. Kiczales, “Design pattern implementation in Java and AspectJ,” in *ACM Sigplan Notices*, vol. 37, no. 11. ACM, 2002, pp. 161–173.

[20] E. Figueiredo, C. Sant’Anna, A. Garcia, T. T. Bartolomei, W. Cazzola, and A. Marchetto, “On the maintainability of aspect-oriented software: A concern-oriented measurement framework,” in *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on.* IEEE, 2008, pp. 183–192.

[21] C. Sant’Anna, A. Garcia, C. Chavez, C. Lucena, and A. Von Staa, “On the reuse and maintenance of aspect-oriented software: An assessment framework,” in *Proceedings of Brazilian Symposium on Software Engineering*, 2003, pp. 19–34.

[22] R. Kazman, G. Abowd, L. Bass, and P. Clements, “Scenario-based analysis of software architecture,” *Software, IEEE*, vol. 13, no. 6, pp. 47–55, 1996.

[23] K. Rajnish, A. K. Choudhary, and A. M. Agrawal, “Inheritance metrics for object-oriented design,” *International Journal of Computer Science & Infomation Technology*, vol. 2, no. 6, 2010.

[24] LEGO Mindstorms, “Ev3.” accessed: 2015-01-19. [Online]. Available: http://www.ev-3.net/en/archives/850

[25] Lejos, “Java for LEGO Mindstorms.” accessed: 2015-01-19. [Online]. Available: http://www.lejos.org

[26] Oracle, "Java for LEGO©Mindstorms EV3." accessed: 2015-01-19. [Online]. Available: http://www.oracle.com/technetwork/java/embedded/downloads/javase/javaseemeddedev3-1982511.html

[27] G. Oliveira, R. Silva, T. Lira, and L. P. Reis, "Environment mapping using the lego mindstorms nxt and lejos nxj," in *14th Portuguese Conference on Artificial Intelligende, EPIA 2009*, 2009, pp. 267–278.

[28] G. Casella, "An introduction to empirical bayes data analysis," *The American Statistician*, vol. 39, no. 2, pp. 83–87, 1985.

[29] Lejos, "Lejos api, java for lego mindstorms." accessed: 2015-01-19. [Online]. Available: http://www.lejos.org/nxt/pc/api/index.html

[30] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[31] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein *et al.*, *Introduction to algorithms.* MIT press Cambridge, 2001, vol. 2.

[32] D. Brugali, "Software abstractions for modeling robot mechanisms," in *Advanced intelligent mechatronics, 2007 IEEE/ASME international conference on.* IEEE, 2007, pp. 1–6.

[33] J. A. Bondy and U. S. R. Murty, *Graph theory with applications.* Macmillan London, 1976, vol. 6.

[34] J. Simpson, C. L. Jacobsen, and M. C. Jadud, "Mobile robot control," in *Communicating Process Architectures 2006: WoTUG-29: Proceedings of the 29th WoTUG Technical Meeting, 17-20 September 2006, Napier University, Edinburgh, Scotland*, vol. 64. IOS Press, 2006, p. 225.

[35] C. SantÁnna, A. Garcia, U. Kulesza, C. Lucena, and A. Von Staa, "Design patterns as aspects: A quantitative assessment," *Journal of the Brazilian Computer Society*, vol. 10, no. 2, pp. 42–55, 2004.

[36] Metrics 1.3.6, "Metrics 1.3.6 - Getting started." accessed: 2015-01-19. [Online]. Available: http://metrics.sourceforge.net/

[37] G. Canfora, L. Cerulo, and M. Di Penta, "Identifying changed source code lines from version repositories." in *MSR*, vol. 7, 2007, p. 14.

[38] A. Bouius and G. Bouma, "Rippletool: Finding the Difference!" accessed: 2015-01-19. [Online]. Available: www.linuxpower.gotdns.com:90/RippleTool

[39] Git, "Git distributed revision control and source code management." accessed: 2015-01-19. [Online]. Available: http://git-scm.com/

[40] GitStats, “GitStats - git history statistics generator.” accessed: 2015-01-19. [Online]. Available: http://gitstats.sourceforge.net/

[41] M. Shaw, “Beyond objects: A software design paradigm based on process control,” *ACM SIGSOFT Software Engineering Notes*, vol. 20, no. 1, pp. 27–38, 1995.

[42] EsperTech, “Esper Reference v5.0.0.” accessed: 2015-01-19. [Online]. Available: http://esper.codehaus.org/

[43] S. Malakuti, “The Eventreactor Language.” accessed: 2015-01-19. [Online]. Available: http://ftp3.ie.freebsd.org/pub/download.sourceforge.net/pub/sourceforge/e/ev/eventreactor/Documentation/EventReactor.pdf

[44] S. Malakuti and M. Aksit, “Event-based modularization of reactive systems,” in *Concurrent Objects and Beyond.* Springer, 2014, pp. 367–407.

[45] S. Malakuti and C. Wilke, “Energy aspects: modularizing energy-aware applications,” in *Proceedings of the 3rd International Workshop on Green and Sustainable Software.* ACM, 2014, pp. 23–30.

[46] A. Popovici, T. Gross, and G. Alonso, “Dynamic weaving for aspect-oriented programming,” in *Proceedings of the 1st international conference on Aspect-oriented software development.* ACM, 2002, pp. 141–147.

[47] TinyVM, “TinyVM.” accessed: 2015-01-19. [Online]. Available: http://tinyvm.sourceforge.net/