Dipl.-Ing. Johannes Iber, BSc

An Approach for Adding Resilience to Industrial Control Systems

DOCTORAL THESIS

to achieve the university degree of

Doktor der technischen Wissenschaften

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Dipl.-Inform. Dr.sc.ETH Kay Uwe Römer

Institute of Technical Informatics

Advisor
Dipl.-Ing. Dr. techn. Christian Kreiner

Graz, March 2019

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present doctoral thesis.

Date	Signature

Acknowledgments

This doctoral thesis has been carried out at the Institute of Technical Informatics at Graz University of Technology, in cooperation with the industrial partner Andritz Hydro GmbH in Vienna.

I deeply appreciated my mentor and advisor, Dipl.-Ing. Dr. techn. Christian Kreiner, for his guidance, discussions, and his warmly and sincere nature. Sadly, his life ended too early in the last year of my doctorate studies.

I thank Rudolf Neuner from Andritz Hydro GmbH. He patiently explained to me how things work in practice and was always open for my suggestions and ideas.

I thank my supervisor, Univ.-Prof. Dipl.-Inform. Dr.sc.ETH Kay Uwe Römer, for guiding me through the last phase of my doctoral thesis.

Furthermore, my thanks go to Dipl.-Ing. Dr. techn. habil Andreas Riel who kindly agreed to serve as a second adviser of my thesis.

I am very thankful to the love of my life, Mag.-pharm Theresa Holzer. Without her, I would not be the person I am today.

Special thanks go to all colleagues working with me in the same research project, also known as the "A-Team", consisting of Dipl.-Ing. Dr. techn. Andrea Höller, Dipl.-Ing. Dr. techn. Tobias Rauter, Dipl.-Ing. Michael Krisper, and Dipl.-Ing. Jürgen Dobaj. Each of them is an incredible person and it was a joy to spend time with them.

I am very grateful to my parents, Roswitha and Josef Iber. Without them nothing would have been possible and they always believe in me. I thank my sister Mag. iur. Andrea Iber for reminding me to finish my thesis.

I thank my friends Florian Gollowitsch, B.Ed. M.A., Christoph Kronawetter, B.Ed, and Armin Liesinger. I always had a good time with them during my studies.

Last but not least, I thank all my former colleagues at the university. Each of them broadened my technical horizon. We also had a lot of fun, which is equally important.

Graz, March 2019
Johannes Iber

Abstract

Industrial control systems are virtually everywhere in our society. They are used in industries such as electricity, manufacturing, transportation, chemistry or even the food industry. At the time of writing, these systems are becoming more technically advanced, versatile, more interconnected, and in certain areas, such as sensing devices, more heterogeneous.

This thesis has been carried out in the domain of industrial control systems for hydropower plants. We confirm the following hypothesis: *System knowledge enables automated resilience in industrial control systems.* System knowledge is information about the concrete configuration of an industrial control system. Resilience is the persistence of service delivery that can justifiably be trusted, when facing changes. We confirm this hypothesis with contributions targeting design and run time.

Regarding design time, we propose modeling languages for specifying the system configuration and non-functional properties. We are applying contract-based design for capturing the non-functional behavior of single components. The proposed concepts are tailored for capturing non-functional properties such as security, performance, or redundancy. We complement the design time contributions with design patterns for designing configurability into domain-specific modeling languages.

Regarding run time, we analyzed the potential of self-adaptive software systems for adding resilience in the context of hydropower plant units. The goal of such systems would be to defend the hardware/software stack below the control logic against hardware failures, security attacks, software bugs, misconfiguration, and faults in the physical environment. We contribute a decentralized hierarchical self-adaptive software system named Scari for lifting the identified potential. We developed a prototype implementation and experimented with it in distinct scenarios. A key element in this self-adaptive system is the reuse of design time information at run time and utilizing the system knowledge for adaptations. To the best of our knowledge, we are the first ones applying a self-adaptive software system for adding resilience to industrial control systems. Through our work on self-adaptive software systems, we encountered three design patterns grasping the trade-off between distributing data and information. Furthermore, we found a design pattern for separating processing and coordination in computer systems.

Kurzfassung

Industrielle Steuerungssysteme sind praktisch überall in unserer Gesellschaft zu finden. Sie werden in Branchen wie Elektrizität, Fertigung, Transport, Chemie oder sogar der Lebensmittelindustrie eingesetzt. Zum jetzigen Zeitpunkt werden diese Systeme technisch fortschrittlicher, vielseitiger, stärker vernetzt und in bestimmten Bereichen, wie z.B. bei Sensoren, heterogener.

Diese Arbeit wurde im Bereich der industriellen Steuerungssysteme für Wasserkraftwerke durchgeführt. Wir bestätigen die folgende Hypothese: *Systemwissen ermöglicht automatisierte Resilienz in industriellen Steuerungssystemen.* Systemwissen ist Information über die konkrete Konfiguration eines industriellen Steuerungssystems. Resilienz ist die Persistenz der Leistungserbringung, der man trotz Veränderungen weiterhin vertrauen kann. Wir bestätigen diese Hypothese mit Beiträgen die auf die Design- und Laufzeit abzielen.

In Bezug auf die Designzeit präsentieren wir Modellierungssprachen zur Spezifikation der Systemkonfiguration und nicht-funktionaler Eigenschaften. Wir verwenden Contract-Based Design zur Erfassung des nicht-funktionalen Verhaltens einzelner Komponenten. Die vorgeschlagenen Konzepte sind auf die Erfassung von nicht-funktionalen Eigenschaften wie Sicherheit, Performance oder Redundanz zugeschnitten. Wir ergänzen die Designzeitbeiträge durch Entwurfsmuster zur Gestaltung der Konfigurierbarkeit in domänenspezifischen Modellierungssprachen.

In Bezug auf die Laufzeit analysierten wir das Potenzial selbstadaptiver Softwaresysteme zur Erhöhung der Resilienz im Kontext von Wasserkraftwerkseinheiten. Das Ziel solcher Systeme wäre es, den Hardware/Software-Stack unterhalb der Steuerungslogik gegen Hardwareausfälle, Sicherheitsangriffe, Software-Bugs, Fehlkonfigurationen und Fehler in der physikalischen Umgebung zu schützen. Wir präsentieren ein dezentrales hierarchisches, selbstadaptives Softwaresystem namens Scari, um das identifizierte Potenzial zu heben. Wir entwickelten einen Prototypen und experimentierten mit diesem in verschiedenen Szenarien. Ein Schlüsselelement in diesem selbstadaptiven System ist die Wiederverwendung von Designzeitinformationen zur Laufzeit und die Nutzung des Systemwissens für automatisierte Anpassungen. Nach bestem Wissen und Gewissen sind wir die Ersten, die ein selbstadaptives Softwaresystem einsetzen um die Resilienz von industriellen Steuerungssystemen zu erhöhen. Durch unsere Arbeit an selbstadaptiven Softwaresystemen stießen wir auf drei Entwurfsmuster, die sich mit dem Kompromiss zwischen der Verteilung von Daten und Informationen befassen. Darüber hinaus haben wir ein Entwurfsmuster zur Trennung von Verarbeitung und Koordination in Computersystemen gefunden.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

ACPU Application CPU.

CCPU Communication CPU.

CPS Cyber Physical Systems.

D-Bus Desktop Bus.

DoS attack Denial-of-Service attack.

DSL Domain–Specific Language.

DSML Domain–Specific Modeling Language.

FUP Function Plan.

GPL General–Purpose Language.

GPML General–Purpose Modeling Language.

IoT Internet of Things.

MAPE-K Monitor Analyze Plan Execute - Knowledge.

MDE Model-Driven Engineering.

MDSE Model-Driven Software Engineering.

OMG Object Management Group.

OODA Observe Orient Decide Act.

PLC Programmable Logic Controller.

POU Program Organization Unit.

QEMU Quick Emulator.

SCADA Supervisory Control And Data Acquisition.

Scari Secure and Reliable Infrastructure.

TLS Transport Layer Security.

UML Unified Modeling Language.

1. Introduction

Industrial control systems are virtually everywhere in our society. They are used in industries such as electricity, manufacturing, transportation, chemistry or even the food industry. At the time of writing, these systems are becoming more technically advanced, versatile, more interconnected, and in certain areas, such as sensing devices, more heterogeneous.

This thesis is devoted to adding resilience to industrial control systems by leveraging system knowledge at design and run time. By doing so, an industrial control system continues to be reliable and secure despite system parts are failing or under attack.

In the following, we provide a brief description of our industrial domain and motivate this research by explaining the forces driving this thesis. Based on these forces we postulate a hypothesis and derive two research questions. Our research contributions provide answers to these research questions and indicate the correctness of the postulated hypothesis. We conclude this Chapter with an overview of the remainder of this thesis.

1.1. Motivation

This thesis has been carried out in cooperation with the Andritz Hydro GmbH (Vienna), in the scope of the HyUnify project. The context of the project were next generation programmable logic controllers targeting the hydropower domain. The goals were to develop and demonstrate concepts that ease the configuration and interconnection of control devices, increase the availability and reliability of control devices, and add advanced security features.

Basically, hydropower plants are used in different configurations to utilize water energy for producing electricity. During the last decades, computer-based controllers have been intensively deployed to automatically control, monitor and protect various processes in the plants. Figure 1.1 shows the plant internal structure and the installation including the control functions that are performed by embedded systems. In summary, these control functions are the following:

Turbine Control. Controlling the produced energy by regulating the volume of water flowing into the turbine blades.

Protection. The plant is monitored according to some operational requirements and characteristics, including the behaviour of the current produced and frequency. In

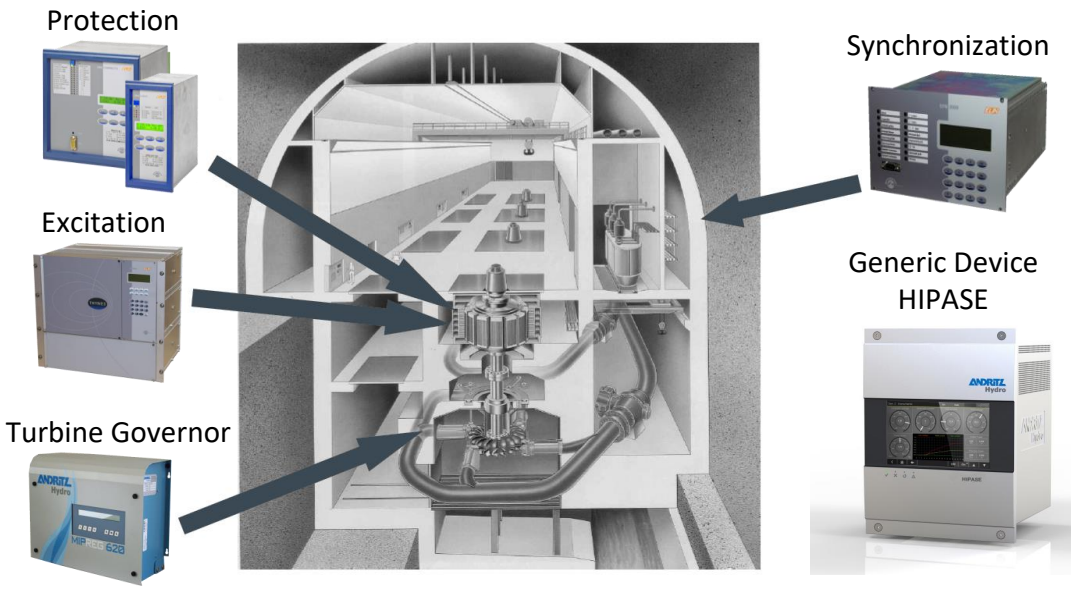


Figure 1.1.: Standard functions used for controlling processes in hydropower plants (source: Andritz Hydro).

cases of deviation from the desired behaviour, the protecting devices are capable to turn the plant into the safe state before any damage can be incorporated.

Excitation. The energy produced by generators originates from the magnetic field that is produced by rotating electromagnets, which are usually supplied with the current from external excitation machines. This configuration is realized in order to be able to dynamically control the strength of the magnetic field with the aim to balance the produced and consumed energy.

Synchronization. In many cases, multiple generators are used in a composition to produce the energy and are connected to the distribution network. To realise this scenario, it is very important that they are synchronized with respect to current characteristics. Otherwise, disastrous damages are likely to occur.

There are four distinct forces driving the development of the HIPASE device and the research presented in this thesis.

The **first force** is that many possible ways exist on how to realize and to deploy the four control functions due to different nature and configurations of plants.

The HIPASE device, shown in Figure 1.1 on the right, provides a common platform for the application software, by utilizing the principles of component-based software engineer-

ing [1]. That is, the functions are realized as compositions of software components, which correspond to small functions, such as ones that provide basic logic and arithmetic operations, but also some complex functions such as controllers and filters that can be found in the IEC 61131 [2] standard for example. The configuration of a device depends on the available sensors and actuators, and the physical properties of the target hydropower turbine. Each set of hydropower turbines found in hydropower plants has been uniquely engineered for the target plant in order to achieve the highest efficiency. Thus, hydropower turbines behave in slightly different ways.

Not only the configuration of the application software varies, but also the number of deployed HIPASE devices and their interconnections are dependent on the hydropower plant. This is due to the fact that the customers of hydropower equipment decide what devices are installed and what requirements concerning availability, reliability, security etc. need to be fulfilled. In some hydropower plants, several hot-standby devices may be available, while other plants rely on failsafe mechanisms independent of the control devices. Some systems are composed out of parts from competing hydropower equipment suppliers. The ongoing IEC 61850 standardization is an effort of enabling this interchangeability and interoperability of plant equipment [3].

The **second force** are other renewable energy sources, such as wind and solar, that become integrated into the power grid on a grand scale [4].

The complicated and limited predictability of wind and solar concerning energy conversion has an impact on the technology of hydropower plant unit control systems. The energy conversion of a plant unit needs to be adjusted in shorter periods of time in order to meet the quickly changing demand. In the past, plant operators could plan the production of electricity several days in advance. Nowadays, they have to intervene more often to meet the production goals. Hydropower plants along a river can be owned and operated by distinct companies which have differing production goals as they are competing on the electricity market. If an operator decides to produce more electricity, the water level can unexpectedly rise for other operators along a river. Such situations can lead to additional unplanned interventions. Among other things, this need of intervention leads to the integration of fast state-of-the-art communication technologies such as Ethernet and the ability of more fine-grained remote controls.

The **third force** is an industry-wide trend of collecting more knowledge about systems in use and to leverage this knowledge for optimizing operation and maintenance processes.

This is also known as “big data” [5]. Ideally, automatic mechanisms can learn about a system through various sensors, collect data and analyze it in real time. One result of such mechanisms would be to optimize the efficiency of single plants and turbines. Another one would be that these mechanisms plan dynamic maintenance intervals for individual turbines. Today, these turbines are maintained in fixed intervals. Dynamic maintenance intervals could extend the operational time of single turbines. This push for observing systems in use, has effects on the used technology in control devices. For instance, the cycle time of tasks deployed on the HIPASE device ranges from 1 millisecond to 100

milliseconds. Observing the consumed and produced data of tasks in real time, requires fast CPUs and networks for shifting data to the observer. Furthermore, this observation dependent performance load on CPUs and networks should not influence the primary function of the hydropower equipment, which is to control hydropower turbines.

The **fourth force** is that critical infrastructures, such as hydropower plants, are increasingly the target of security attacks.

For instance, the Industrial Control Systems Cyber Emergency Response Team (ICS-CERT) is an organization of the department of homeland security and provides a trusted party to report incidents in the USA [6]. They published that the number of reported security incidents in the energy sector increased from 18 incidents in the year 2010 to 59 incidents in the year 2016 [7, 8]. In the hydropower domain, this is also a result of the technical advancement necessary for serving the other three forces. Systems that have been isolated physically before are now becoming more and more complex and exposed. At some point, an entity part of the control system may be connected to the Internet or offers an interface for exchanging data with the, potentially hostile, environment.

1.2. Hypothesis and Contributions

The focus of our research was on the above mentioned control device for hydropower plants. From the four forces above the following hypothesis emerged:

Hypothesis *System knowledge enables automated resilience in industrial control systems.*

By using the term *system knowledge*, we refer to knowledge about the concrete configuration of an industrial control system.

Laprie [9] defines *resilience* as *the persistence of service delivery that can justifiably be trusted, when facing changes.* In our industrial context, service delivery means that a hydropower unit, consisting of a turbine, sensors, actuators and dedicated control devices, continues to convert energy as expected. Regarding *changes*, we are focusing on changes concerning the threats the system is facing. In a hydropower unit, the threat changes can have their source in the changes to the system or its environment. For instance through hardware faults, software bugs, misconfiguration, or security attacks.

By using the term *automated resilience*, we intend that software handles the persistence of service delivery, when facing changes, without human intervention.

Based on the hypothesis, we derived two research questions in order to narrow our research:

RQ1 *What kind of system knowledge is required?*

RQ2 *How to integrate system knowledge for automated resilience?*

During the course of answering these research questions, we created four contributions that we shall elaborate on in the remainder of this thesis:

C1 Modeling languages for specifying the system configuration and non-functional properties at design time. We are applying contract-based design for capturing the non-functional behavior of single components.

C2 Identification of the potential of self-adaptive software systems in industrial control systems. We analyzed our industrial setting regarding what detection and adaptation mechanisms can be applied against hardware faults, misconfiguration of the control logic, software bugs, security attacks, and changes in the environment.

C3 A decentralized hierarchical self-adaptive software system named Scari for integrating detection, reasoning and adaptation mechanisms at run time. This system is supported by a decentralized hierarchical and model-based knowledge base which reflects the current system knowledge.

C4 Design patterns providing architectural solutions for recurring problems. These are derived from our work on **C1** and **C3**.

Based on **C1** and **C3**, we give the following answer to **RQ1**: *Knowledge about application logic, hardware/software components, and non-functional properties, needs to exist for enabling automated resilience.*

Based on **C1**, we give the following answer to **RQ2**: *At design time, system knowledge can be leveraged for automated verification of industrial configurations.*

Based on **C2** and **C3**, we give the following answer to **RQ2**: *At run time, system knowledge can be leveraged for automatically detecting faults and adapting to situations.*

In conclusion, our research confirms the postulated hypothesis.

For the sake of clarity, Figure 1.2 provides an overview of the hypothesis, research questions, answers, contributions and peer-reviewed papers. Starting with the hypothesis, we derived two research questions. We answer them with three summarizing answers, which are based on three contributions. The fourth contribution, namely design patterns, originated from our contributed modeling languages and self-adaptive software system. These design patterns are architectural solutions to recurring problems and provide an intellectual value on their own. In total, we published nine peer-reviewed papers.

1.3. Structure

The contributions to our research are devoted to the phases design and run time. Design time is the phase starting with the development of a hardware or software component

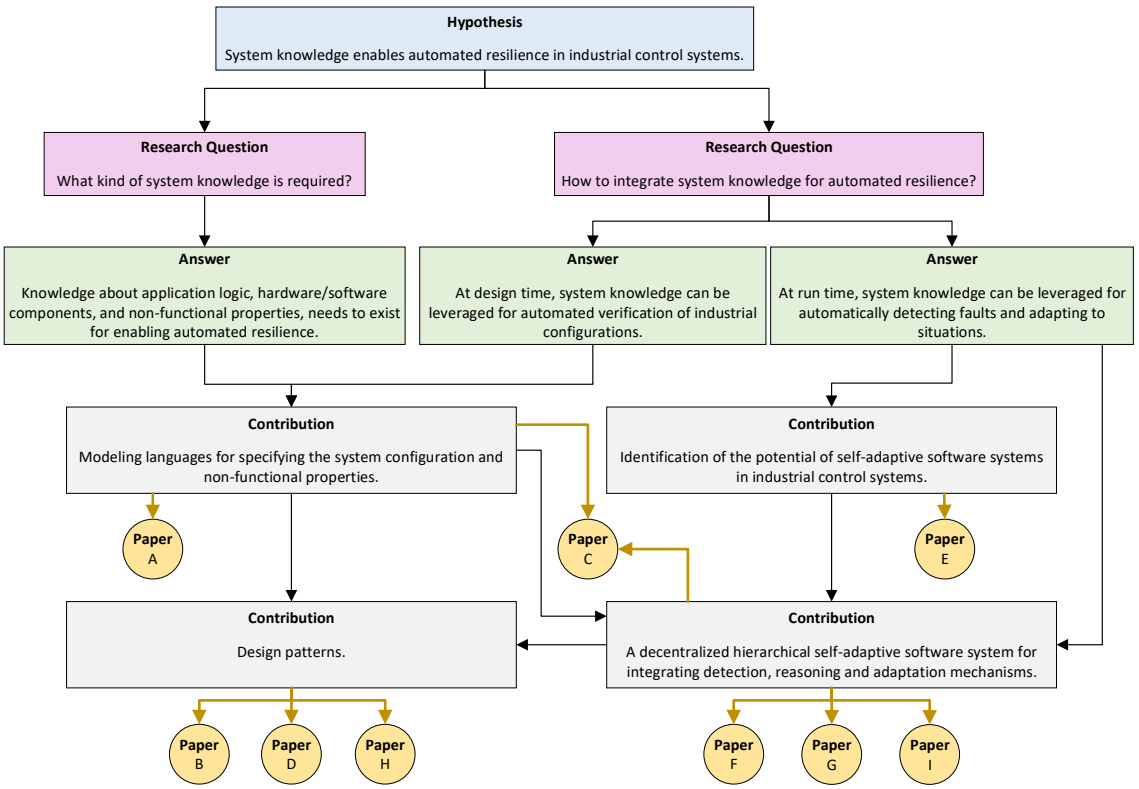


Figure 1.2.: Overview of the thesis.

and ending with the commissioning of a control system. Run time is the phase a control system is in operation. The remainder of this thesis is structured as follows:

Chapter 2 provides an introduction to model-driven engineering, contract-based design, and self-adaptive software systems.

Chapter 3 relates this research to the state-of-knowledge and compares our approach to the literature.

Chapter 4 explains the industrial setting and provides an overview of our approach. It concludes with an exemplary use case.

Chapter 5 presents our research targeting the design time of industrial control systems in the hydropower domain. Here we present **C1** and parts of **C4**. We propose modeling languages for specifying the system configuration and non-functional properties. At the end of this Chapter we briefly present four design patterns.

Chapter 6 is devoted to our research targeting the run time of industrial control systems. Here we present **C2**, **C3**, and parts of **C4**. We identify the potential of a self-adaptive system in our industrial context. We propose a novel self-adaptive software system named Scari for adding resilience mechanisms. At the end of this Chapter we briefly present four

design patterns.

Chapter 7 concludes this thesis and gives directions for future work.

Chapter 8 showcases nine peer-reviewed publications.

2. Background

Here, we provide a brief introduction to the theoretical background and key terms used in this thesis. Section 2.1 provides a short introduction into Model-Driven Engineering (MDE). We give an overview of contract-based design in Section 2.2. Section 2.3 presents the concept of self-adaptive software systems. In order to show what the goals of self-adaptive software systems are, we discuss in Section 2.4 a hierarchy of self-* properties. Usually, a self-adaptive software system is realized with a closed-loop mechanism. We present and discuss three feedback loops used for adaptation in Section 2.5. These loops represent the essence of architectures that aim to fulfill one or several of the presented self-* properties. In Section 2.6, we show how adaptation loops can be organized in distributed systems. Finally, in Section 2.7, we present the concept "Models@Run.time" that aims to utilize models based on the MDE principles in order to support self-adaptive software systems.

2.1. Model Driven Engineering

MDE is a methodology where models are the key artifacts of all development-related activities and tasks [10]. We apply MDE in our approach at design and run time.

The core principle is "Everything is a model" [11]. Model-Driven Software Engineering (MDSE) is a subset of MDE and only focuses on the development of software. However, in the literature, the terms MDSE and MDE are often used interchangeably.

MDE promises improvements in productivity, portability, interoperability, maintainability and documentation of software or development processes [12]. The work in [13] mentions *it is difficult to provide absolute measures of the benefits of MDE.* For instance, studies found in the literature have reported *productivity gains ranging from -27% to +1000%* [14]. Whittle et al. [15] surveyed 450 MDE practitioners and interviewed 22 more from 17 different companies representing 9 different industrial sectors. They highlight that the use of MDE is widespread and surprisingly *the companies who successfully applied MDE largely did so by creating or using languages specifically developed for their domain, rather than using general-purpose languages such as UML.*

Kühne [16] uses, in the context of MDE, the following definition for models: *A model is an abstraction of a (real or language-based) system allowing predictions or inferences to be made.* Note that a model is always a model *of* something.

Figure 2.1 illustrates the core principle of MDE [10]. Note that a metamodel is in fact a model.

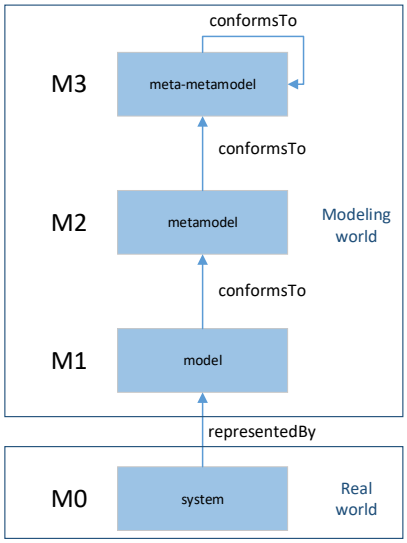


Figure 2.1.: Four-layer metamodeling architecture typically used in MDE (adapted from [11]).

- **M3**: This layer is the basis of the metamodeling architecture. Its purpose is to provide a modeling language for defining modeling languages. Usually, a meta-metamodel is defined reflexively, that means it can define itself. In practice, it does not make any sense to define further meta layers [10]. In Figure 2.1, this behavior is described by a *conformsTo* relationship.
- **M2**: The purpose of this layer is to describe modeling languages which are used on the next layer for specifying the actual model. It has to conform to the meta-metamodel at layer M3, similar to the way a programming language has to conform to its grammar. For instance, UML itself resides on this level.
- **M1**: Models at this layer represent and abstract modeled systems. They have to conform to the corresponding metamodel. An example would be an UML model describing classes of a software.
- **M0**: This layer is not part of the modeling world but part of the real world. It consists of real systems, which are abstracted and represented by M1 models.

For demonstration purposes, we present the example hierarchy used in the Unified Modeling Language (UML) 2.4.1 Infrastructure specification [17]. Figure 2.2 illustrates this hierarchy. Note that the Object Management Group (OMG) uses the notion *instanceOf* instead of *conformsTo*. Meta Object Facility (MOF) is an OMG standard [18] for describing meta-models, such as UML. As we can see in Figure 2.2, UML concepts, such as *Attribute*, *Class* or *Instance*, are defined by the element *Class* of MOF. The user model

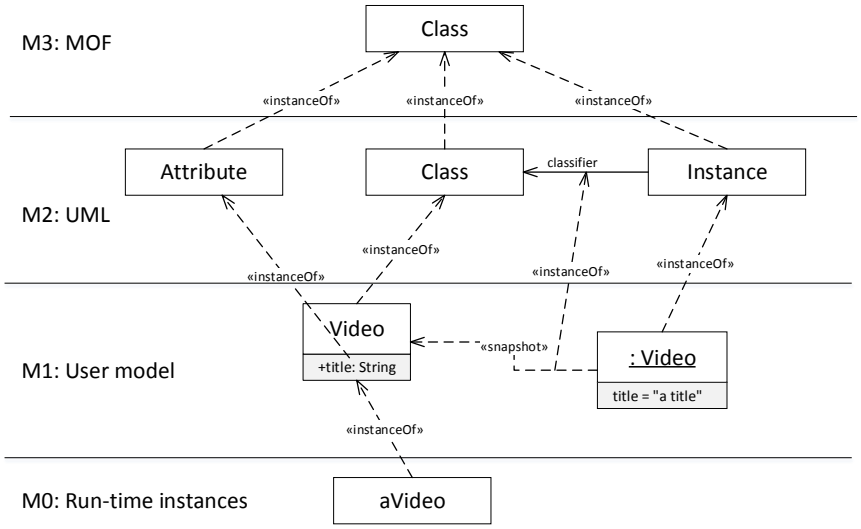


Figure 2.2.: UML example four-layer metamodel hierarchy (adapted from [17]).

on layer M1 uses instances of the UML layer, for defining the class *Video* and an instance specification (snapshot) of *Video*. Note that class specifications and instances (objects of classes) are defined on the same layer. Finally, such a *Video* class represents the real world concept of a video.

According to Brambilla et al. [10], a modeling language is defined through three parts:

Abstract Syntax: The abstract syntax of a modeling language is the metamodel. It describes the structure of a language and how different elements are connected and can be combined. It is independent of any particular representation or encoding [10]. Usually, a metamodel contains classes, attributes and associations for describing the language. Such an abstract syntax can be further improved by constraints defined with constraint languages. For instance, a simple constraint would be if the name of an element always has to start with an upper-case letter.

Concrete Syntax: Metamodels only define the abstract syntax but not the concrete notation of a modeling language. Concrete syntaxes are specific visual representations of a metamodel. They can be either textual or graphical. Of course, it is possible to define both for one metamodel. Designers work with concrete syntaxes when they manipulate a metamodel. For instance, if the concrete syntax is graphical, they use one or more diagrams.

Semantics: The correct usage and meaning of elements or the meaning of the different ways they can be combined is described by semantics. Brambilla et al. [10] point

out that *the semantics of a language can be defined in various ways: by defining all concepts, properties, relationships and constraints through a formal language; through practical implementations of code generators which implicitly define the semantics of the language by generating code; or by defining in-place transformations for simulating the model's behavior.*

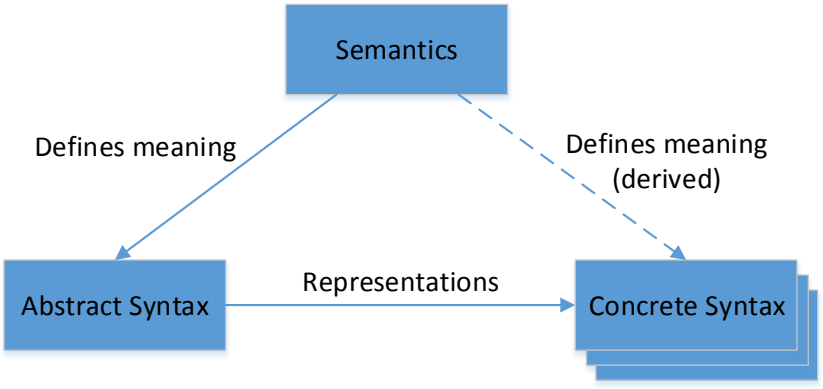


Figure 2.3.: The three parts of a modeling language and their relation to each other (adapted from [10]).

Figure 2.3 highlights these parts and shows the relationships between them. Several concrete syntaxes can represent the abstract syntax. The semantics defines the meaning of the abstract syntax and indirectly the meaning of concrete syntaxes. Brambilla et al. [10] state that all three parts are mandatory for a modeling language to be well-defined. For instance, a partial or wrong specification of the semantics enables the wrong usage of a language and leads to misinterpretations of the meaning of language elements and purpose. Different people may understand the concepts and models differently.

Modeling languages typically allow to define different models targeting *static* (or *structural*) or *dynamic* aspects of a problem or solution. The first aspect describes modeled entities and their relations, while the dynamic aspect describes their behavior, e.g., actions and interactions [10].

Generally, languages can be classified into two categories:

Domain–Specific Modeling Language (DSML) is a language designed for the purpose to target a specific domain. A DSML should *ease the task of people that need to describe things in that domain*[10].

General–Purpose Modeling Language (GPML) is a language that can be applied

to any domain. Therefore, a GPML is more complex and more complicated to understand than a DSML.

Brambilla et al. [10] mention that *this distinction is not so deterministic and well-defined*. As an example, UML can be used to model any kind of vertical domain (GPML) but can be seen as a DSML tailored to specify software systems. Also, it is possible to extend and customize UML for domain-specific needs by leveraging UML Profiles.

A DSML and a Domain–Specific Language (DSL), used in the non-modeling world, are very similar. Therefore, instead of using DSML and GPML, we use the terms DSL and General–Purpose Language (GPL) in the context of this thesis. Brambilla et al. [10] note that DSL is, in the modeling community, instead of DSML by far the most widely adopted acronym. The same is true for the acronym GPL.

2.2. Contract-based Design

In our proposed approach, we are leveraging contract-based design for capturing non-functional properties in order to describe how a component behaves. Contract-based design usually sees a component as an abstraction - a hierarchical entity that represents a single unit of design [19, 20]. Therefore, in the context of contract-based design, a component can represent, for instance, a Program Organization Unit (POU), an interface module, a software application or a control device.

The essence of this paradigm is to decompose a component into different independent views referred to as contracts, which capture the behavior of a target functional or non-functional property under certain conditions [20, 21]. This approach significantly reduces the complexity of design and verification because the single properties become manageable.

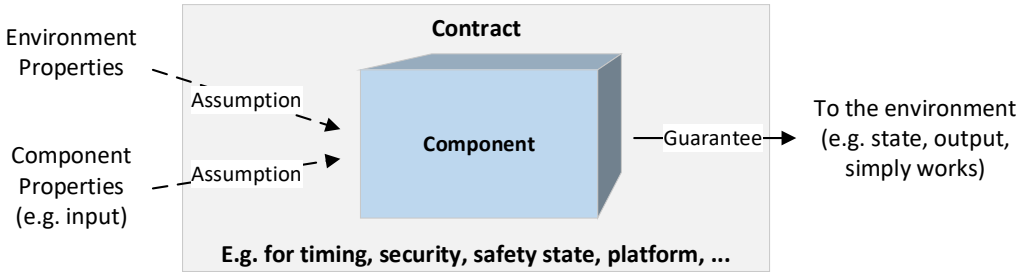


Figure 2.4.: Contract assumptions and guarantees of a component.

Informally, a contract is a set of assumptions and guarantees. Figure 2.4 illustrates the main purpose of a contract.

An assumption asserts what a contract expects from the component environment (this can include interactions with other components) and its own properties. Additionally, an assumption provides a certain context for the guarantees. The condition contained in an assumption can reference for instance input data, events or system properties. In general, the available variables are set or inferred by the analysis environment.

A guarantee describes what a component provides to the environment if the corresponding assumptions become valid. In the simplest case a guarantee states that a component just works under the constrained context. More complex contracts define limits on e.g., output data, environment characteristics or non-functional properties such as timing.

Historically, contract-based design is influenced by Meyer's design-by-contract principle [22] for object-oriented software [23]. The main difference is that contract-based design goes much further and provides means to integrate components in the design hierarchy [24]. This is achieved by capturing the context by assumptions (which may include platforms, other components, etc.) under which a component behaves as specified by the guarantees. Furthermore, a system can be viewed by selecting only appropriate contracts of interest.

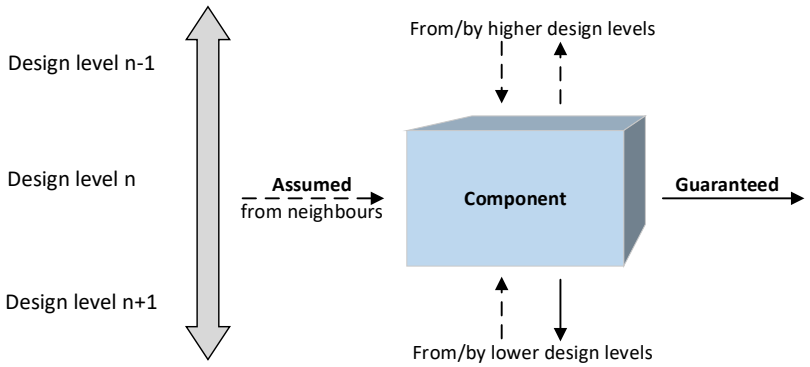


Figure 2.5.: Contract over different design levels (adapted from [25]).

Figure 2.5 illustrates that contract-based design not only allows analyzing components on a horizontal design level (e.g., interaction between software modules, hardware devices, etc.). It also enables analysis to take place on a vertical level between different layers of abstraction [23].

A solid theoretical foundation has been devised by several authors, including Benveniste et al. [23, 26], and Sangiovanni-Vincentelli et al. [19]. In the following, we briefly describe the relevant parts for this thesis of the contract-based design theory based on the descriptions provided by Benveniste et al. [26] and Vanherpen [27].

A contract C consists of a set of assumptions A and guarantees G describing the preconditions and postconditions of a component in terms of its set of design variables VAR,

respectively. Or, more formally:

$$C = (VAR, A, G)$$

A component M is said to satisfy a contract C, formulated as $M \models C$, whenever it fulfills the set of guarantees under the given set of assumptions. Using set theory, this can be formalized as [26]:

$$M \models C \text{ if and only if } M \cap A \subseteq G$$

A contract is said to be *consistent* if its set of implementations is nonempty ($M \neq \emptyset$) and *compatible* if its set of environments E is nonempty ($E \neq \emptyset$). Note that E is an environment of C if and only if $E \subseteq A$. A consistent contract, however, does not imply a consistent design. If, for example, component M interfaces with component M', consistency can only be guaranteed if variables belonging to the set of guarantees of component M equals the variables belonging to the set of assumptions of component M'.

Contract-based design theory defines two operators relevant for this thesis: the *conjunction* operator $\wedge$ and the *composition* operator $\otimes$. Consider two contracts representing different views on a POU named A:

$$C_A^D \begin{cases} variables : \begin{cases} inputs : dx \\ outputs : dy \end{cases} \\ types : dx, dy \in \mathbb{R} \\ assumptions : dx < 5 \\ guarantees : dy > 2 \end{cases} \quad \text{and } C_A^T \begin{cases} variables : \begin{cases} inputs : c \\ outputs : ta \end{cases} \\ types : c \in \mathbb{L}, ta \in \mathbb{R}_+ \\ assumptions : c = \text{"}imx28\text{"} \\ guarantees : ta \leq 50\mu s \end{cases}$$

C_A^D is a contract that guarantees that data point dy is greater than two if the input data point dx is less than five. C_A^T specifies that the execution time of POU A is less than or equal to fifty microseconds, assuming that it is executed on an i.MX28 processor.

Contracts that relate to different aspects that may exist on a single component can be combined using the conjunction operator $\wedge$. These different aspects relate to the different concerns stakeholders have with respect to the component under design.

Let $C' = (A', G')$ and $C'' = (A'', G'')$ be contracts related to the implementation of two different viewpoints on a single component M, then the resulting contract $C = C' \wedge C''$, can be obtained as follows [26, 27]:

$$A = (A' \cup A'')$$
$$G = (G' \cap G'')$$

$C_A^D \wedge C_A^T$ results in the contract C_A:

$$C_A \begin{cases} variables : \begin{cases} inputs : dx, c \\ outputs : dy, ta \end{cases} \\ types : dx, dy \in \mathbb{R}, c \in \text{Ł}, ta \in \mathbb{R}_+ \\ assumptions : \begin{cases} dx < 5 \\ c = \text{"}imx28\text{"} \end{cases} \\ guarantees : \begin{cases} dy > 2 \\ ta \leq 50\mu s \end{cases} \end{cases}$$

Note that a conjunction of contracts relaxes the assumptions and enforces the guarantees. As such, if there exists an implementation M that satisfies the conjunction of two contracts, the implementation satisfies either contracts as well. This is formally written as [26, 27]:

$$\text{If } M \models C' \wedge C' \text{ then } M \models C' \text{ and } M \models C''$$

In our industrial setting it is often the case that distinct components are interconnected through their interfaces. For instance, a module of a control device is interconnected with an ACPU. Further on, an ACPU is interconnected with an CCPU. Similar, POUs are interconnected in order to build FUPs. In such cases, the composition operator $\otimes$ is proposed by [26] to combine contracts. Let now $C' = (A', G')$ and $C'' = (A'', G'')$ be the contracts of two connected components M' and M'', respectively. Then the composition $C = C' \otimes C''$ can be obtained as follows:

$$A = (A' \cap A'') \cup \neg(G' \cap G'')$$
$$G = (G' \cap G'')$$

Following contract C_B describes a POU named B:

$$C_B \begin{cases} variables : \begin{cases} inputs : dy, c \\ outputs : dz, tb \end{cases} \\ types : dy, dz \in \mathbb{R}, c \in \text{Ł}, tb \in \mathbb{R}_+ \\ assumptions : \begin{cases} dy > 1 \\ c = \text{"}imx28\text{"} \end{cases} \\ guarantees : \begin{cases} dz = 1 \\ tb \leq 30\mu s \end{cases} \end{cases}$$

Interconnecting POU A with POU B would lead to the contract composition $C_A \otimes C_B$. This results in the following contract C_{AB}:

$$C_{AB} \begin{cases} variables : \begin{cases} inputs : dx, c \\ outputs : dz, ta, tb \end{cases} \\ types : dx, dz \in \mathbb{R}, c \in \text{Ł}, ta, tb \in \mathbb{R}_+ \\ assumptions : \begin{cases} dx < 5 \\ c = "imx28" \end{cases} \\ guarantees : \begin{cases} dz = 1 \\ ta \leq 50\mu s \\ tb \leq 30\mu s \end{cases} \end{cases}$$

For a more detailed description of the contract-based design theory and how contracts can be refined between vertical design abstraction layers, we refer to [19, 23, 26].

2.3. Self-Adaptive Software Systems

A significant contribution of our approach is a self-adaptive software system that enables to add resilience mechanisms to individual and compositions of control devices.

Historically, the intention of building self-adaptive software systems has been around for some time. Though not being the first talking and writing about self-adaptive software systems but making significant investments, IBM introduced in 2001 the Autonomic Computing Initiative in response to their observation that the *main obstacle to further progress in the IT industry is a looming software complexity crisis* [28]. They argued that systems were becoming too interconnected, too diverse and complex for even the most skilled system integrators to install, configure, optimize, maintain, and merge. Back then, *IBM researchers predicted that by the end of the decade the IT industry would need up to 200 million workers, equivalent to the entire US labor force, to manage a billion people and millions of businesses using a trillion devices connected via the Internet* [29]. Based on this idea of the future, they envisioned the need to develop computer systems that can manage themselves when given high-level objectives. Since then, the term Autonomic Computing has emerged into a broader context related with Organic Computing, bio-inspired computing, self-organizing systems, ultrastable computing, and adaptive systems, to name a few [29]. As pointed out by Salehie and Tahvildari [30], the term *self-adaptive software system* is focused on the domain of software systems. In the following, we only use this term instead of Autonomic Computing or others as it narrows the scope. Furthermore, we use this definition of a self-adaptive software:

> Self-adaptive software modifies its own behavior in response to changes in its operating environment. By operating environment, we mean anything observable by the software system, such as end-user input, external hardware devices and sensors, or program instrumentation. [31]

The fundamental reason for applying self-adaptive software systems is the increasing cost of handling the complexity of software systems to achieve their goals [30, 32].

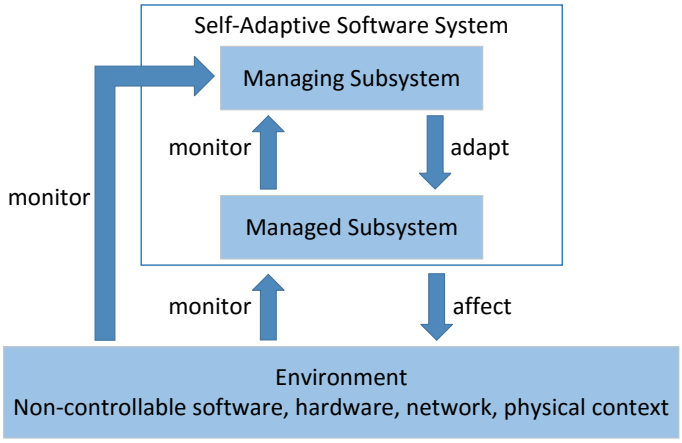


Figure 2.6.: Parts of a self-adaptive software system (adapted from [33]).

Typically, self-adaptive software systems follow an external (architecture) approach [30]. An internal approach interweaves application and adaptation logic based on programming language features such as exceptions, conditions, and parametrization. The issue with an internal self-adaptive software system is that sensors, actuators, parallel adaptation processes and actual purpose of an application are complicated to engineer within one software design. This further leads to notable drawbacks, e.g., with respect to scalability, testability and maintainability. In an external approach, as illustrated in Figure 2.6, the domain-specific application logic named *Managed Subsystem* is monitored by a *Managing Subsystem*. The *Managing Subsystem* is where the actual adaptation logic resides. It additionally monitors the *Environment* that may consist of other software, hardware, network, or of the physical context (including humans). Based on monitored data and analyzed problems the *Managing Subsystem* decides whether and what to adapt inside the *Managed Subsystem*.

2.4. Hierarchy of Self-* Properties

In this thesis, we are proposing a software architecture that enables self-adaptiveness of control devices. Self-Adaptiveness is a broad term and represents the sum of several self-* properties. Salehie and Tahvildari [30] discuss these properties in detail and represent them in the hierarchy illustrated in Figure 2.7.

The top level named *General Level* contains global properties of self-adaptive systems. Terms, found in literature, which are basically a subset of self-adaptiveness are *self-managing*, *self-governing*, *self-maintenance*, *self-control*, *self-evaluating*, and *self-organizing*.

The *Major Level* terms are coined by the IBM Autonomic Computing Initiative and serve as the defacto standard in self-adaptive systems [30]. The following four properties

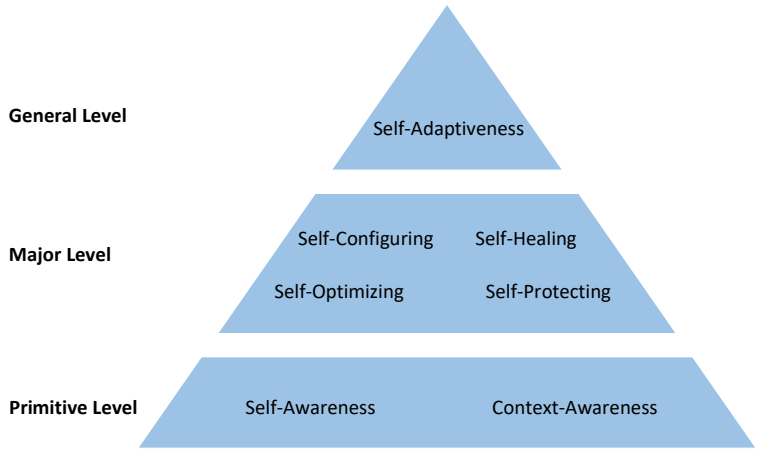


Figure 2.7.: Hierarchy of self-* properties (adapted from [30]).

are generally also known as *self-CHOP* [34]:

- **Self-configuring**: A system reconfigures itself automatically in response to changes following high-level policies.
- **Self-healing**: A system automatically detects, diagnoses, and reacts to software and hardware failures by *healing* itself.
- **Self-optimizing**: A system continually seeks opportunities to improve its own performance and resource allocation.
- **Self-protecting**: This property has two aspects. One is that a system automatically defends itself against malicious attacks or cascading failures. The other one is that it mitigates the effects of attacks.

The *Primitive Level* represents the base of the *Major Level* and consists of two properties. Without them a self-adaptive system would not be able to realize the properties from the *Major Level*:

- **Self-awareness**: A system is aware of its own states and behaviors.
- **Context-awareness**: A system is aware of its operational environment.

All these properties above define what self-adaptive systems are targeting to achieve. In our approach we are aiming for all *Primitive Level* and *Major Level* properties except self-optimization. The vision of the proposed system is to autonomously configure, heal, and protect itself based on its own state and the operational environment.

2.5. Adaptation Loops

The proposed self-adaptive software system of this work is built around an adaptation loop. There are different options of how a self-adaptive software system can be organized. Muccini et al. [35] reveal in a systematic literature review that concerning Cyber Physical Systems (CPS), the so-called Monitor Analyze Plan Execute - Knowledge (MAPE-K) loop is by far the dominant adaptive mechanism with a share of 60%. It is followed by multi-agent and self-organization based technologies (both have 29% - some studies combine technologies). Multi-agent systems are large-scale open decentralized systems that consist of autonomous components or systems [36] that work together for achieving a common goal. Self-organization techniques are inspired by nature, where behavior emerges, e.g., from cells [37]. Multi-agent systems and self-organization techniques are out-of-scope of this work as they do not fit our industrial setting.

In the following, we present three loops that try to grasp the necessary steps and activities of a self-adaptive software system. After that, we shortly discuss the commonalities between them and highlight five design patterns of how loops based on the MAPE elements can be organized.

2.5.1. MAPE-K

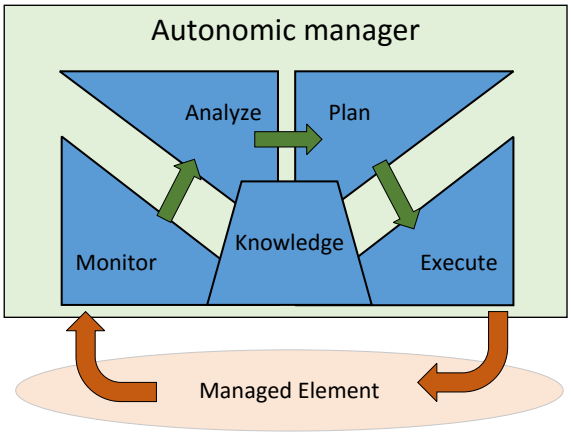


Figure 2.8.: MAPE-K loop (adapted from [28]).

Figure 2.8 illustrates the MAPE-K loop, introduced by Kephart and Chess [28] defining IBMs Autonomic Computing vision. It consists of the steps *Monitor*, *Analyze*, *Plan*, *Execute*, and a shared part representing *Knowledge*. The target of MAPE-K is the *Managed Element*, which is monitored with sensors and changed with actuators. The *Monitor* step gathers information about the *Managed Element* that is usually related to the current performance and load of the system [38]. The *Analyze* step considers the data, identifies

problems and attempts to find the source or cause of it. The *Plan* step reacts to the results of the *Analyze* step and creates a set of actions to remedy a problem. The last step, which is named *Execute*, implements these actions and changes the *Managed Element* through actuators. *Knowledge* is the central point where all the information within a MAPE-K loop comes together. The *Monitor* stores its observed data at this point. The *Analyze* step uses it to find anomalies. The *Plan* step leverages it to create actions and gathers its policies and goals from there. Finally, the *Execute* step stores its record of executed actions in it.

As we can see in Figure 2.8 there is an *Autonomic Manager* around the MAPE-K loop. That is basically an interface for controlling and monitoring the adaptive system. [28] foresaw a plethora of *Autonomic Managers* each managing for instance a hardware resource (e.g., CPU, printer, storage) or software resource (e.g., database, service, legacy system).

2.5.2. OODA

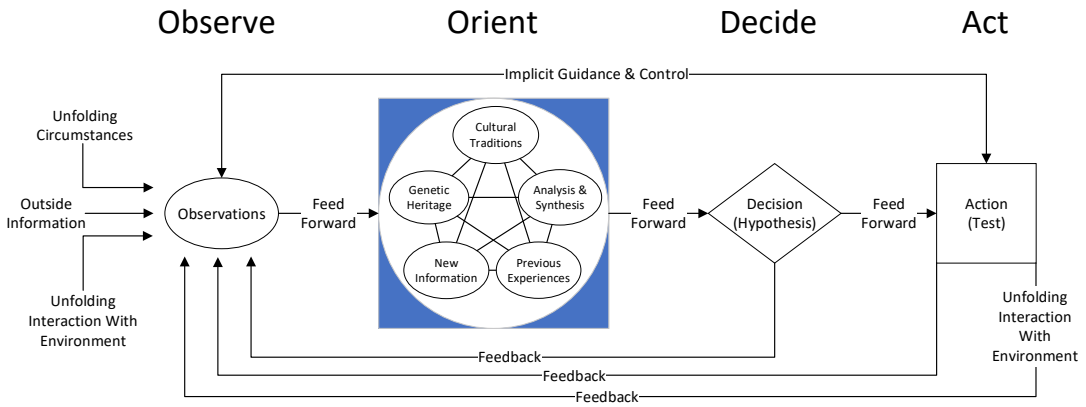


Figure 2.9.: OODA loop (adapted from [39]).

Colonel John Boyd was a United States Air Force fighter pilot and Pentagon consultant who developed the first version of his Observe Orient Decide Act (OODA) loop for explaining how to achieve success in air-to-air combat in the 1950's. Later, he expanded his groundbreaking work and hypothesized that it is the essence of winning and loosing of organizations and people [40]. As pointed out by other authors, it lends itself well to self-adaptive software systems [41, 42]. In the OODA loop, *Observe* means to gather, monitor, and filter data. Figure 2.9 illustrates the type of data that can be observed; implicit guidance and control refers to the significant influence of the *Orientation* step. In the *Orient* step a list of options is derived through analysis and synthesis, previous ex-

perience, new information, and of course as the loop is intended for humans, genetic and cultural heritage. The derived list of options is then fed forward to the *Decide* step where the best hypothesis is selected via a ranking. In the last step, the selected option is acted out and in a way tested in the environment. As pointed out by John Boyd, *orientation shapes observation, shapes decision, shapes action, and in turn is shaped by the feedback and other phenomena* [40]. He demonstrated that it is crucial to go through the OODA loop faster and better than an opponent when facing direct combat. Further, he noted that the *entire loop (not just orientation) is an ongoing many-sided implicit cross-referencing process of projection, empathy, correlation, and rejection.* In our approach we illustrate how we transfer the OODA loop to our variant of a loop that targets to adapt systems.

2.5.3. CADA

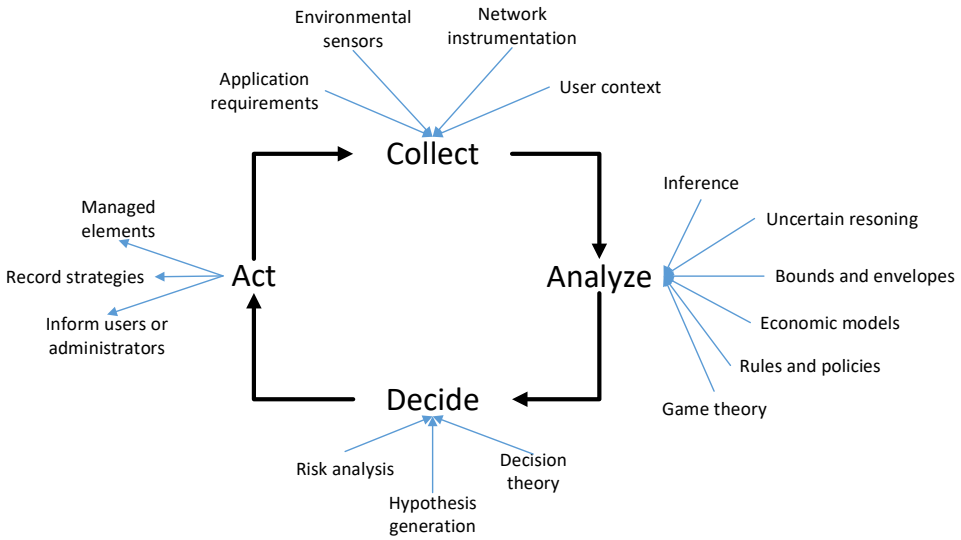


Figure 2.10.: CADA loop (adapted from [43]).

Dobson et al. [43] describe the generic *Collect* - *Analyze* - *Decide* - *Act* loop for autonomic communication systems. The field of autonomic communication targets to improve the ability of networks and services to cope with unpredicted changes concernings topology, load, task and so on. As we can see in Figure 2.10 it is similar to MAPE-K and the OODA loop, but more generic. In the *Collect* activity data is gathered from several sources, in the *Analyze* activity analyzed, then a *Decision* is made, and finally acted out in the *Act* activity. The loop is annotated with several techniques and approaches, which can be applied for implementing the single activities. As mentioned by Cheng et al. [44], reasoning in self-adaptive systems typically involve these four activities.

2.5.4. Discussion of the Adaptation Loops

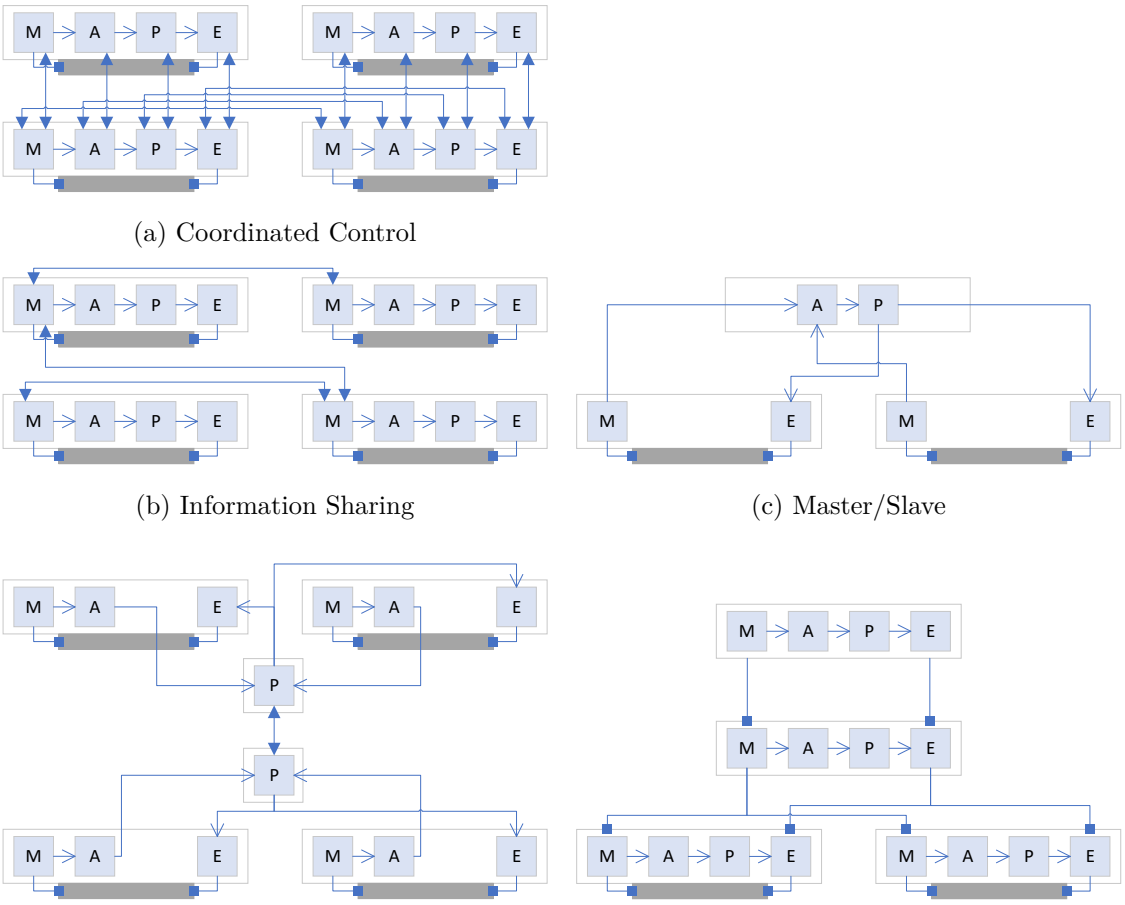
In essence, the three presented adaptation loops are variants of the same idea, which is to have a chain of activities that lead to an appropriate response to a problem of the managed system. However, the loops vary concerning the different steps and feedback. MAPE-K introduces the *Knowledge* part as common information source for each activity taking place. OODA emphasizes that the different steps give feedback to what is observed and the loop is essentially driven by the *Orient* phase. The *Orient* phase of OODA corresponds to the *Analyze* and *Plan* steps of the MAPE-K loop. OODA introduces an explicit separate *Decide* phase that is embedded into the *Plan* step of MAPE-K. The CADA-loop is a generic version of an adaptation loop. We are including it because it highlights the different technologies which can be applied in each step.

An important design decision concerning self-adaptive systems is whether a loop is realized time-driven or event-driven. A time-driven MAPE would adapt periodically. An event-driven MAPE is triggered by an observation made by a monitor. In this thesis we present an adaptation loop that is in principle event-driven, but the *Analyze* step is split into distinct reasoning mechanisms that could be time-driven.

2.6. Design Patterns for Decentralized Control in Self-Adaptive Systems

We propose a decentralized self-adaptive system that is organized as a hierarchy. There exist different ways of how a distributed system can be controlled by one or several adaptation loops. Weyns et al. [33] gathered five patterns for decentralized control in self-adaptive systems that describe how MAPE loops can be related to each other. We shortly describe the essence of these patterns in the following:

- **Coordinated Control Pattern**: Consider a distributed system where each node has its own MAPE loop. This pattern proposes that all the *Monitor*, *Analyze*, *Plan*, and *Execute* steps coordinate their operation with corresponding peers of other loops. For instance in Figure 2.11a, *Analyze* entities interact with each other to make a decision about the need for an adaptation.
- **Information Sharing Pattern**: In this pattern all *Monitors* in a distributed system share their observed states with each other, while *Analyze*, *Plan* and *Execute* entities are acting independently from their counterparts on other nodes. Figure 2.11b shows an example of this pattern.
- **Master/Slave Pattern**: There exists a central master component that is responsible for the *Analyze* and *Plan* steps of adaptations. Figure 2.11c illustrates the situation where two slave loops are controlled by one master component. The slave nodes in such a system are responsible for monitoring states and executing actions.



(a) Coordinated Control

(b) Information Sharing

(c) Master/Slave

(d) Regional Planning

(e) Hierarchical Control

Figure 2.11.: Five patterns of how MAPE can be organized (adapted from [33])

- **Regional Planning Pattern**: Such a distributed system is partitioned into regions where in each region a central component performs the *Plan* step. The *Monitor*, *Analyze*, and *Execute* steps are deployed on other nodes. Figure 2.11d shows that the central *Plan* components can be connected with each other.
- **Hierarchical Control Pattern**: This pattern organizes MAPE loops in hierarchies such as in Figure 2.11e. For instance a loop is in control of a node. If it cannot adapt, a situation can be escalated to a higher loop that possesses a broader control of the target distributed system. Furthermore, the information passed between the different levels may be filtered or aggregated.

2.7. Models@Run.time

Our proposed self-adaptive software system supports the distinct activities with run time models derived from design time. Models@Run.time is a term for describing the research field of utilizing software models, specified according to the MDE principles, for self-adaptive software systems [45]. The term *run.time* refers to the novelty opposed to the fact that traditional MDE has been applied for describing the architecture of software and systems at design time. One of the most prominent examples of such a design time technology is UML standardized by the OMG [46].

A run time model is essentially a software model that represents at run time parts of a real system and is causally connected to it (e.g., a system change leads to a change in the model). Such a software model possesses several properties that are in our opinion beneficial for self-adaptive software systems:

- Design-time models are in many domains already available and can be transformed to *living* specifications at run time.
- A model can be queried in order to find resources and to learn something about a system.
- Software models are based on modeling languages and adhere to semantics. Simply put, a mechanism cannot easily construct a model randomly and arbitrarily.
- Validation is an important aspect of software models and constraints can be provided for ensuring that a run-time model is correct.
- An adaptation mechanism can explore if a change would be correct by forking a model and trying out different configurations.
- Transformation is an essential part of MDE. Manipulating and transforming models to executable artifacts offers systems an opportunity to self-modify. Furthermore, a changed run time model could be transformed to input formats for a variety of simulation and verification software.
- Run time models which change over time can be transformed back to design time models.

Giese et al. [47] distinguish between three different kinds of run-time models within a self-adaptive software system. Note that not all kinds have to be present within an adaptive system but all of them are useful for each activity in an adaptation loop:

- **System Models**: This kind of model reflects an abstract view of the system itself. It allows an adaptive system to reason about the system and to simulate different kinds of configurations. Consequently, such a model needs to be in sync with the real system.
- **Context Models**: A run time model can be used to reflect the context of a system and to specify it in a processable way. The characteristics of the context cast in such

a model can either be derived directly from the environment by sensors or indirectly derived from other observations.

- **Requirements Models**: This kind of model captures the requirements and goals of a self-adaptive system. In a way it sets the boundaries of what a system can do. The collect, analyze, plan/decide, or act parts of an adaptation loop can be partly or fully configured with this kind of model. Usually this relationship is unidirectional, meaning that a system is not supposed to change its requirements. However, it can prioritize one over another.

3. Related Work

Here, we first summarize literature about contract-based design and highlight our addition to the state of knowledge (**C1**). Then we compare related work dealing with self-adaptive software systems with our proposed work named Secure and Reliable Infrastructure (Scari) (**C3**) in detail.

3.1. Contract-Based Design

Promising applications of contract-based design have been shown for several domains. This paradigm has been demonstrated for smart integrated energy management systems [48], aircraft electric power systems [20], mixed-signal integrated circuits [49], and automotive systems [23, 50, 51]. Despite these examples, contract-based design is still at its infancy [52].

Most state-of-the-art approaches either tackle single non-functional properties or take a relatively theoretical approach without concrete modeling languages. Therefore, there only exist a few modeling languages for realizing contract-based design.

Warg et al. [53] present a prototype modeling tool named SafetyAdd for contracts. Their work solely focuses on safety integrity levels.

Sievers and Madni [54] propose contract stereotypes for SysML [55] ports. Additionally they propose trained Hidden Markov models for describing flexible contracts.

Grabowski et al. [56] present a template language called SSPL that allows the specification of requirements and assertions on every system architecture level and show how contract-based requirements refinement can go hand-in-hand with architecture refinement in SysML. They provide an Eclipse-based tool supporting their method.

According to their website [57], the CHESS project sought to improve MDE practices and technologies to better address safety, reliability, performance, robustness and other non-functional concerns while guaranteeing correctness of component development and composition for embedded systems. A result of this EU project was a modeling language which allows a specification of components and corresponding contracts. The modeling language is based on UML and the profiles SysML and MARTE [58]. It allows to associate contracts with component definitions and to refine existing contracts [59].

Amorim et al. [60] propose ConSerts M, which are modular and composable contracts created at development time as part of a sound and mostly traditional safety argumentation. The focus is set on ensuring safety through the system lifecycle, even if parts of

the system are replaced or updated as part of maintenance or upgrades. Additionally, the authors propose a combination of their contract approach with an ontology-based run time reconfiguration for the use in automotive applications [61].

VerSaI (Vertical Safety Interfaces) [62] is a contract-based modeling approach, which assists the integrator of an integrated architecture in checking whether the application software components are able to run safely on the execution platform of the system. The VerSaI language is a metamodel based formalization of the typical dependencies between the certificates of an application and a platform.

Approach	MB	Types	Configurable Constraints	State-Based Modeling
SafetyAdd [53]	✓	-	-	-
Sievers and Madni [54]	✓	-	-	-
Grabowski et al. [56]	✓	-	-	-
CHESS [59]	✓	-	-	-
ConSerts M [60]	✓	-	-	-
VerSaI [62]	✓	-	-	-
This work	✓	✓	✓	✓

Table 3.1.: Comparison of contract-based design approaches.

Table 3.1 compares our work with the related work. Similar to other approaches, our work is model-based. However, we are adding to the state of knowledge the concept of contract types that allow to define which assumptions and guarantees are possible for a concrete contract instance. Furthermore, we are proposing a constraint language, which is configurable by a contract type. For instance, it allows to define that only equality and inequality operators are allowed to be used for a certain contract type. This should ease the transformation of contracts to other tools. Last but not least, we are proposing a state machine where each state represents a valid set of contracts. This is useful for capturing the distinct states of a device as a whole without adding this information to each single contract.

3.2. Self-Adaptive Software Systems

In the following, we are focusing on self-adaptive software systems that are architecture-based. An architecture is a set of elements or components, the relations between them, as well as the properties of both and denotes the high-level structure of a system. A self-adaptive system that is architecture-based maintains a model of itself and adapts itself to realize particular quality objectives using a feedback loop [63].

Table 3.2 shows conferences, workshops, journals and surveys we searched for self-adaptive software systems in order to compare our work to the current state of knowledge. We performed this search in a systematic way. First, we selected metrics of interest based on our proposed self-adaptive software system and used by the surveys of Salehie and Tahvildari [30], Krupitzer et al. [63], and Krupitzer et al. [64]. Then, we manually went

ID	Conferences/Workshops/Journals/Surveys
SOSA	International Conference on Self-Adaptive and Self-Organizing Systems
ICAC	International Conference on Autonomic Computing
Adaptive	International Conference on Adaptive and Self-Adaptive Systems and Application
SEAMS	International Symposium on Software Engineering for Adaptive & Self-Managing Systems
ICSE	International Conference on Software Engineering
ASE	International Conference on Automated Software Engineering
ECSA	European Conference on Software Architecture
MRT	International Workshop on Models@run.time
MRT-ICAC	International Workshop on Models@run.time for Self-aware Computing Systems
MRT-SeAC	Joint International Workshop on Models@run.time and Self-aware Computing Systems
SeAC	Workshop on Self-Aware Computing
DAS	Workshop on Distributed Adaptive Systems
SISSY	International Workshop on Self-Improving System Integration
SOSeMC	Self-Organizing Self-Managing Clouds Workshop
AKSAS	International Workshop on Architectural Knowledge for Self-Adaptive Systems
TAAS	ACM Transactions on Autonomous and Adaptive Systems
TSE	IEEE Transactions on Software Engineering
[30]	Self-Adaptive Software: Landscape and Research Challenges
[63]	A survey on engineering approaches for self-adaptive systems
[64]	Comparison of approaches for self-improvement in self-adaptive systems (extended version)
[65]	Self-adaptive systems: A survey of current approaches, research challenges and applications

Table 3.2.: Searched sources for comparing our work.

through the yearly proceedings or published journals. We selected suitable publications based on the title and abstract. Next, we analyzed the publications by reading them. We further expanded the search of approaches based on referenced related work. Thus, the presented approaches were not necessarily published in the enumerated sources in Table 3.2. Where available we analyzed the single approaches based on the corresponding PhD theses.

Table 3.3 compares our work concerning the self-CHOP properties. Scari (our work) is self-configuring, similar to most self adaptive systems, because the knowledge base reflects the current state of the system. The monitors, reasoning, and planning mechanisms adjust themselves dynamically through the changing knowledge. Scari aims to heal and protect control systems. This is contrary to most self-adaptive systems, which are focusing on self-optimizing their managed subsystem. In our case, self-optimization would need to change the control logic, which our approach tries to preserve as far as possible.

Out of these 24 approaches we selected nine approaches that provide a detailed description of their software architecture. In the following we briefly present these approaches:

Rainbow [68] is the most cited self-adaptive software system. It observes entities with probes and gauges for updating a central architecture model of the target system. With a DSL one can define model constraints that trigger adaptations. The example applications of Rainbow are instantiations of web-server-based systems.

MADAM [79] is a middleware and adapts mobile component-based applications based on context changes. The context includes issues describing the system infrastructure (such

Approach	Self-Configuring	Self-Healing	Self-Optimizing	Self-Protecting
Quo [66]	✓	-	✓	-
IBM Oceano [67]	✓	-	✓	-
Rainbow [68]	✓	✓	✓	-
Tivoli Risk Manager [69]	-	-	-✓	-
KX [70]	✓	-	-	-
Accord [71]	✓	-	-	-
ROC [72]	-	✓	-	-
TRAP [73]	✓	-	-	-
K-Component [74]	✓	-	-	-
Self-Adaptive [75]	✓	✓	-	-
CASA [76]	✓	-	✓	-
J3 [77]	-	-	✓	-
DEAS [78]	✓	-	-	-
MADAM [79]	✓	-	✓	-
M-Ware [80]	✓	-	✓	-
ML-IDS [81]	-	-	-	✓
PLASMA [82]	✓	✓	-	-
IBM ACRA [83]	✓	✓	-	-
DIVA [84]	✓	-	✓	-
Mistral [85]	✓	-	✓	-
RESIST [86]	✓	-	✓	-
DARE [87]	✓	✓	-	-
MARTAS [88]	✓	-	✓	-
Kubernetes [89]	✓	✓	-/✓	-
Scari (this work)	**✓**	**✓**	-	**✓**

Table 3.3.: Comparison concerning self-CHOP properties.

as battery level and network resources) and the user (such as position, noise, and user needs). It generates distinct variants and selects one variant based on a utility function.

IBM ACRA [83] stands for *A*utonomic *C*omputing *R*eference *A*rchitecture. It is based on a hierarchical organization of MAPE-K loops. IBM applied it to IT infrastructures in order to automate tasks and responses to situations. For instance, an administrator could realize an automatic workaround (reboot the server) when an out-of-system-resources incident occurs.

DIVA [84] targets dynamic software product lines in which variabilities are bound at run time. It leverages aspect-oriented modeling techniques to refine features and automatically build complete configurations before the actual adaptation happens. The current configuration is determined by a goal-based reasoning component.

Mistral [85] optimizes the trade-off between performance and power consumption as well as between the cost of an adaptation in cloud computing scenarios. It is decentralized and hierarchical.

RESIST [86] is a purely proactive approach that optimizes the reliability of embedded systems. It estimates reliability as the probability that a system performs its required functions under stated conditions for a specified period of time. The authors evaluated their system with a mobile emergency response system.

DARE [87] targets component-based software architectures in distributed settings in order to heal and optimize them. Although it is decentralized, it selects the entity with the lowest IP address to be the leader and recovery node for the others. The behavior of DARE is shown in abstract randomized experiments.

MARTAS [88] deals with the automated management of Internet of Things (IoT). It utilizes quality models for minimizing packet loss, latency, and power consumption.

Kubernetes [89] is not an architecture-based self-adaptive system. Still, we included it because it is wildly popular in the domain of cloud computing for orchestrating software containers. These software containers can realize for instance web applications. Kubernetes is a centralized approach, which ensures that the deployment of software containers on nodes conforms to a user-definable configuration. The observed metrics are saved in a key-value store. So-called controllers process values continuously within one central loop and change the system accordingly. Interestingly, it targets similar applications such as Rainbow without an architecture model and a DSL for the controllers.

Approach	**Domain**	**Model-Based**	**External**	**MAPE**
Rainbow [68]	Web Server Management	✓	✓	All
MADAM [79]	Applications for Mobile Computing	✓	✓	All
IBM ACRA [83]	IT Infrastructure	-	✓	All
DIVA [84]	Dynamic Software Product Lines	✓	✓	All
Mistral [85]	Cloud Computing	✓	✓	All
RESIST [86]	Dynamic Mobile and Embedded Systems	✓	✓	All
DARE [87]	Component-Based Software Architectures	✓	✓	All
MARTAS [88]	Internet-of-Things	✓	✓	All
Kubernetes [89]	Software Container Management	-	✓	All
Scari (this work)	**Industrial Control Systems**	**✓**	**✓**	**All**

Table 3.4.: Comparison concerning domain, model-based, external, and MAPE.

Table 3.4 compares the approaches concerning their domain, whether they are model-based, external and apply all MAPE activities. What sets our approach apart is that, as far as we know, we are the first ones applying self-adaptive system to an industrial control system and especially in the context of hydropower plants. Except Kubernetes, all compared approaches are model-based. Kubernetes relies on configuration files with predefined keywords for setting up the self-adaptive system. Furthermore, all approaches are external, which means the managed subsystems are monitored by managing subsystems, and implement all MAPE activities.

Table 3.5 compares the approaches concerning the time they are reacting, the reason for an adaptation, the targeted level they are adapting and the available adaptation techniques. Scari can operate reactive and proactive to situations. Reactive means that after a situation happened the system adapts. Proactive means that a system can adapt beforehand to avoid possible situations. Scari is able to do this because it can incorporate autonomous reasoning mechanisms, which we named syndrome processors that can recommend on their own for instance new honeypot settings or memory tests. Other systems

Approach	Time	Reason	Level	Adaptation Technique
Rainbow [68]	Reactive	TR	App/TR	Parameter/Structure
MADAM [79]	Reactive	Ctx	App	Parameter/Structure
IBM ACRA [83]	Reactive	Ctx/TR	App/Comm/TR	Parameter/Structure
DIVA [84]	Reactive	Ctx	App	Parameter/Structure
Mistral [85]	Reactive/Proactive	TR	TR	Structure
RESIST [86]	Proactive	TR	App/Comm/TR	Structure
DARE [87]	Reactive	TR	App/TR	Structure
MARTAS [88]	Reactive	TR	Comm/TR	Parameter
Kubernetes [89]	Reactive	TR	TR	Structure
Scari (this work)	**Reactive/Proactive**	**Ctx/TR**	**App/Comm/TR**	**Parameter/Structure**

Table 3.5.: Comparison concerning reaction time, reason for an adaptation, targeted level, and kind of adaptation technique.

that are only able to react need to wait for input from monitoring activities. Concerning the reason for an adaptation, we distinguish between a change in the technical resource (TR) or a change in the context (Ctx). Some approaches only monitor their technical resource for a change, while others monitor their context. Scari is capable of both because we do not restrict what the monitors are allowed to observe and they are not bound to the knowledge base. For instance, Rainbow is restricted to technical resources because everything, which is observed needs to coexist and be described in the knowledge base in advance. Level refers to the target of adaptation. Scari aims at the application (App), communication (Comm) and technical resource (TR) levels. Note that Scari is not adapting the control (application) logic, but it can deploy tasks, which are part of a control application, to other technical resources. Furthermore, it can adjust the input/output data points and reroute communication. Concerning adaptation techniques, we have the possibility of adapting parameters and the structure of industrial control systems. Summing up, we needed to create a more flexible approach regarding the above enumerated metrics than other approaches found in literature in order to fulfill the self-* properties.

Approach	Autonomous Reasoning Mechanisms	Decision Criteria
Rainbow [68]	-	Models, Policies
MADAM [79]	-	Goal, Utility
IBM ACRA [83]	-	Rules, Policies
DIVA [84]	-	Goal
Mistral [85]	-	Utility
RESIST [86]	-	Utility
DARE [87]	-	Policies
MARTAS [88]	-	Models, Utility
Kubernetes [89]	-	Rules
Scari (this work)	✓	Models, Rules, Policies

Table 3.6.: Comparison concerning autonomous reasoning mechanisms and decision criterias.

Table 3.6 compares our approach concerning autonomous reasoning mechanisms and

decision criteria. The first metric describes that reasoning mechanisms can be implemented in a way that they follow their own cycle. This is possible in Scari because of the loose coupling between monitors, syndrome processors, and the adaptation activities. As far as we know, we are the first proposing this feature. *Decision critera* is the metric for identifying the need for adaptation and for choosing suitable adaptation plans [63]. A self-adaptive system leveraging *models* as decision criteria, works out suitable adaptation plans through analysis of the models. For *rule*-based or *policy*-based approaches, rules or policies determine, how the system should react in different situations and how to adapt. *Goal*-based approaches aim at fulfilling specific system goals. These goals influence how the system should perform. During the planning process, the adaptation logic must define adaptation plans for achieving these goals. The goals can be contradicting, which must be solved by the adaptation logic. In *utility*-based approaches, utility is a function of the system value for the user and involved costs. The goal is to maximize the overall system utility. The adaptation logic evaluates the utility values of adaptation strategies and selects the one with the highest utility. One of the challenges is the difficulty of defining utility functions, as well as the complexity and the uncertainty in calculating adaptation costs and utility values [63].

Due to the flexibility of the autonomous reasoning mechanisms (syndrome processors), our approach is able to incorporate decisions based on models, rules, and policies. We are not proposing a goal-based approach because we do not have changing goals during run-time. We are not proposing utility functions because our approach is not targeted at self-optimization. This would also involve the control logic, where a centralized approach would be more suitable for optimizing a hydropower unit, plant or a network of hydropower plants. Central entities that can optimize plants already exist in the hydropower domain through control rooms that operate plants along a river.

Approach	**Decentralized**	**Hierarchical**	**Parallel Adaptations of Distinct Nodes**
Rainbow [68]	-	-	-
MADAM [79]	-	-	-
IBM ACRA [83]	✓	✓	✓
DIVA [84]	-	-	-
Mistral [85]	✓	✓	✓
RESIST [86]	-	-	-
DARE [87]	✓	-	-
MARTAS [88]	-	-	-
Kubernetes [89]	-	-	-
Scari (this work)	✓	✓	✓

Table 3.7.: Comparison concerning decentralization, hierarchical organization, and parallel adaptations of distinct nodes.

Table 3.7 compares the approaches concerning decentralization, hierarchical organization, and the possibility of parallel adaptations of distinct nodes. As pointed out by other authors [33, 63, 65, 90, 91], in the area of autonomic and self-adaptive software systems a

centralized approach with one adaptation loop is the dominant topology found in literature. According to Krupitzer et al. [63], the main challenge with decentralized approaches is that the dynamic adaptation and recovery is carried out by using partial knowledge of the system. Various degrees of decentralization are possible [63]. In fully decentralized approaches, each subsystem has a complete adaptation logic and different patterns of communication are possible. Between the extreme of a fully decentralized approach with only equal entities and the extreme of a centralized approach, hybrid approaches exist that add central components to decentralized approaches or distribute the adaptation logic functionality to subsystems [33]. In our comparison, we count every self-adaptive software system that is not centralized to the spectrum of decentralized approaches, similar to the way it is done by Weyns et al. [33] and the authors of the presented decentralized approaches.

We compare the approaches whether they are organized in a hierarchical way and if it is possible to carry out parallel adaptations on distinct nodes. DARE is the only decentralized approach not capable of this because it selects one leader node that reasons and plans for all other nodes.

Scari is not a fully decentralized approach. Through the hierarchical organization, higher nodes can adapt lower nodes. However, this suits industrial control systems, which are usually also organized in a hierarchy.

Approach	**Decentralized Knowledge Bases**	**Hierarchical Knowledge Bases**
Rainbow [68]	-	-
MADAM [79]	-	-
IBM ACRA [83]	✓	-
DIVA [84]	-	-
Mistral [85]	-	-
RESIST [86]	-	-
DARE [87]	✓	-
MARTAS [88]	-	-
Kubernetes [89]	-	-
Scari (this work)	✓	✓

Table 3.8.: Comparison concerning decentralized knowledge bases and their possible hierarchical organization.

Table 3.8 compares, whether the approaches utilize decentralized knowledge bases and their possible hierarchical organization. A decentralized knowledge base resides on each entity in contrary to a centralized one. It can contain all available knowledge or only aspects. In Scari, an entity contains the knowledge about itself and child entities. In IBM ACRA, the knowledge bases are not necessarily connected. In DARE, every entity maintains a partial knowledge base and the overall architecture is discovered through message exchanges. What sets Scari apart is that we propose a hierarchical knowledge base, which reflects the hierarchical organization of the self-adaptive system.

In addition to these comparisons, we are the first ones incorporating contract-based

design into run-time models for self-adaptive software systems.

4. Overview

In this Chapter, we first illustrate the target industrial system. Based on this illustration, we show the concept of our approach in Section 4.2. Finally, we provide a brief use case demonstrating the combination of design and run time in Section 4.3.

4.1. Industrial Control System

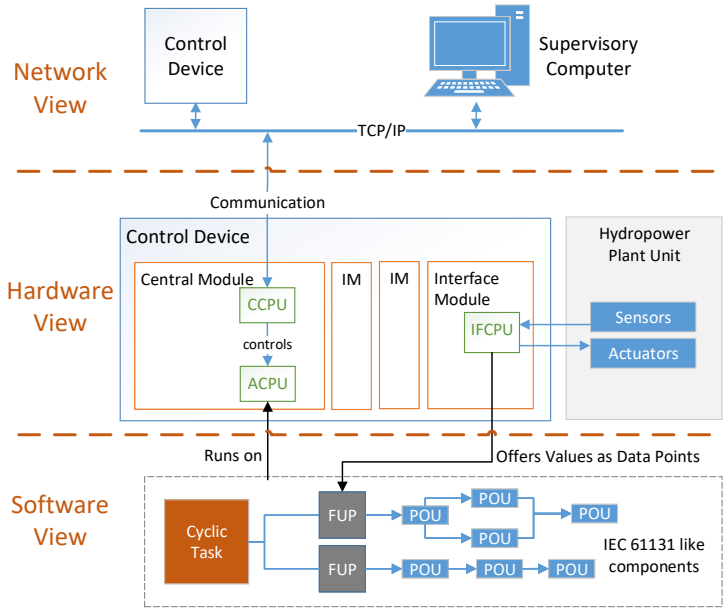


Figure 4.1.: Overview of the target industrial control system.

The context of this thesis are distributed control devices that operate hydropower plant units. Figure 4.1 illustrates a simplified overview of the Supervisory Control And Data Acquisition (SCADA) architecture we were dealing with in our research project.

On network level, control devices are connected via Ethernet and operated by a supervisory system. These supervisory computers are mainly responsible for two things. One

responsibility is to observe the state of physical processes. The other one is to adjust parameters of control devices in order to control the energy conversions. The observation and adjustment actions are done by using so-called data points which are variables with a certain basic data type such as Integer or Boolean.

The control devices are connected to hydropower plant units. Their functional responsibility is to operate these units through one of the four different phases namely excitation, synchronization, protection and turbine control.

Technically, these devices have a Programmable Logic Controller (PLC) architecture. Concerning the hardware design, a control device is built out of one central module and several interface modules. A central module consists of a Communication CPU (CCPU) and an Application CPU (ACPU). The CCPU is responsible for network connections and controlling/monitoring the ACPU. It runs a customized Linux distribution and can be accessed by various protocols such as SSH and Modbus. From the security point of view it protects the ACPU and verifies incoming commands. At the time of writing, the current ACPU is a single-core processor and the next generation will be based on a multi-core processor. The ACPU executes the control logic. It runs a real-time operating system in order to ensure guaranteed cycle times. The interface modules connect the control device with sensors and actuators of the hydropower plant unit. Central modules and interface modules are connected via Ethernet.

The control logic software executed by the ACPU of a central module is component-based and heavily influenced by the IEC 61131 standard for programmable logic controllers [92]. Basically, the control logic is hierarchically build out of components, compositions and tasks. A component is called Program Organization Unit (POU) and a composition is named Function Plan (FUP). POUs are coded with the C-programming language and stored as binaries on the devices. Such POUs implement basic functions, e.g simple logic gates, or complex algorithms. Based on these POUs, reusable FUPs are designed that implement the specific control logic for a hydropower plant unit. Technically, FUPs are serialized as XML files and loaded by a POU scheduler. Finally, such FUPs are called by cyclic tasks, for instance every 10 milliseconds. Again, tasks are serialized as XML files.

FUPs operate on data points that are set and read by the interface modules. At the start of a cyclic task the necessary data points are collected, then the FUPs are executed, and subsequently the calculated data points are written back. The interface modules receive these data points and actuate accordingly. Further, data points are shared with other control devices or supervisory computers over one or several redundant networks. This can either be realized by a network directly between ACPUs or via the CCPU.

4.2. Approach

Figure 4.2 illustrates an overview of our approach. On the left hand side, we find models representing the resources and control logic of the industrial control system. The models

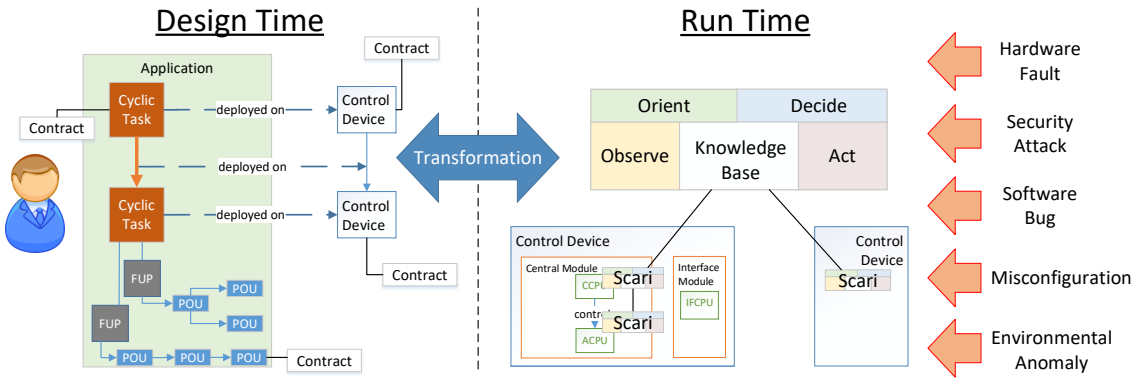


Figure 4.2.: Overview of the presented approach.

about the system resources are gathered from the devices. The control logic and the corresponding deployments are created manually by plant engineers. System configurations are intended to be verifiable concerning functional and non-functional properties. The aim is to ensure that the overall configuration of the industrial control system is correct at design time and the time span of the commissioning is minimized.

On the right hand side, we find a self-adaptive software system that detects anomalies and reacts to them. The intention of this system is to add resilience to the control devices of the control system. The aim is to tolerate hardware faults, defend devices against security attacks, detect software bugs, reveal misconfigurations, and react to faults in the device environment. The self-adaptive system is deployed on each device and organized in a hierarchical way. The models from design time are used for detection, reasoning, and structural adaptation of single control devices and networks.

These two parts are connected by a transformation step illustrated in the middle of Figure 4.2. From design to run time, the control logic is deployed on single devices. The system model is split into parts because control devices only need to know their own cyclic control logic. From run to design time, the system configuration is assembled from these parts.

In the following, we discuss design time, run time, and transformation in more detail.

4.2.1. Design Time

With regards to design time, the goal is to have an abstract representation of the intended control system in order to ensure that the target system is correct in terms of functional properties such as data point compatibility and also concerning non-functional properties such as timing and security.

Our approach is to use domain-specific modeling languages that enable plant engineers to specify the control logic in the form of tasks and FUPs. Note that each control device in

our setting utilizes cyclic tasks. If these control devices are interconnected they exchange data points between cyclic tasks. With our modeling languages it is possible to specify how tasks are deployed and on what central module of a control device.

Figure 4.3 illustrates the four main kinds of domain-specific modeling languages that are used for describing the control logic and an industrial control system.

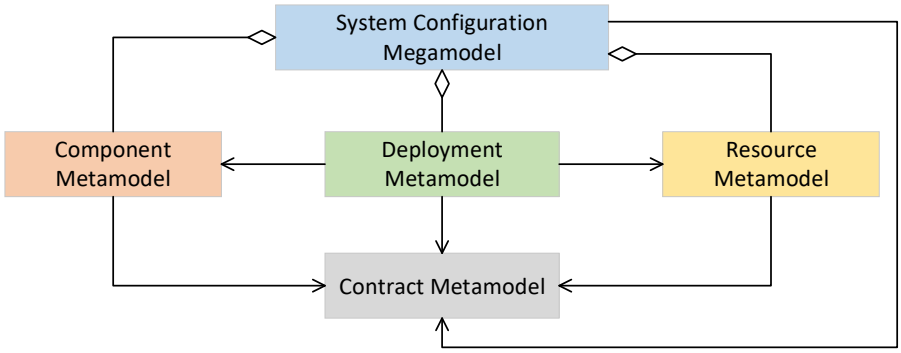


Figure 4.3.: Overview of the design time metamodels.

The *Component Metamodel* is used for specifying control applications, cyclic tasks, FUPs, and the interfaces of POUs. An application is a container modeling the interaction between cyclic tasks. This is necessary because output data points from tasks can be used as input by tasks running on other devices. An application model contains this dependency information. Models based on the *Component Metamodel* are supposed to be manually created by plant engineers.

The *Resource Metamodel* represents devices part of the control system. It includes interface modules, central modules, control devices, hydro-power plant devices, network devices and software applications. These kinds of models are supposed to be created by querying the real networked system and should be managed by the tooling.

The *Deployment Metamodel* is used in order to specify where the control logic defined with the *Component Metamodel* is located at models defined with the *Resource Metamodel*.

The *System Configuration Megamodel* is the entry model for processing a configuration. It refers to the used applications, deployments, and resources. A megamodel is a model containing both the models and the relations between those models [93].

Supporting these modeling languages we use a *Contract Metamodel* for modeling contract-based design. Each entity, such as a task or hardware part, can basically refer to contracts in order to enhance them with arbitrary non-functional properties such as timing, hardware requirements, uncertainty and security. We chose contract-based design because it fits the component-based design of our system. It allows to couple components with assumptions and guarantees in order to verify correct configurations. Single contracts in

our proposed modeling language are dedicated to specific non-functional viewpoints.

Figure 4.2 illustrates on the left hand side how models created with the modeling languages from above can look like. It shows an application containing two tasks where one depends on the other. The dependency is deployed onto a specific physical connection, while the tasks reside on separate control devices. As we can see, each entity can be annotated with contracts.

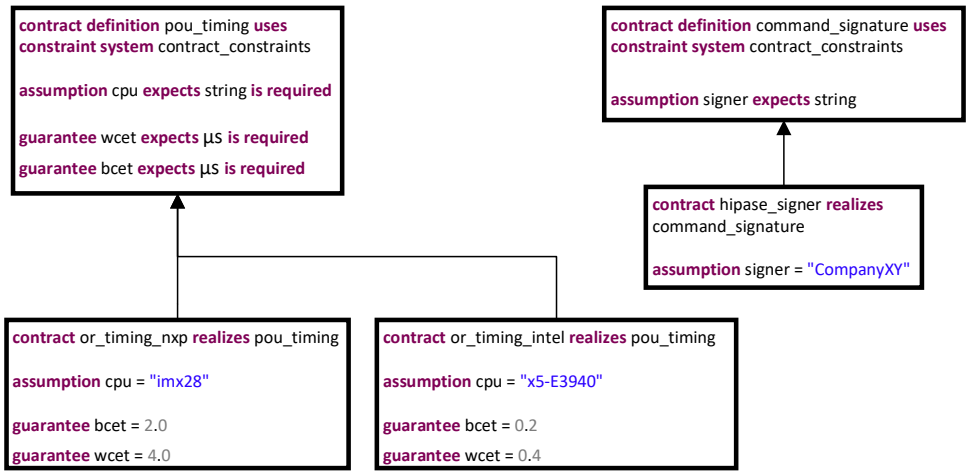


Figure 4.4.: Exemplary contracts.

Figure 4.4 illustrates exemplary contracts for timing and security. We propose contract definitions as type specifications for contracts. In Figure 4.4, the contract definition *pou_timing* specifies what assumptions and guarantees are possible. The contracts *or_timing_nxp* and *or_timing_intel* describe the timing of a the POU *Or* for distinct CPUs. Similarly, the contract definition *command_signature* serves as a type for the contract *hipase_signer*.

In order to support these contracts, we are proposing a configurable constraint language for limiting the design space of contracts. Furthermore, we propose a state machine where each state can contain a different set of contracts. This is useful for describing devices which have distinct operation modes.

Based on these models a system configuration can be verified in distinct ways. One way is structural. In the simplest form a configuration of this kind can be verified as to whether the connected data points between tasks, FUPs, and POUs are compatible. Another simple verification is to check whether connected distributed tasks deal with the same hydropower plant unit or if they are misconfigured.

Utilizing contract-based design enhances these structural checks by adding a flexible mechanism that allows to analyze a configuration from multiple view points. For instance,

a POU interface can be enhanced by a timing contract that guarantees a specific timing based on an assumed CPU, or it states how much memory it needs in order to guarantee its functionality. A task can be enhanced by adding contracts stating that it needs data points within a certain period and that they need to be measured using redundant and independent means. The structural information and the contracts can be processed by tools that are specialized for specific properties. The errors and warnings produced by such tools could be expressed using a separate modeling language that can annotate entities from the metamodels above.

In addition to the verification at design time, the contracts can be utilized as input for run time monitoring and verification.

4.2.2. Run Time

The goal at run time is to have a software system that detects various anomalies and faults, and adapts the control devices in order to increase the reliability and security. We are not aiming to change the control logic itself. Instead, we want to ensure that the hardware and software stack below the intended control logic can perform as long as possible. In order to achieve this, we are reusing the models from design time as an information source and verification input. Furthermore, models at run time reflect the current parametrization and structure of the devices.

Our overall aim is to detect and adapt to anomalies and faults from five distinct areas:

- **Hardware Fault**: As an example, permanent memory cell faults in RAM or CPU registers, used by the ACPU or CCPU, can be detected with memory checks. After such a detection, a faulty location could be circumvented by reconfiguration of the operating system or by moving the control logic to another module or device. Further, permanent hardware faults in interface modules could be recognized and handled by using an alternative interface module. A control device should not only be able to recognize its own faulty hardware but also that of others. As data points are distributed to other control devices controlling other parts of the same hydropower plant unit, they should be able to observe and analyze that something might be wrong with the hardware of other control devices or networking devices. Ultimately, the control logic running on one device could be migrated to alternative ACPUs, central modules or devices.

- **Security Attack**: Each control device in our setting knows from whom it receives or sends data, based on its configuration. This information could be used for detecting network security attacks or attackers that imitate control devices. Infected devices can be detected when the data points they are distributing suddenly develop odd behavior or do so over time and do not reflect the real environment. Additionally, unexpected behavior of devices can hint to security incidents, e.g., attempting access to control devices which they are not supposed to. Revealed attacks can

be handled by blocking and isolating infected devices or network resources. Other kinds of attacks are for instance attempts to access restricted resources or physical manipulation of sensors.

- **Software Bug**: Software running on top of modules can be updated in order to gain new features or fix bugs. Albeit being a very useful feature to be able to update the software, these changes may introduce new bugs. Resource and performance monitors, which observe the behavior of tasks could be used to detect changed and different behavior patterns.

- **Misconfiguration**: Although our approach verifies the control logic during run time concerning various aspects, it is still possible that for instance data points are not transmitted between devices in time. Thus, monitors observing the behavior of tasks, interface modules, etc. could detect faults in the configuration of the control logic.

- **Environmental Anomaly**: The sensors and actuators, which are the connection between the PLCs and the power plant environment (e.g., the water turbines) can break or drift over time. Detecting such environmental anomalies and reacting to them is also an important area in order to make a control system more reliable.

Detection and adaptation for these areas could be done by implementing separate mechanisms specialized to single faults with corresponding adaptation methods. However, this involves the orchestration of these mechanisms in parallel and ensuring that adaption mechanisms do not interfere with each other. Furthermore, some anomalies are cross-cutting. For instance, if a drift of a data point on a device is observed, this could indicate a hardware fault or a security attack. If a hardware fault and a security mechanism reacted the same time, it could be fatal for the hydropower plant. We are thus proposing an infrastructure that allows orchestration of such diverse detection and adaption mechanisms and ensures that only one is carried out at one point in time. We name our infrastructure **Scari** (**Sec**ure **a**nd **r**eliable **i**nfrastructure).

Figure 4.5 illustrates the adaptive loop of Scari. It consists of 5 parts, which are *Observe*, *Orient*, *Decide*, *Act*, and a common *Knowledge Base*.

The *Observe* part consists of *monitors* that are specialized in discovering and measuring specific anomalies, for instance a drift in data or a security-related violation. These monitors notify an arbitrary number of interested *syndrome processors*, residing in the *Orient* part. The *syndrome processors* are implementing a specific detection mechanism, e.g., for hardware faults or security attacks. Technically, they receive notifications about a situation and react on them based on rules or more advanced mechanisms. If one or several *syndrome processors* diagnose a problem, they recommend plan types for handling or analyzing a situation. For instance, based on the received notifications, a hardware

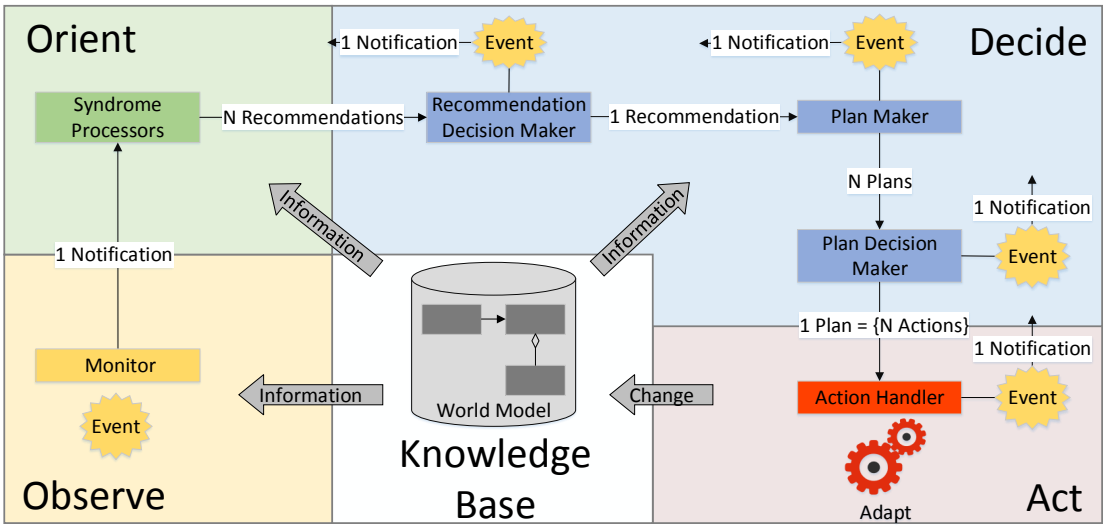


Figure 4.5.: Scari loop.

fault *syndrome processor* may recommend circumventing a damaged module, while a security *syndrome processor* may recommend isolating a device. Next, the *recommendation decision maker* selects the best recommendation on the basis of a definable prioritization for the covered events and chosen plan types. The selected recommendation is then forwarded to the *plan maker* that creates the actions for the plan type. Some plan types may be realized in different ways. We thus added a *plan decision maker* that selects the plan with the least affected Scari instances and the lowest number of used actions. In the final part, the plan is executed by an *action handler*, and the system is adapted to the situation diagnosed by a *syndrome processor*. Each of the entities of the *Decide* and *Act* parts feeds back its states as events. This enables the *syndrome processors* to log the state of their recommendations and to be notified in turn that the system has adapted. The *knowledge base* provides support for the other four parts. It contains the deployed models, including contracts, from design time and additional run time information. It serves as a source of knowledge for the *Observe*, *Orient* and *Decide* parts, while the *Act* part stores the executed changes of the system there.

As depicted in Figure 4.6, Scari instances run on the ACPU and CCPU of each central module and on different layers above. Each of these instances incorporates the five parts of the Scari adaptive loop and can react autonomously. We chose a decentralized architecture because the control devices in our industrial setting can be deployed in unique system configurations incorporating devices from other companies. The Scari instances can be organized in layers in order to escalate situations to entities with more adaptation options. As illustrated in Figure 4.6 notifications about events and system knowledge are sent

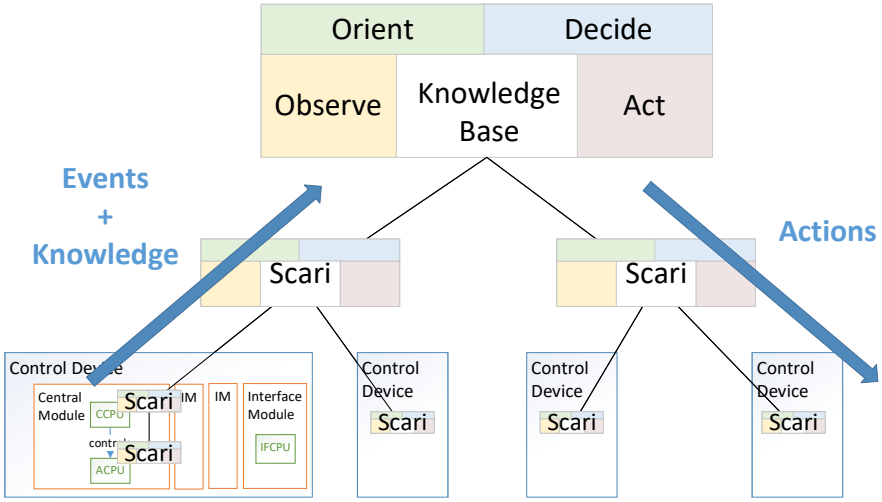


Figure 4.6.: Scari deployed in layers.

upwards to higher layers while actions from higher layers can change lower layers. This hierarchical organization of Scari has two advantages:

One is that a knowledge base only needs to *know its subgraph*. If the knowledge within a Scari instance changes, information is only propagated up to the parent nodes. Distributing all information on all available Scari instances would additionally lead to more network traffic.

The second advantage is that an adaptive loop only needs to *handle its subgraph*. A loop does not need to manage other parts of the overall control system, which also eases the configuration of Scari. If it is not possible to adapt to an event occurring on a node, it can be escalated to a parent node that has more knowledge, more resources, and can therefore leverage more powerful adaptation mechanisms. In our hydropower setting it is conceivable that these adaptation layers are even laid over different hydropower plants, where they are acting on a greater time scale.

4.2.3. Transformation

Figure 4.7 showcases models at design and run time. At design time the deployment of tasks to resources is explicitly modeled. At run time the tasks are already deployed and an application is split into distinct PLC configurations. Objects of type *Me* are referring to the specific device models at run time. Scari organizes models hierarchically and distinct modeling languages can be utilized by the different layers.

Design time provides to run time system knowledge, which can be utilized by the distinct participants of Scari. For instance, monitors can utilize contracts for observing performance behavior. Also syndrome processors can configure themselves based on the

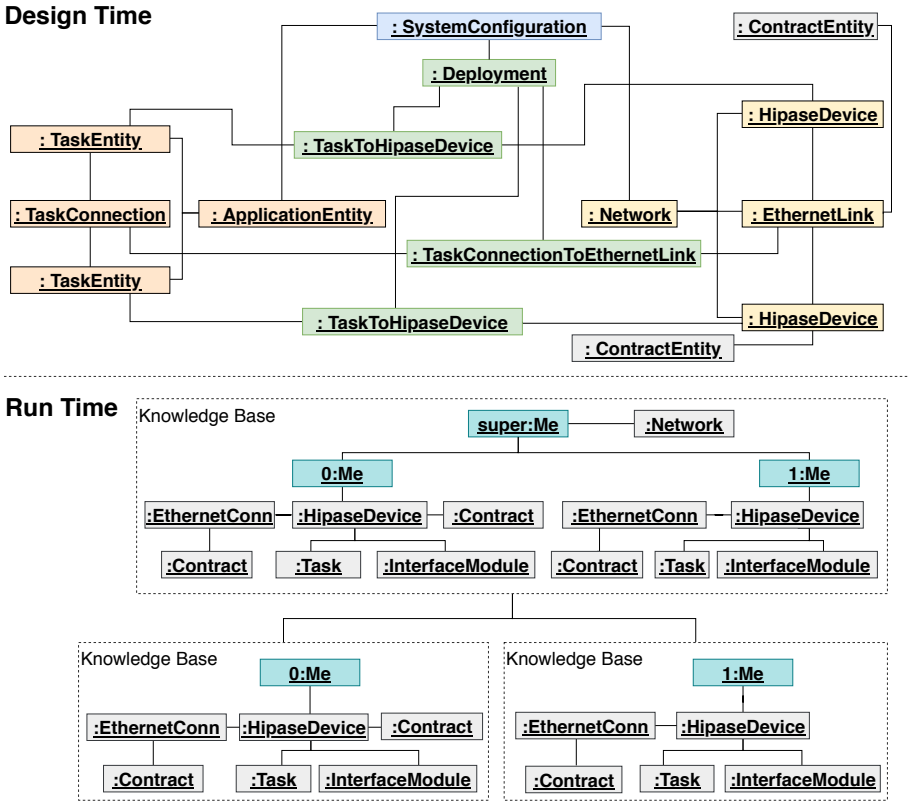


Figure 4.7.: Models at design and run time.

provided system knowledge. Furthermore, a plan maker could utilize the design time verification mechanisms for verifying generated plans.

From run to design time, the system knowledge gets updated and adjusted to real world behavior. For instance, contract and model violations can be observed. Scari can adapt the system knowledge in order to update contracts e.g., concerning performance or reliability.

The transformation of design-time models to run-time models can reside on a supervisory computer or computers used during commissioning. The generated run-time models are deployed by using the action handlers, which set the corresponding knowledge bases or by bootstrapping a Scari instance. The first way is preferable if the knowledge base already contains information.

The transformation of run-time models to design-time models works by cloning a higher layer knowledge base to a computer where the run-time models are transformed to a design-time configuration reflecting the current system configuration. It is only necessary to clone a higher layer knowledge base because it incorporates the knowledge from lower layers.

4.3. Use Case

We demonstrate using the following examples how models at design time look like and what our proposed self-adaptive system is able to do at run time.

4.3.1. Design Time

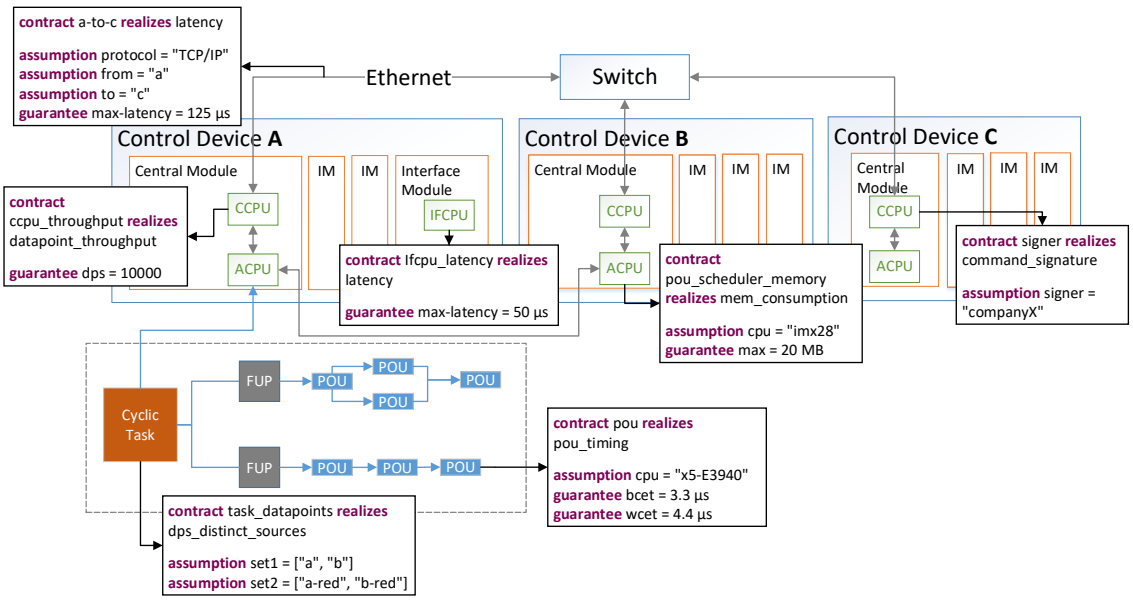


Figure 4.8.: Exemplary design time use case.

Figure 4.8 illustrates a visualization of models at design time, which could be specified with our proposed modeling languages. A cyclic task with two FUPs is deployed on the central module of control device *A*. Data points are received from the *ACPU* of control device *B* physically over an Ethernet connection. Control device *C* is connected to the same hydropower unit but the *ACPU* is not connected with the others.

Figure 4.8 shows distinct kinds of contracts describing non-functional properties of hardware and software. The shown non-functional properties are timing, latency, redundancy, throughput, memory consumption, and security. The contracts describing timing, throughput, memory consumption and the latency contract *ifcpu_latency* would be captured during the development and production of the hardware and software components through testing. If the software of the PLCs gets updated at run time these contracts would need to be updated as well. The contract *task_datapoints* describes that data points contained in *set1* and *set2* need to be provided by distinct sources. This constraint must be fulfilled by the design time model and at run time. The latency contract *a-to-c* must be

measured during the commissioning of such a system. The security contract *signer* states that only commands signed with the specified certificate are allowed to be accepted. "companyX" would point to a specific public key.

In order to verify a model of this kind, it can be transformed into data consumable by specialized tools. For instance, timing analysis software could calculate the worst case, average case and best-case behavior of task chains, taking the Ethernet networks into account. The results would then be presented to the engineer for optimizing the system configuration.

If the various verifications are successful the design-time models can be transformed and deployed to the PLCs. In our approach, this information is preserved as architectural models at run time for detecting anomalies and adapting to situations. The contracts can play an important role for the detection. Furthermore, the contracts can be utilized for verification of planned adaptations for ensuring their correctness.

4.3.2. Run Time

In the following we discuss three use cases at run time where Scari can be applied for dealing with a problematic situation. The models from design time support the different Scari activities in adapting to a problematic situation.

Memory Fault ACPU

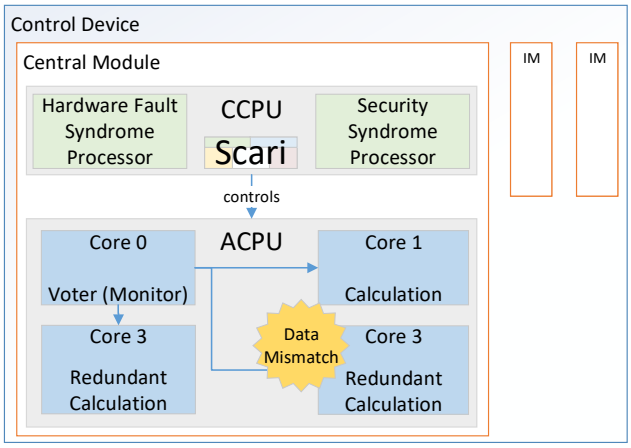


Figure 4.9.: Exemplary memory fault on the ACPU.

Figure 4.9 shows a hardware fault residing on the ACPU. In this scenario one calculation is running redundant on core 1, 2 and 3 of the ACPU. All calculations are observed by a

voter (monitor). Now, if a data mismatch happens, the voter notifies the syndrome processors residing on the CCPU. These syndrome processors are implementing the logic for processing notifications and recommending a suitable plan. In our experiments we implemented the syndrome processors as rule-based engines, state machines or time-triggered tasks.

Situation	**Plan**	**Actions**
Emergency	Safe State	• Activate Hot-Standby PLC • Stop PLC • Notify Higher Layer
Data Mismatch	Memory Test	• Stop Calculation on Core • Perform Memory Test
Located Faulty Memory Area	Mask Memory	• Switch Memory Area • Mask Faulty Memory Area
Repaired Memory	Start Calculation	• Start Calculation on Core 3 • Reconfigure Voter

Table 4.1.: Plans for dealing with a permanent memory fault.

Table 4.1 illustrates the plans which could be recommended by the hardware fault syndrome processor. The plan "Safe State" can be recommended if there is no possibility of repairing a system. It could involve activating a hot-standby PLC. However, if the data mismatch is only affecting the calculation on core 3, the hardware fault syndrome processor can recommend testing the used memory area. If the memory test finds a faulty area, the syndrome processor can recommend to switch the memory area and to permanently mask the memory area by configuring the operating system. After a new successful memory test, the hardware fault syndrome processor can recommend the plan "Start Calculation".

This example shows that a PLC is able to heal itself and to continue the control activities with Scari.

Hardware Fault Interface Module

Figure 4.10 illustrates the three control devices from design time. It shows that a permanent hardware fault is happening on an interface module of the control device *B*. The hardware fault manifests itself through missing data at the connected ACPUs. At the same time, control device *C* would obtain the same data from an interface module.

Table 4.2 enumerates plans for dealing with this situation. If control device *B* is connected with a hot-standby device it could activate it and stop itself. If it obtains a redundant interface module it could recommend the plan "Switch Module". On the network layer, the higher-layer Scari instance could recommend to circumvent the faulty interface

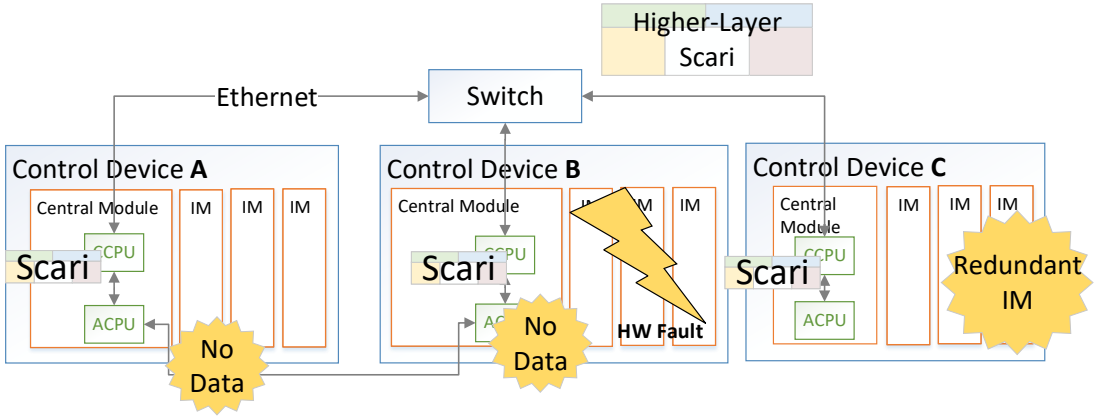


Figure 4.10.: Exemplary hardware fault of an interface module.

Situation	Plan	Actions
Emergency	Safe State	• Activate Hot-Standby PLC • Stop PLC • Notify Higher Layer
HW Fault in Interface Module	Switch Module	• Use Data from Redundant Module • Stop Faulty Interface Module
HW Fault in Interface Module of Child PLC	Reconfigure Data Point Distribution	• Stop Hydropower Unit • Add Data Connections • Reconfigure FUPs • Start Hydropower Unit

Table 4.2.: Plans for dealing with a hardware fault affecting an interface module.

module with a reconfiguration of the data point distribution. The timing contracts can be used during plan creation in order to ensure that data points arrive in time.

Note that the proposed self-adaptive software system does not react in real time. We are assuming that the first reaction in this case would be done by the PLC software itself. Scari would then try to recommend one of the plans in Table 4.2 for the system self-healing.

Manipulated FUP

Figure 4.11 illustrates an overview of a security-related use case. In this example, an attacker was able to deploy a hostile cyclic task to control device *B*. This hostile task may target to slowly destroy a hydropower plant unit, similar to the intention of the famous Stuxnet computer worm [94, 95]. Monitors and syndrome processors located on the CCPU of control devices and connected to the same hydropower plant unit have the means to

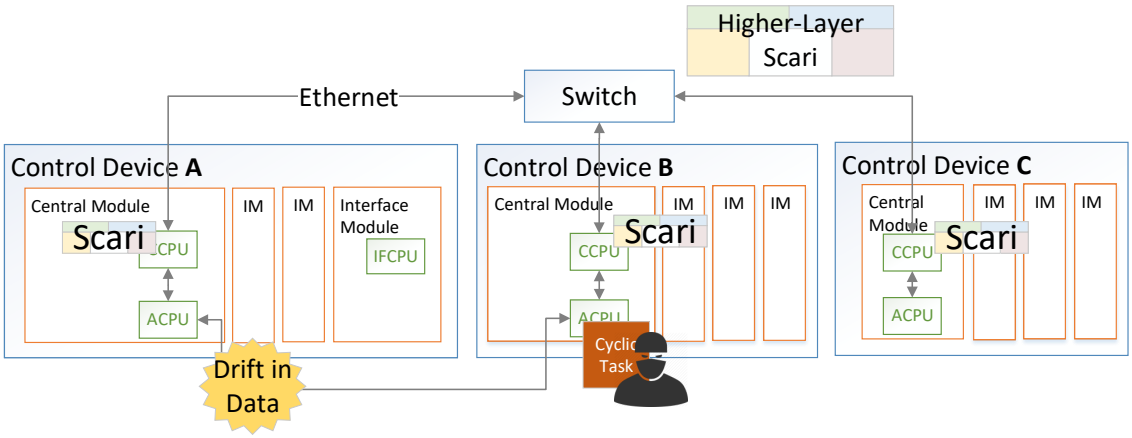


Figure 4.11.: Exemplary control devices under attack.

detect the slow but harmful drift of data points.

Situation	Plan	Actions
Drift in Data	Notify Higher Layer	• Send Notification
Drift in Data	Isolate Device	• Stop Hydropower Unit • Blacklist Device
Isolated Device	Reconfigure Data Point Distribution	• Add Data Connections • Reconfigure FUPs • Start Hydropower Unit

Table 4.3.: Plans for dealing with the security use case.

Table 4.3 enumerates possible plans in such a situation. Control device *A* could react by notifying the higher Scari instance that there is a drift in the data sent from control device *B*. A security syndrome processor located in the higher layer could react by recommending the plan "Isolate Device". If this plan was successful, the syndrome processor could recommend the plan "Reconfigure Data Point Distribution" in order to restore the availability of the hydropower unit.

In the presented example we knew beforehand that it is a security-related situation. Syndrome processors have to analyze notifications and recommend appropriate plans. It might also be the case that there are syndrome processors, which are dedicated to diagnosing hardware faults and they interpret the situation differently. This is a benefit of Scari as it provides a common infrastructure for all kinds of distinct reasoning mechanisms. It allows to compete syndrome processors for the best recommendation. In essence, this is the reason why it is important to prioritize events and plan types in order to make a de-

cision. For instance, one could prioritize security-related events over hardware-related. It depends on the domain and the available means how events and plan types are prioritized.

5. Design Time

This Chapter is devoted to the phase starting with the development of a hardware or software component and ending with the commissioning of a control system. We propose modeling languages for describing the available hardware and software resources, the control (application) logic and the deployment of the control logic (**C1**). Additionally, we propose a modeling language for specifying contracts and contract types based on the contract-based design paradigm. It also incorporates a state machine for contracts. We use this contract-based design modeling language for capturing the behavior of non-functional properties. From our work on modeling languages, we derived four design patterns that describe how to design configurability into domain-specific language elements (**C4**).

In Section 5.1, we motivate our work by enumerating our goals concerning design time. Section 5.2 discusses requirements to the modeling languages that are derived from our motivation. Section 5.3 presents our modeling languages. Section 5.4 explains how we implemented the modeling languages technically. Section 5.5 showcases how the modeling languages can be utilized. In Section 5.6, we explain how our modeling languages fulfill the requirements discussed in Section 5.2. Section 5.7 enumerates the limitations of our approach. Finally, Section 5.8 enumerates design patterns that we found during our work on modeling languages.

5.1. Motivation

At the time of writing, the PLCs of our project partner are programmed individually with graphical function block diagrams. These diagrams comply with IEC 61131-3 [2]. IEC 61131-3 specifies a set of standardized POUs. This set is extended with hydropower specific ones. A PLC in our setting executes tasks periodically for instance every 10 milliseconds. This is also true if there exist interconnections between control devices. Thus, it is crucial that a PLC finishes a task and transmits data points to a second PLC in time, before the second PLC restarts the receiving task.

There are two goals we pursue with our work at design time. The first goal is to have models of the control devices and the deployed tasks in order to verify whether a system configuration is correct concerning functional and non-functional properties. At the time of writing, a correct system configuration is accomplished by trying out. An automated verification would simplify the commissioning of control devices. For instance, if the worst-case and best-case execution time of the single POUs, hardware modules and

bus connections are known through testing, it could be estimated beforehand if a data point will be sent between two control devices in time.

The second goal is to annotate and use these models at run time for enforcing and verifying constraints and assumptions. For instance, a constraint at run time may be that two input data points for a cyclic task are acquired and pre-processed from distinct sensors and hydropower equipment. As security increasingly becomes important in the hydropower domain, another constraint could be that commands are signed by a fixed set of SCADA entities. The constraints themselves are supposed to be specified at design time.

These two goals are also aligned with our research questions. Regarding the first research question, we determine what kind of system knowledge is required for creating models for verification. Concerning the second research question, the modeled system knowledge provides means for evaluating options for actions in order to add resilience.

5.2. Requirements

In the following, we discuss requirements derived from the motivation above. Requirements are marked with a "REQ" plus a number in square brackets.

Models for representing an industrial control system need to reflect the *tasks and FUPs [REQ1]* in order to grasp the control logic. Furthermore, it is necessary to describe *interface modules and available data points [REQ2]* for relating the tasks to the physical environment. The control device of our industrial setting can be deployed in clusters for realizing parallel functionalities. Therefore, it is also necessary to model the *interconnections of control devices and tasks [REQ3].* An engineer should be able to verify tasks in distinct scenarios. Therefore, a modeling language should support a *loose coupling between task and control device [REQ4].* Last but not least, we want to express assumptions and constraints for instance about security, timing, or performance. Further on, these constraints should be verifiable by specialized tools. Thus, an approach should provide a *generic modeling concept for non-functional properties [REQ5].*

5.3. Modeling Languages

Figure 5.1 provides an overview of all proposed design time modeling languages.

The *System Configuration Megamodel* is the entry model for processing a configuration. It refers to the used applications, deployments, and resources. A megamodel is a model containing both - the models and the relations between them [93].

The *Component Metamodel* is used for specifying control applications, cyclic tasks, FUPs, and the interfaces of POUs. An application is a container modeling the interaction between cyclic tasks. This is necessary because output data points from tasks can be used as input by tasks running on other devices. An application model contains this dependency

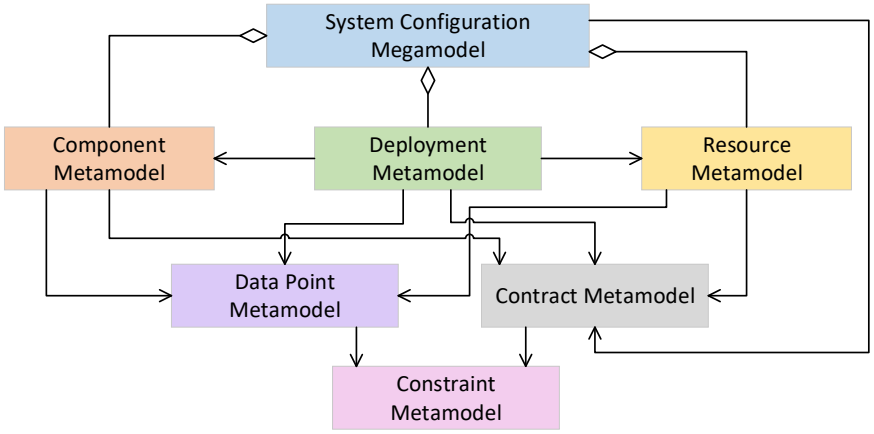


Figure 5.1.: Overview of all design time metamodels.

information. Models based on the *Component Metamodel* are supposed to be manually created by plant engineers.

The *Resource Metamodel* represents devices part of the control system. It includes interface modules, central modules, control devices, hydro-power plant devices, network devices and software applications. This kind of model is supposed to be created by querying the real networked system and should be managed by the tooling.

The *Deployment Metamodel* is used in order to specify where the control logic defined with the *Component Metamodel* is located in models defined with the *Resource Metamodel*.

The *Data Point Metamodel* provides modeling concepts for defining data-point types and values. It is used by the *Component*, *Resource* and *Deployment* metamodels.

The *Contract Metamodel* offers means to define contract definitions, contracts, contract state machine definitions and contract state machines. These mechanisms capture the non-functional behavior of resources, components, and deployments.

The *Constraint Metamodel* is used for defining data types, variables and constraints. It is the foundation of the *Data Point* and *Contract* modeling languages.

Figure 5.2 presents a model as it would be defined with our approach. The right hand side shows resources such as Hipase devices, their modules and Ethernet connections between devices. This information should be provided by the existing devices in a hydropower plant and automatically discovered at the time an engineer configures a plant. The left-hand side shows the control logic consisting of an application, tasks, and FUPs. Between these two models are the deployment objects that link control logic with resources. The system configuration object at the top of Figure 5.2 serves as entry point to the models. In addition to these models, contract entities and definitions are shown that specify non-functional properties. They can be attached to each object in the model.

Figure 5.3 shows how a contract is defined with our proposed modeling languages. The

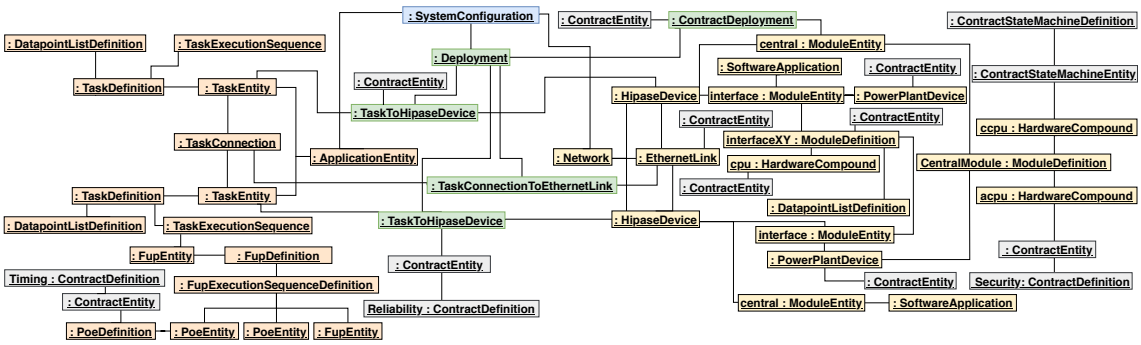


Figure 5.2.: Example model containing resources, components, deployments and contracts.

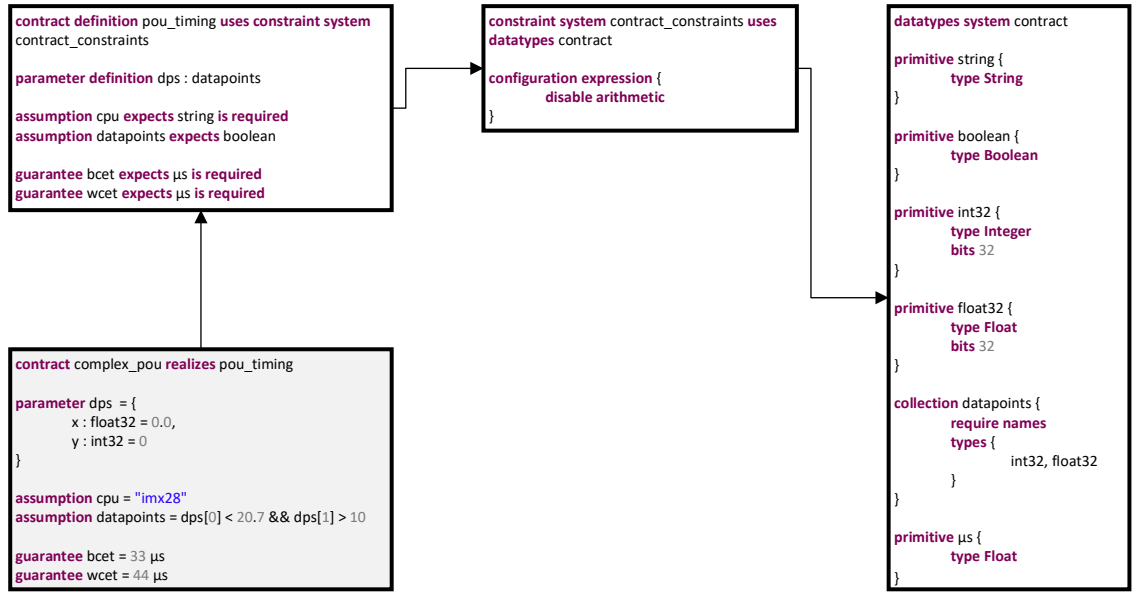


Figure 5.3.: Exemplary contract, contract definition, constraint system and data type system.

data-type system on the right-hand side illustrates the available data types for the contract. A collection of type *datapoints* holds the single data points which have to be specified with a name and type. The illustrated constraint system references the data type system and disables the possibility of arithmetic operations within the assumptions and guarantees. This is useful if a contract is transformed to a tool, which verifies for instance the timing. A constraint system can configure each aspect of our proposed constraint language. Figure 5.3 illustrates a contract definition for the timing of POUs. It defines a parameter *dps* of type *datapoints*, assumptions concerning CPU and data points, and guarantees for best

and worst case execution times. The assumption and guarantee definitions specify the expected return type of the constraints and whether an instantiation is required. The contract *complex_pou* realizes the contract definition. It specifies two data points; an assumption targeting the CPU and a constraint referencing the parameter *dps* with the contained data points. If the assumptions are fulfilled the contract guarantees best and worst execution times.

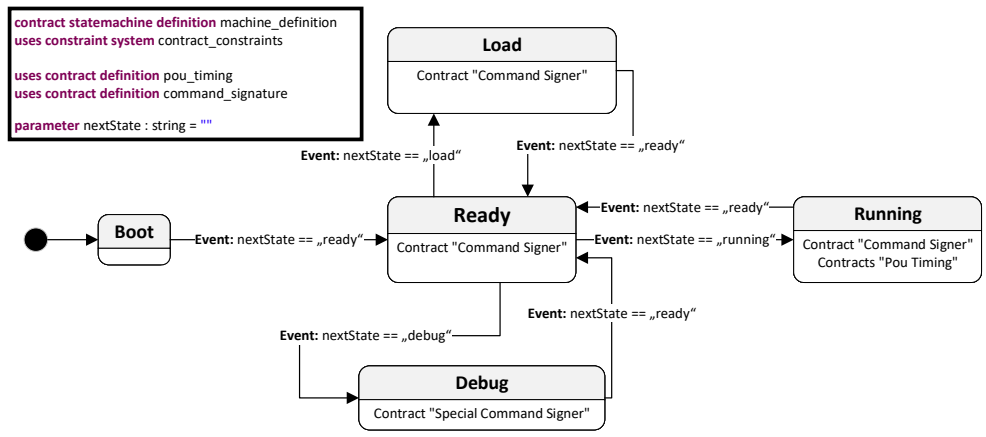


Figure 5.4.: Exemplary contract state machine for a PLC.

Figure 5.4 shows an exemplary contract state machine for a PLC. Depending on the state, a different set of contracts becomes valid. For instance, certificate signatures of other participants are accepted if the contract “Special Command Signer” becomes valid in the state *Debug*. Also, there is only a POU timing provided if the control device is in the state *Running*. The contract-state-machine definition *machine_definition* names available contract definitions and parameters for triggering the state changes. Similar to contracts, a contract state machine must comply with the definition.

Here we enumerate possible non-functional properties graspable by contracts in our industrial setting:

Timing Contracts can be used for describing the worst, average, and best case timing behavior of interface modules, devices, network connections, and software applications. Depending on the described component, a timing contract could state assumptions about the CPU, operating system, processed data points, and workload.

Memory usage Contracts can capture the memory usage of software applications in order to ensure that a device is capable of executing them. This is useful because the available memory on the PLC of our project partner depends on the number of processed data points.

Security Contracts can be used for specifying run-time constraints, e.g., which certificate needs to be used for sending commands to a PLC. Similar to this, a contract could state what applications can be executed on a communication partner in order to be trusted. Furthermore, contracts could be used for specifying trust boundaries.

Redundancy Concerning run time, a contract can state for instance that a data point needs to be provided by distinct modules and sensors.

Throughput Contracts could state the amount of data points a component can transmit per time slot. For instance, the CCPU transmits data points between the supervisory computer and the ACPU. A contract guaranteeing a certain amount of data points per time unit could help optimizing the supervisory computer.

In the following, we discuss the responsibility, rationale, structure and behavior of the metamodels illustrated in Figure 5.1. Note that we chose textual DSLs as concrete syntaxes of the modeling languages. Therefore, objects are often not connected directly with each other but only through a specific reference object. Furthermore, a lot of the semantics is realized by the textual languages.

We verified the correctness of our proposed modeling languages by transforming existing PLC configurations from our project partner into models. We then generated equal configuration files from these models. Therefore, we conclude that our modeling languages preserve the contained information.

5.3.1. Constraint

Responsibility: The constraint modeling language is used for specifying data types, variables and expressions. Furthermore, it enables to specify data type and constraint systems. A data type system defines available data types and their possible conversions. A constraint system restricts what expression statements and variables are allowed to be used. This modeling language is reused by the contract and data point modeling languages.
Rationale: The specification of data types is important for ensuring that data is semantically correct. We designed our own constraint language in order to make it configurable concerning available operators in expressions. We use this feature for specifying contract types that are transformable to input data for third-party verification tools.
Structure & Behavior: Figure 5.5 shows the classes for specifying data types, variables, signatures of method calls, concrete values, and expression statements. Primitive data types refer to built-in basic types such as string or integer. Our proposed constraint language also allows to specify valid data type conversions from and to data types. This enables to cast variables to other data types. The data type conversion ensures that a cast is semantically correct. An object of type *Signature* is specifying the signature of a method that can be called by an expression. This is useful for code generators that interpret such signatures. We did not add the possibility of specifying a method body because signatures merely serve as references to methods realized by an interpreter or in generated code.

Figure 5.6 shows the distinct operations possible. Each operator inherits from the class *IExpression*. Semantically, each operator returns a valid data type and can therefore be used within assumptions or guarantees, which always expect a certain data type. In the case of the comparison and logical operators, our constraint language internally returns a primitive boolean type.

Figure 5.7 illustrates the classes for configuring the constraint language. A *Variable Configuration* specifies variable-related settings such as disabling the possibility to specify concrete values. As mentioned above, objects of type *Signature* can be used for specifying methods known by generators. The *Expression Configuration* is used for a fine-grained control over expressions. The *Data Type System* offers the possibility to package data types and is referenced by the *Constraint System*.

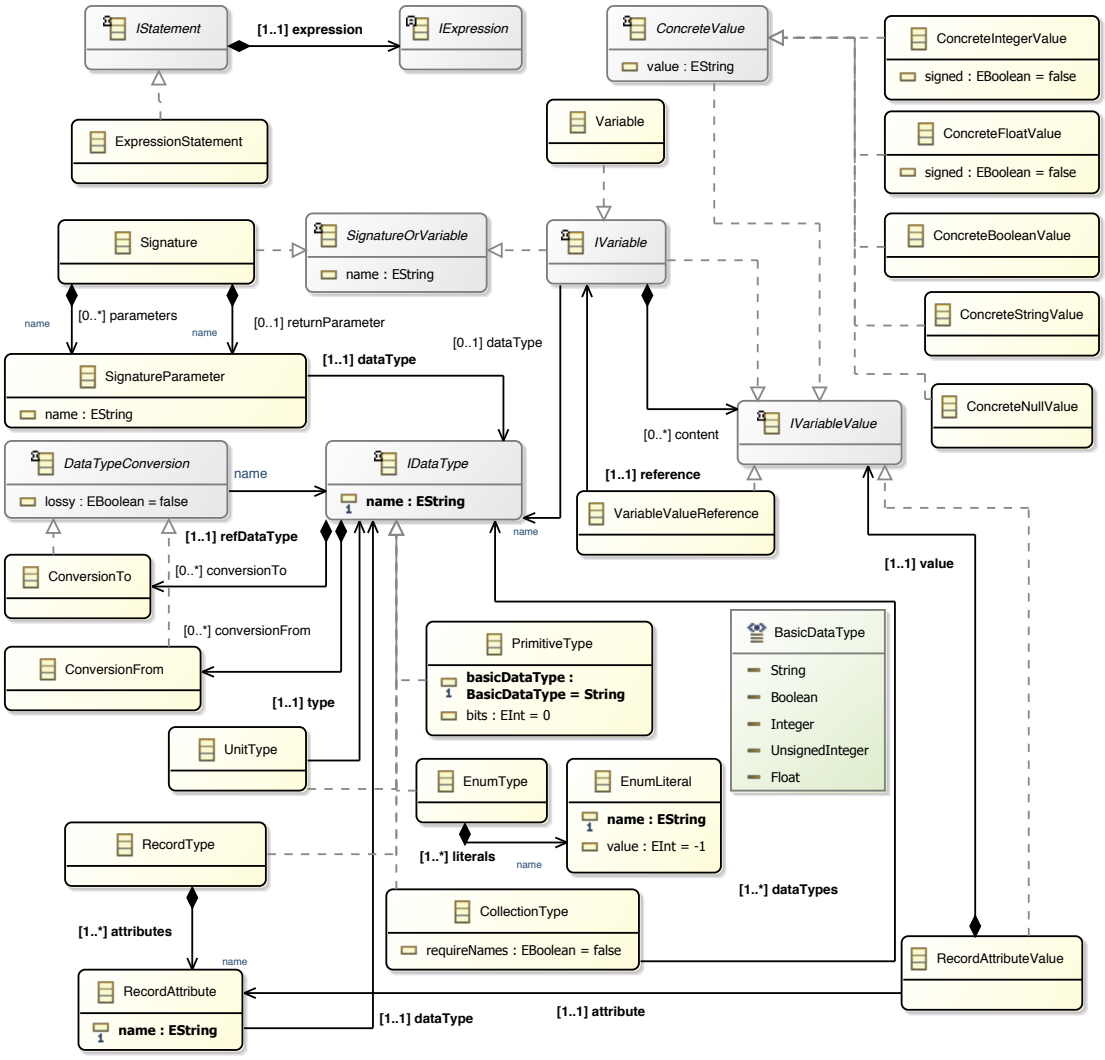


Figure 5.5.: Metamodel for data types, variables, expression statements, and data type conversions.

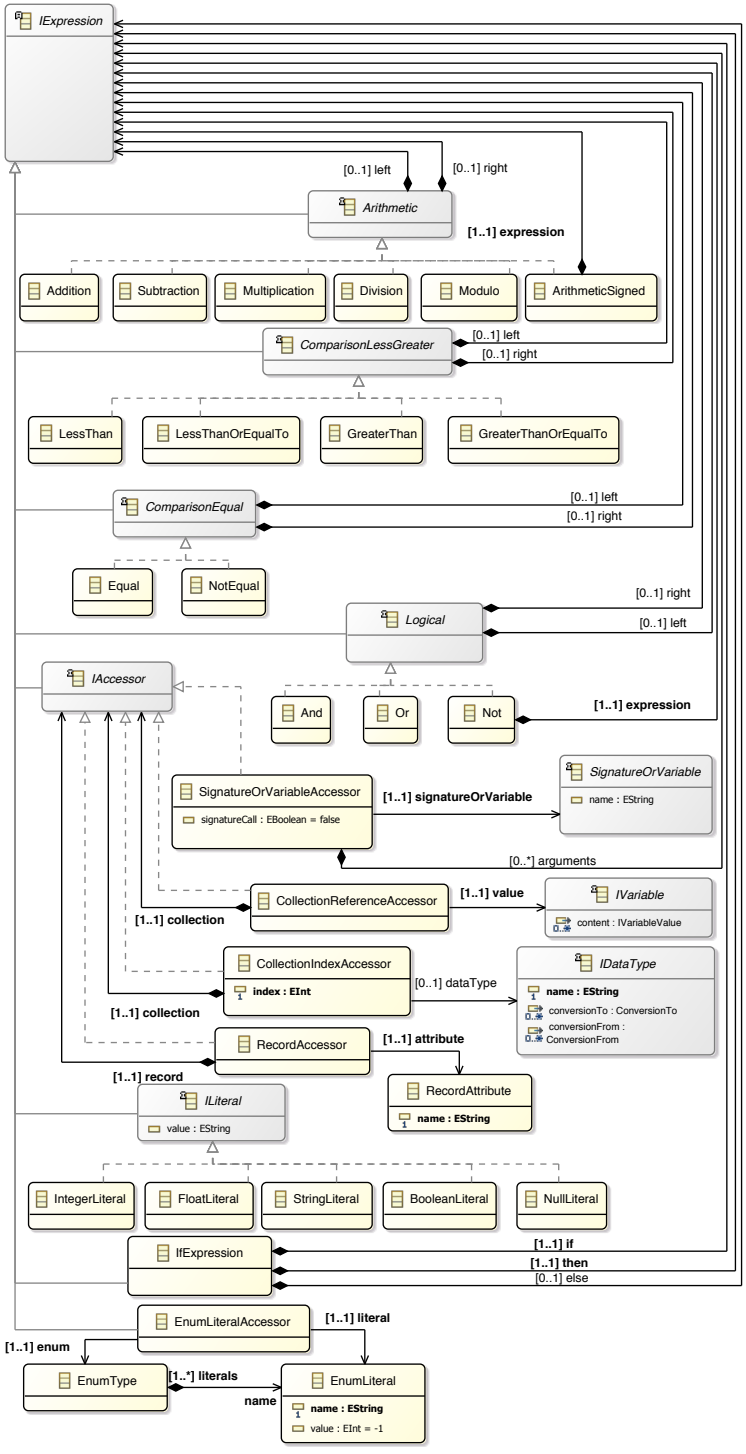


Figure 5.6.: Metamodel for expressions.

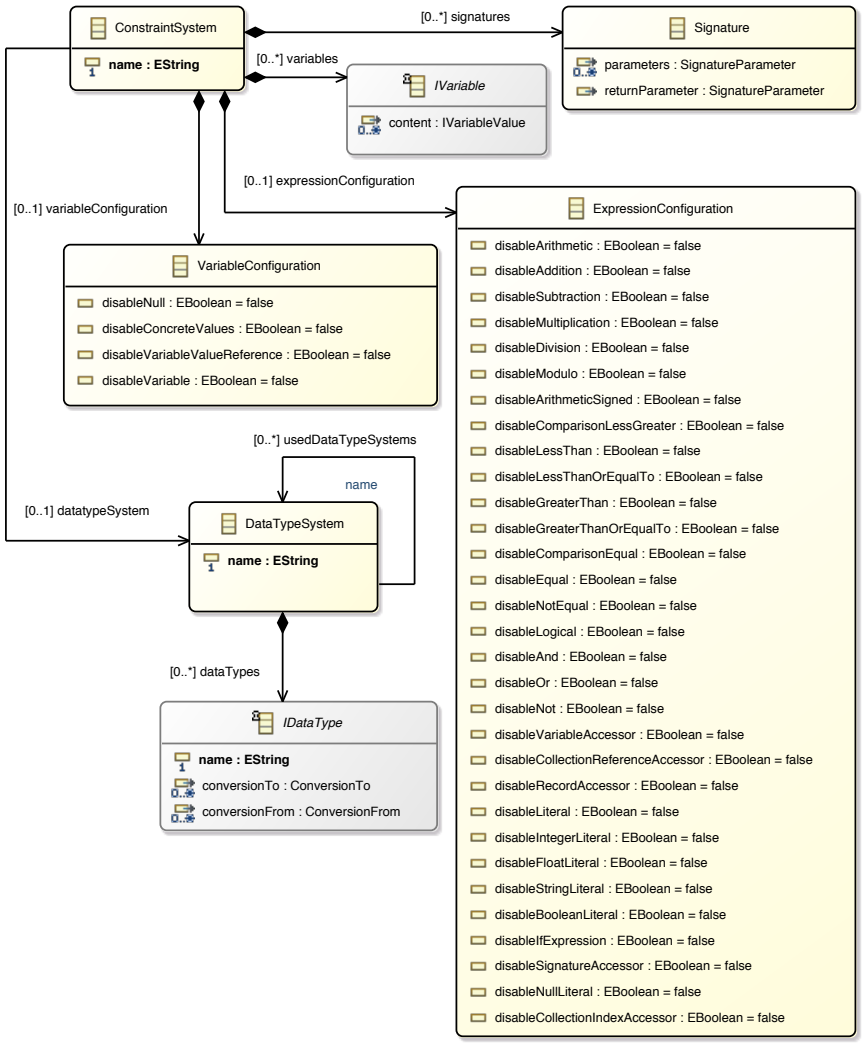


Figure 5.7.: Metamodel for configuring the constraint modeling language.

5.3.2. Contract

Responsibility: The contract modeling language is used for specifying types of contracts and contract state machines. Furthermore, instances of contracts and contract state machines can be defined.
Rationale: Contract-based design is our chosen approach for capturing non-functional requirements and properties of distinct entities. It enables to capture what an entity assumes from the environment and what it in return guarantees. We introduced the concept of contract types for specifying semantically how a contract instance can look like. This is important for generating correct input for arbitrary analysis tools. We introduce the concept of a finite state machine, where single states constitute valid contracts because there can exist cases where the behavior of an entity, including non-functional properties, changes over time or as a result of specific events. The idea is to have finite state machines, where the single states may contain several currently valid contracts. A state machine itself operates on parameters provided by the environment or the internal states of an entity. If a contract state machine changes its current state, former contained contracts can become invalid or overwritten.
Structure & Behavior: Figure 5.8 illustrates our proposed metamodel for contracts. We separate a contract into two parts. A *Contract Definition* represents a type for *Contract Entities*. It states the available parameters, assumptions and guarantees. Furthermore, it represents the target non-functional property. A *Contract Entity* captures the unique behavior concerning the target non-functional property of a component in relationship to its environment.

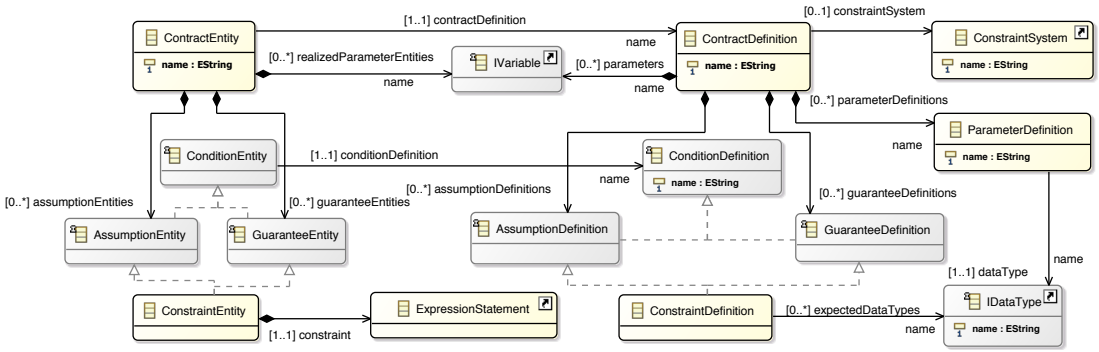


Figure 5.8.: Metamodel for contracts.

Parameters can represent properties of the execution environment, data ports or events. They can be used by *Constraint Entities* in order to set the specific assumption or guarantee. *Parameter Definitions* are used to specify that a variable of a specific data type may exist, but the concrete value has to be defined by the realizing *Contract Entity*. This

can be useful for data arrays where the data points contained are individual for each component.

In the context of assumptions and guarantees, it is possible for a *Constraint Definition* to set expected data types. The associated *Constraint Entity* must provide an expression where the resulting data type equals one of the expected types.

As we can see in Figure 5.8, we use from the constraint modeling language *IVariable* for parameters, *IDataType* for data types, and *Expression Statement* for constraint expressions. The expressions are configured by the *Constraint System* referenced by the *Contract Definition*.

The assumptions and guarantees of the *Contract Entities* must be either automatically gathered by a measurement software or issued by humans. The referenced *Contract Definition* can be used by tools in order to distinguish different kinds of contracts.

Single contracts are sometimes not adequate for representing non-functional properties. There are cases where the behavior of a component, including non-functional properties, changes over time or as a result of specific events. We thus decided to expand the theory of contract-based design and capture such differences concerning contracts by applying the concept of a finite state machine. The idea is to have a finite state machine, where the single states may contain several currently valid contracts. The state machine itself operates on parameters provided by the environment or the internal states of a component.

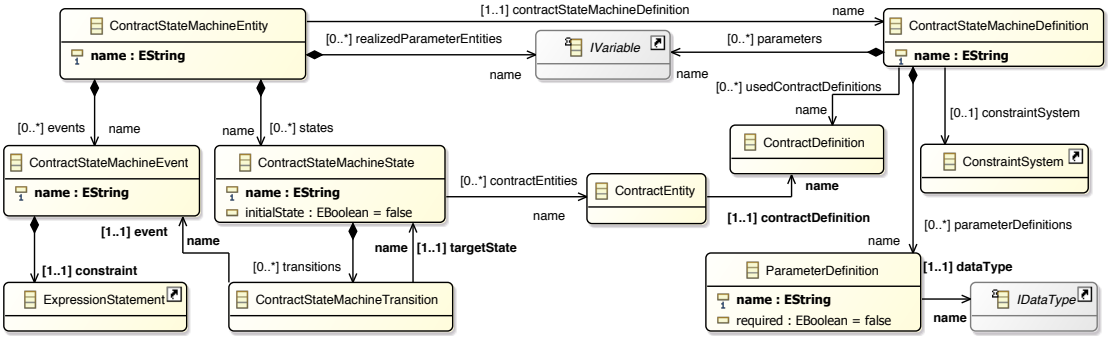


Figure 5.9.: Metamodel for contract state machines.

Figure 5.9 illustrates our proposed metamodel for a contract state machine. We again use the concept of definition and entity in order to separate the specification and actual instance of a so-called contract state machine.

A *Contract State Machine Definition* constitutes allowed *Contract Definitions*, concrete parameters and declarations of parameters, which need to be defined by corresponding *Contract State Machine Entities*.

Parameters are supposed to be used by *Contract State Machine Events* within constraint expressions, which trigger transitions to other *Contract State Machine States*. Such a state

contains zero to infinite *Contract Entities*.

Again, the metamodel elements *IVariable*, *IDataType* and *Expression Statement* and *Constraint System* are provided by the constraint modeling language.

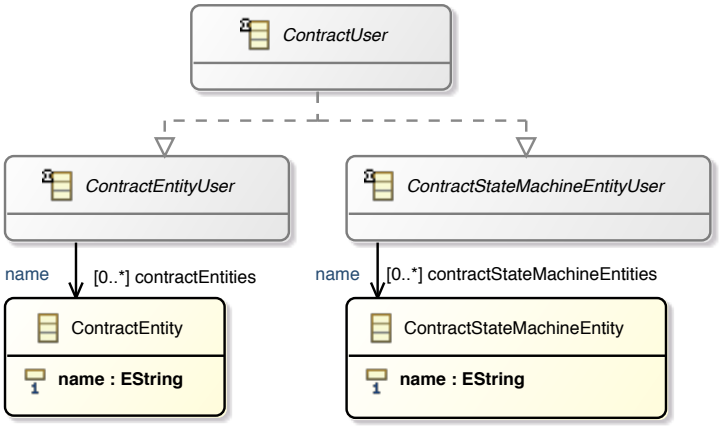


Figure 5.10.: Metamodel for the interface contract user.

Additionally, the modeling language provides the interface *Contract User* for adding the contract functionality to classes found in the modeling languages resource, component, deployment, and system configuration. Figure 5.10 illustrates this class. Every class inheriting from *Contract User* can reference contracts and contract state machines.

5.3.3. Data Point

Responsibility: The data point modeling language is used for specifying data points provided by resources or function plans.
Rationale: We extracted the definition of data points into an own modeling language in order to reuse the metamodel for the resource, component, and deployment modeling languages.
Structure & Behavior: The data point modeling language is based on the constraint modeling language in order to reuse the semantics of data types and variables.

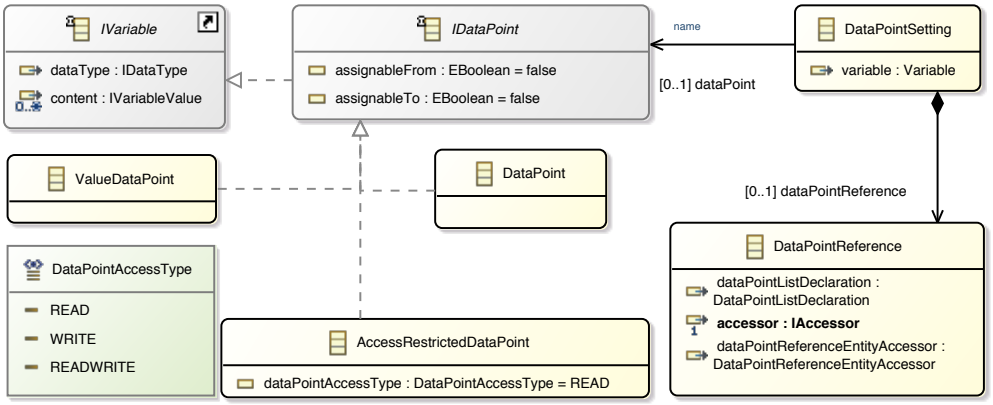


Figure 5.11.: Metamodel for data points.

Figure 5.11 illustrates the interface *IDataPoint* that inherits from IVariable. We differentiate between *Data Points*, *Value Data Points* obtaining a variable value, and *Access Restricted Data Points*. Semantically, we interpret the *Data Point Access Type* in assignments. *Data Point Setting* is utilized by instantiations of FUPs, POUs, and modules for setting a data point.

Figure 5.12 shows the metamodel of the concept of data point references. As illustrated by the exemplary *Data Point Reference*, it first references a *Data Point Reference Entity*, then a *Data Point List Declaration* and finally the target *Data Point* through an *IAccessor*. We decided to implement it that way because the reference follows the same syntax similar to the one used by our project partner.

Figure 5.13 illustrates the concept of data point lists. These are used by FUPs, POUs, tasks, and modules. A *Data Point List Declaration* declares a list and configures distinct aspects. We introduced this concept because it adds some flexibility to our DSLs concerning future device generations.

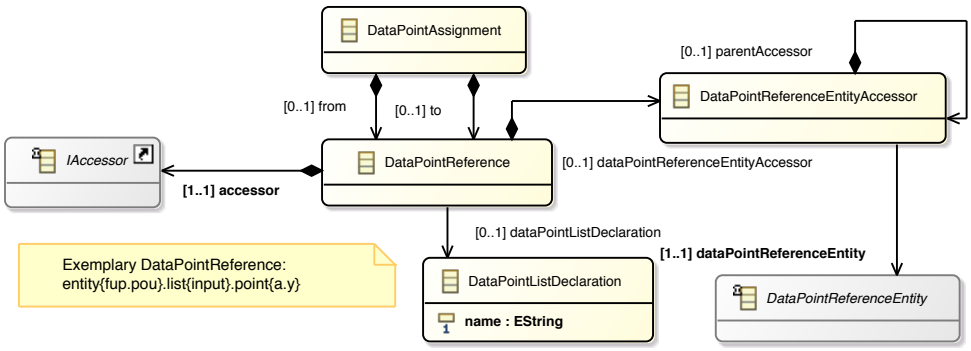


Figure 5.12.: Metamodel for data point references.

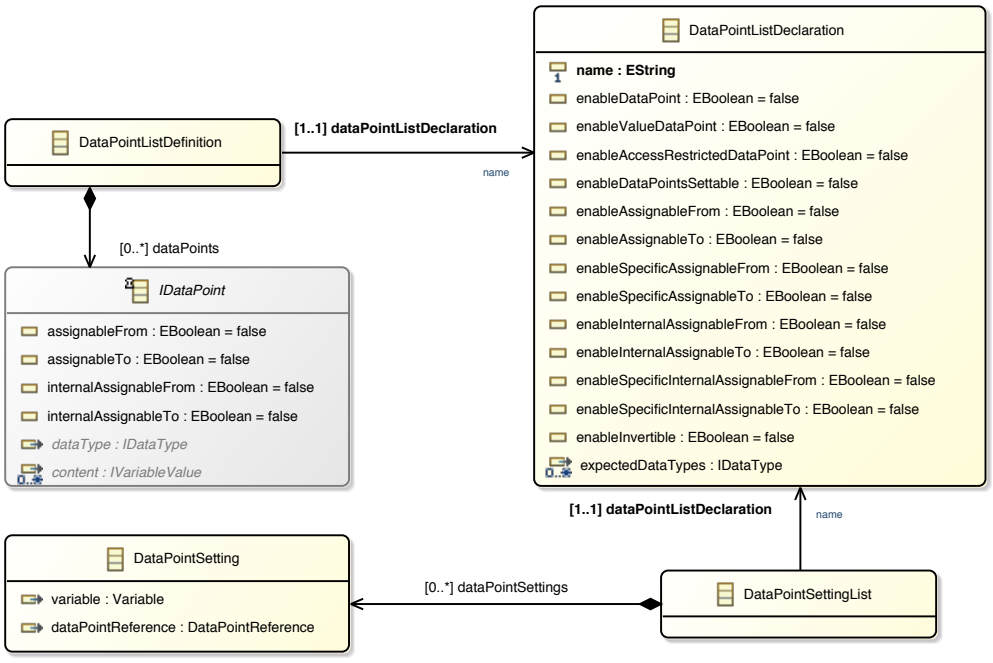


Figure 5.13.: Metamodel for specifying lists of data points.

5.3.4. Resource

Responsibility: The resource modeling language reflects the used hardware, software applications, control devices and network topologies.
Rationale: We separated the resource metamodel into an own modeling language in order to be flexible regarding deployments and applications. We could add new kinds of devices without changing the other modeling languages.
Structure & Behavior: In the following, we discuss the resource modeling language starting with the "smallest" hardware components and ending with the network level.

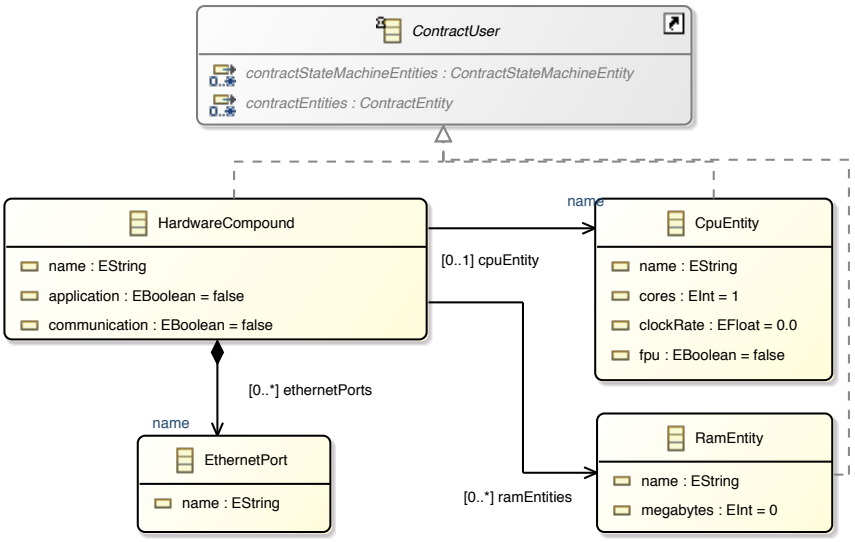


Figure 5.14.: Metamodel for hardware compounds.

Figure 5.14 shows the class *Hardware Compound*, which references a *CPU Entity* and a *RAM Entity*. We decided to reference these an not contain them because the specifications can be reused by other *Hardware Compounds*. It also models Ethernet ports.

Figure 5.15 illustrates that the concept of a module is split into three classes namely *Module Entity*, *Module Definition*, and *Module Declaration*. *Module Declaration* states usable data types and data point lists. We added this class in order to be flexible concerning future modules that may use other types and lists. *Module Definition* defines the concrete module type with *Hardware Compounds* and *Data Point List Definitions*. The *Module Entity* instantiates the module and contains *Software Applications* an *Data Point Setting Lists*. It also references the connected *Power Plant Device*.

Figure 5.16 showcases that a *Hipase Device* contains *Module Entities*. It inherits from *Network Resource* in order to be referable by a network.

Figure 5.17 presents the network layer. It references *Network Resources*, such as a

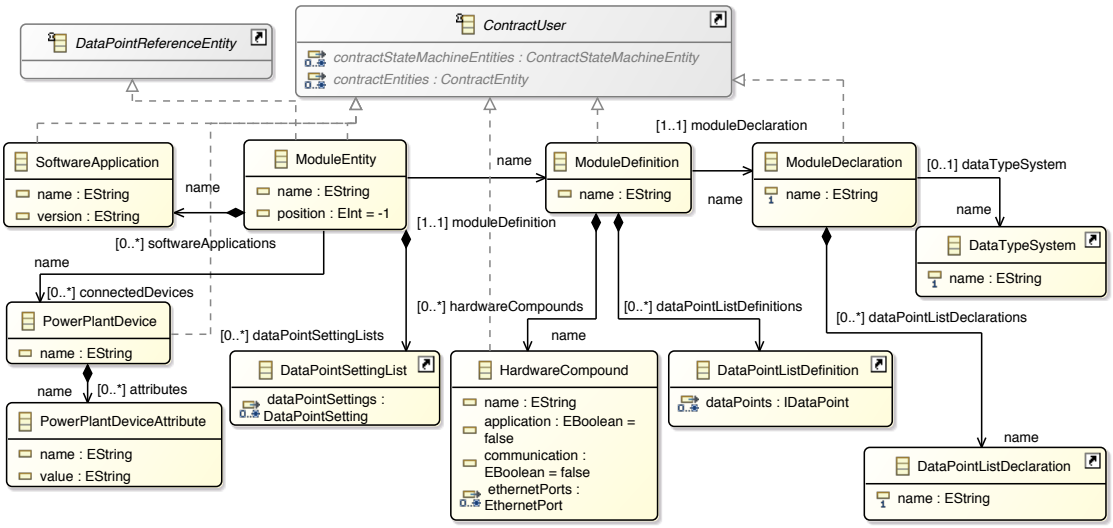


Figure 5.15.: Metamodel for central and interface modules.

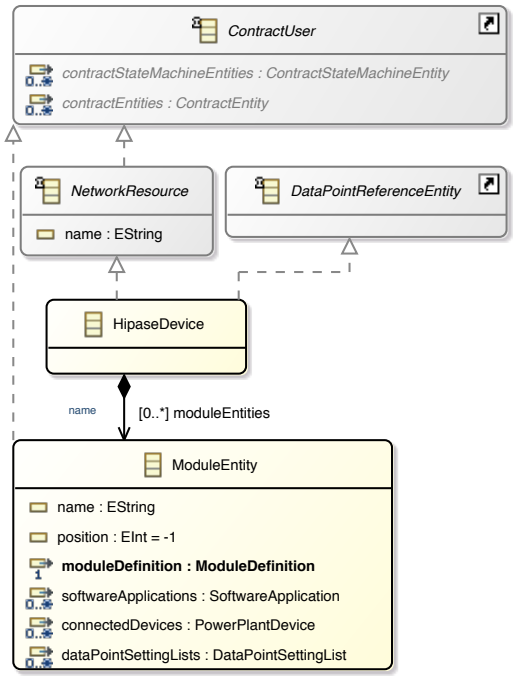


Figure 5.16.: Metamodel for PLCs.

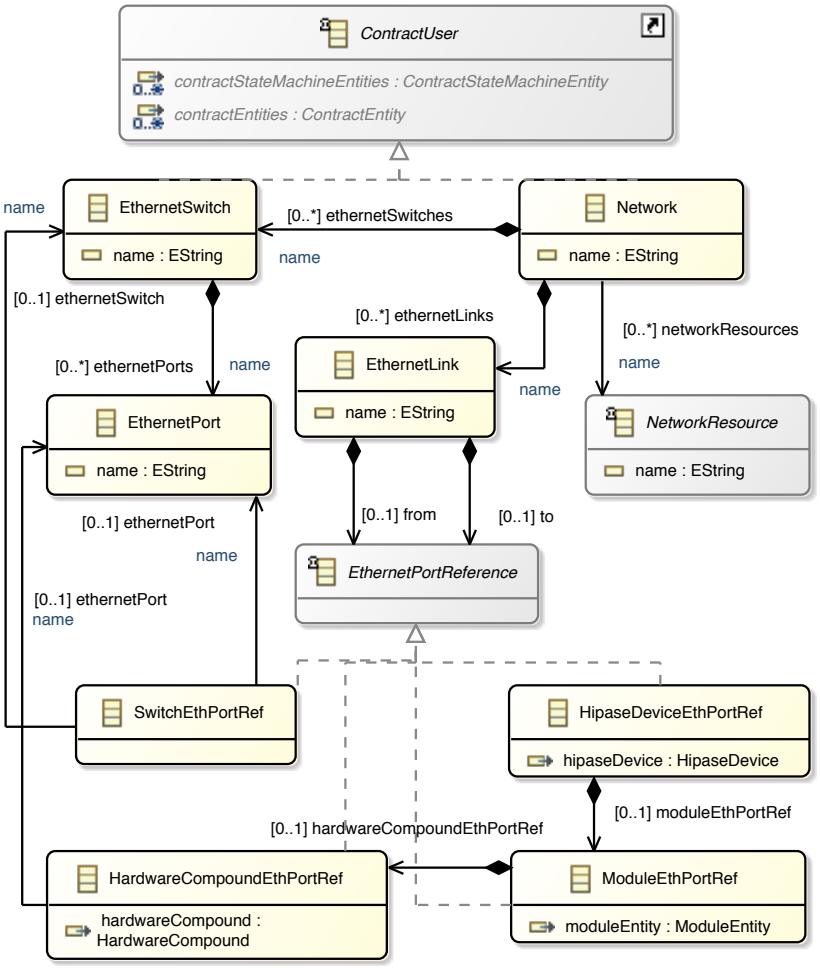


Figure 5.17.: Metamodel for networks.

Hipase Device, and connects the *Ethernet Ports* through *Ethernet Links*. Semantically, the concrete syntax ensures that *Ethernet Ports* contained in the *Network Resources* are available to be referenced by *Ethernet Links*.

The *Network Resources* and *Module Entities* do not provide certain network properties such as an IP address or Modbus configuration. We propose to describe such properties with contracts in order to make them configurable without changing a resource model.

5.3.5. Component

Responsibility: The component modeling language is used for describing POUs, FUPs, tasks and applications. An application contains tasks, their interconnections and data point assignments.
Rationale: We structured the control logic into an own modeling language for making it independent of changes in the resource or deployment metamodels. The POUs, FUPs, and tasks can be specified similar to the way it is done by the IEC 61131 standard for programmable logic controllers [92]. We added the concept of an application, which allows the connection of cyclic tasks independent of their deployment.
Structure & Behavior: In the following, we present the component modeling language starting with POUs and ending with applications.

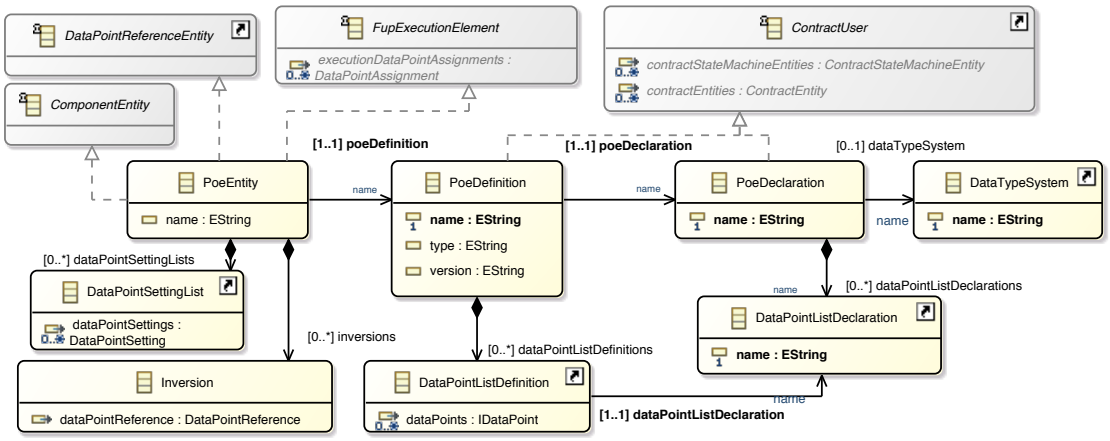


Figure 5.18.: Metamodel for POUs.

POUs are implemented as libraries and written in the C programming language. Figure 5.18 shows the metamodel representing such software components. A *Poe Declaration* states the usable data types and data point lists. We added this class in order to be flexible concerning future POUs that may use other types and lists. The *Poe Definition* specifies the available data points and contracts. This definition is reused by the *Poe Entities* that are instantiated by FUPs. The class *Inversion* inverts the value of a data point.

FUPs are compositions of POUs and FUPs. As illustrated in Figure 5.19, we structured it similar to the concept of a POU. It obtains a class *Fup Execution Sequence Definition* that states the sequence of POUs and FUPs. This sequence is then executed by the PLC in order to control a hydropower unit.

Tasks specify the sequence of FUPs and the cycle time. Figure 5.20 shows the metamodel. Again, we split the concept into the three classes.

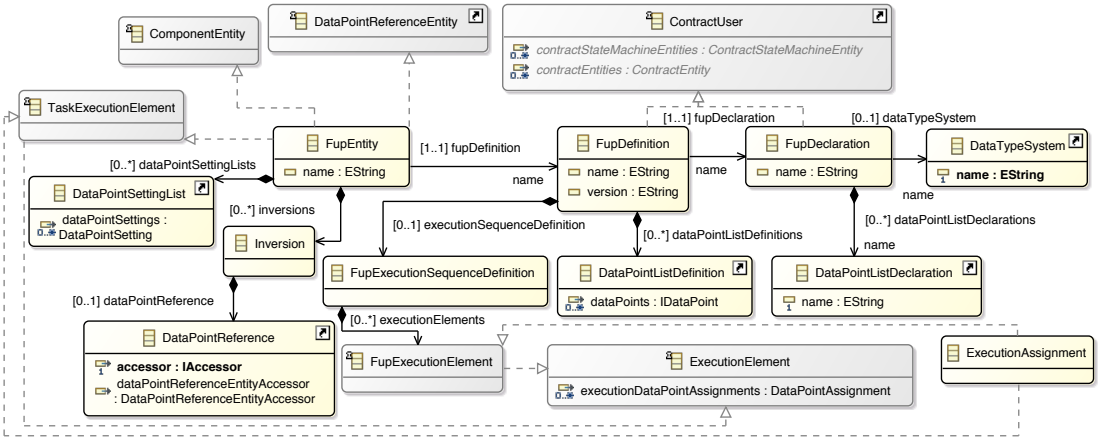


Figure 5.19.: Metamodel for FUPs.

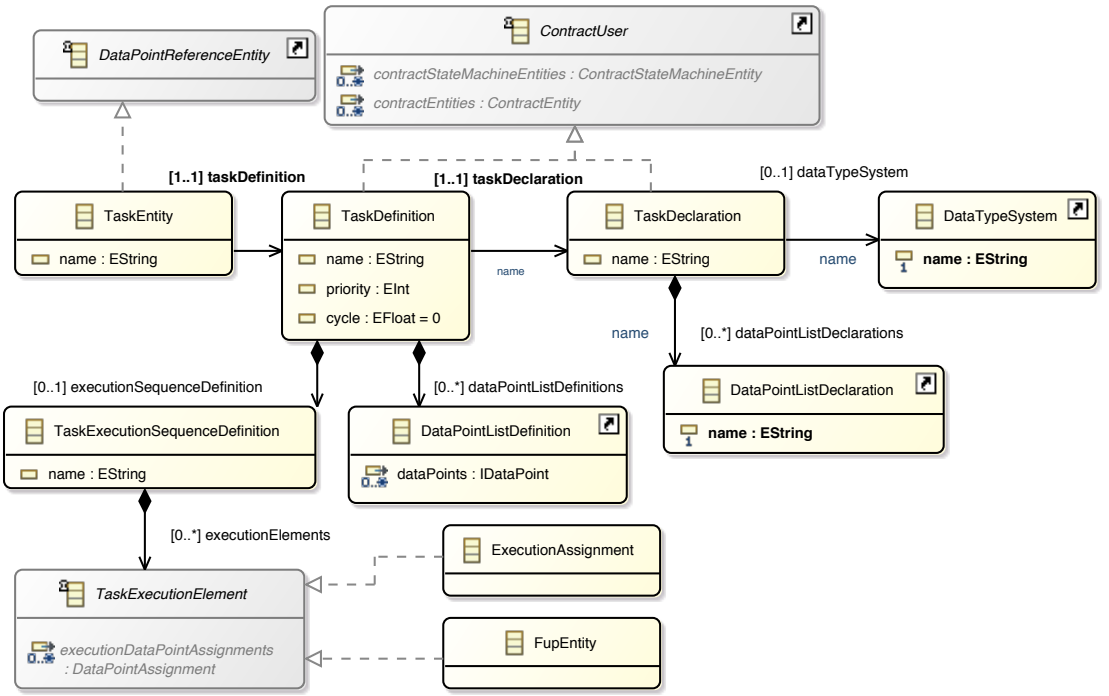


Figure 5.20.: Metamodel for tasks.

The concept of an application is not present in the IEC 61131 standard for programmable logic controllers [92]. We added it to model distributed cyclic tasks that are interconnected.

Figure 5.21 showcases the metamodel. A *Task Connection* allows to interconnect *Task Entities* and data points.

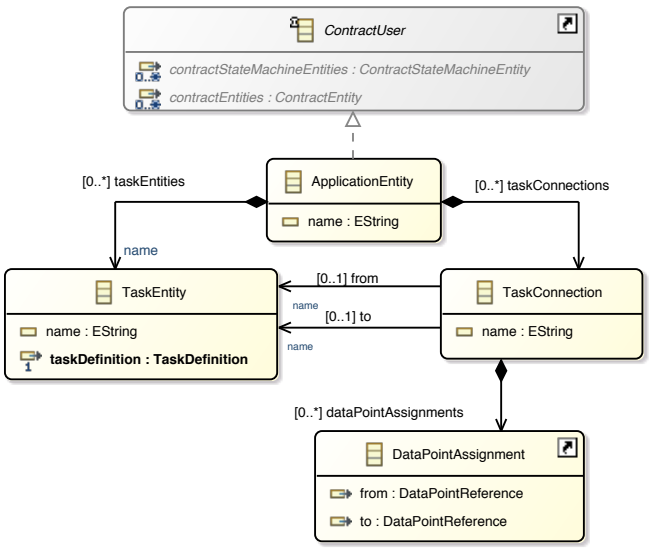


Figure 5.21.: Metamodel for applications.

5.3.6. Deployment

Responsibility: The deployment modeling language is used for specifying how tasks, defined with the component modeling language, are deployed onto the control devices, defined with the resource modeling language. Additionally, it is possible to deploy contracts to resources.
Rationale: This modeling language is necessary for "gluing" control logic together with resources. It allows us to let both exist independent of each other.
Structure & Behavior: Figure 5.22 illustrates the deployment classes. The central class is *Deployment*, which can contain *Contract Deployment*, *Contract State Machine Deployment*, *Task To Hipase Device Deployment*, and *Task Connection to Ethernet Links Deployment*.

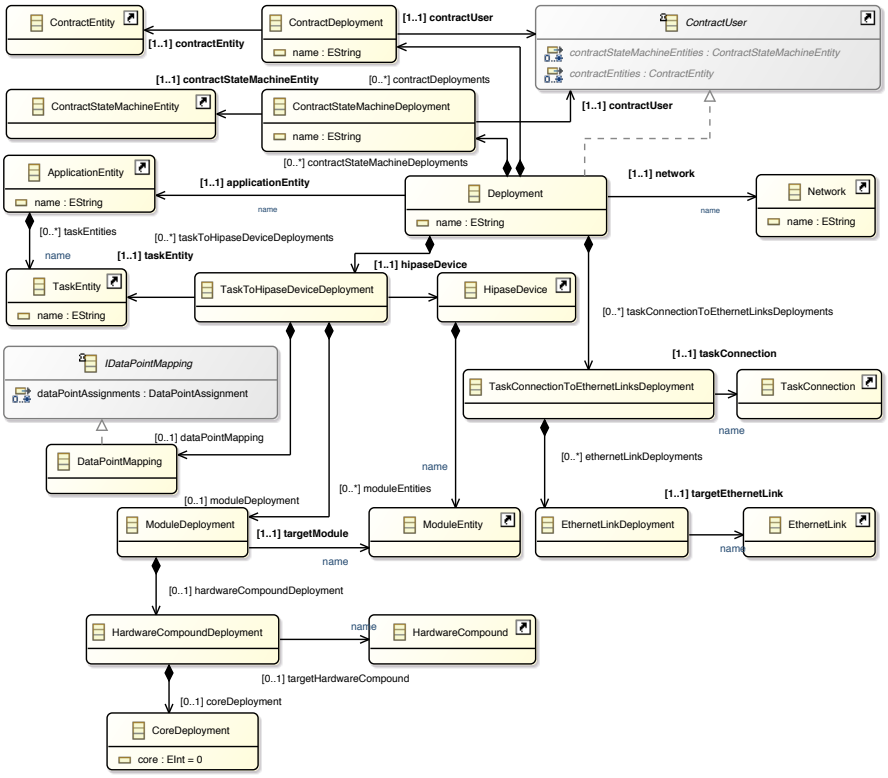


Figure 5.22.: Metamodel for deployments.

The contained classes of the class *Task To Hipase Device Deployment* refine the deployment. These fine-grained deployments could be added by an automated tooling. The *Data Point Mapping* fulfills the crucial rule of interconnecting data points.

5.3.7. System Configuration

Responsibility: The system configuration references used resource, component and deployment models. It serves as entry point for model verifications and transformations.
Rationale: Collecting a complete hydropower plant configuration based on applications, resources, and deployments is a tedious task. This issue is solved by the system configuration modeling language.
Structure & Behavior: Figure 5.23 illustrates one class named *System Configuration* that refers to the application entities, deployment, and network models. Its only purpose is to provide an entry point for model analysis and transformations. It is also useful for verifying that all applications are deployed to resources.

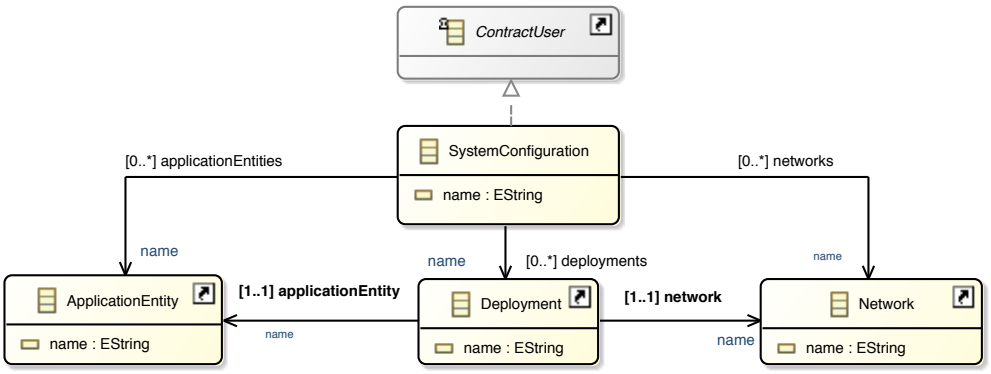


Figure 5.23.: Metamodel for the system configuration.

5.4. Technical Implementation

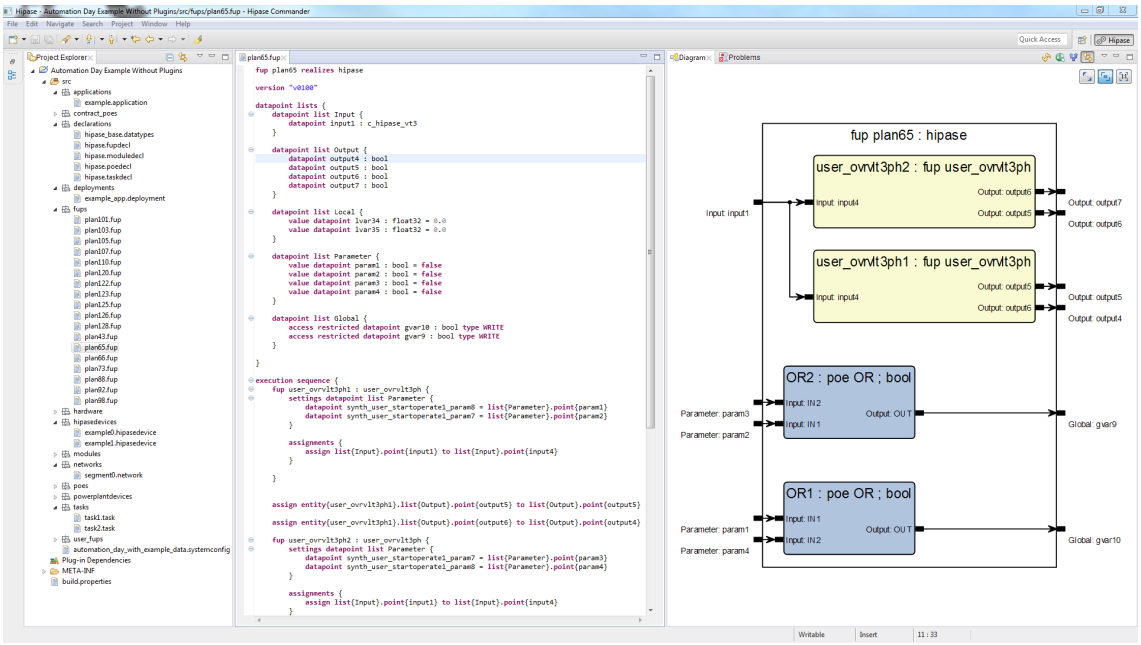


Figure 5.24.: Screenshot of the Eclipse Rich Client Prototype.

We realize the metamodels using the Eclipse Modeling Framework [96]. Additionally, we use Xtext [97] for designing textual domain-specific languages. The use of textual languages influenced the design of the modeling languages in Section 5.3. For instance, references are often not directly between objects, but through an object linking two objects. This is necessary for modeling a textual reference. Furthermore, we leverage the KLighD framework [98] for automatically visualizing the models graphically.

Figure 5.24 shows on the left-hand side various files that represent parts of the system configuration. Because we are utilizing textual languages, there is not one model containing all information but several files containing parts of a complete model. This has two advantages: First, the parts can be referenced and composed arbitrary by distinct system configurations. Second, textual models are utilizing names and not UUIDs for referring to objects. This simplifies replacing a part with a distinct one. With graphical modeling languages a user usually cannot influence the UUID of a newly created object. It is therefore a tedious task to update the UUIDs of related models if one wants to replace an object.

5.5. Utilization of Models and Contracts

We identified three application areas for our proposed modeling languages. In the following, we briefly discuss the areas *verification*, *search of combinations*, and *input for run time*.

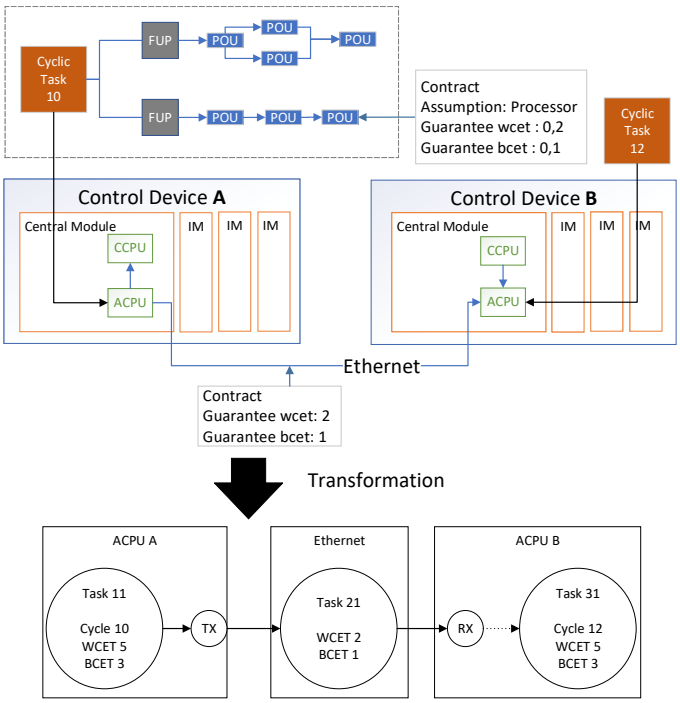


Figure 5.25.: Exemplary verification of timing.

Verification: The presented modeling languages allow to verify an industrial configuration concerning functional and non-functional properties. From a functional point of view, the deployments can be verified if the provided data points of an interface module are compatible with the input and output data points of tasks. Also the data point compatibility of distributed tasks can be verified. From a non-functional point of view, a system configuration can be verified with regards to the distinct properties enumerated in Section 5.3.

The purpose of the proposed contract modeling language is to enable the specification of non-functional properties in order to transform them, with the support of the functional model, into input data for specialized analysis tools. For instance, we experimented with pyCPA[1], which is an open-source tool implementing Compositional Performance Analysis [99], for calculating the end-to-end timing of task chains. Figure 5.25 illustrates how such

[1] https://pycpa.readthedocs.io

a verification works. The model annotated with timing contracts is transformed to a representation analyzable with pyCPA. Then the tool calculates the worst-case response times of tasks and end-to-end timing of task chains based on provided busses, CPUs, corresponding scheduling policies, and a task graph. Finally, the calculated results are presented to the engineer responsible for the hydropower unit.

Search of Combinations: An useful addition to our proposed modeling language would be the automatic deployment of tasks to PLCs. A naïve algorithm could try all deployment possibilities of tasks and utilize verification tools for ensuring a correct deployment.

A more advanced method, without trying all possible combinations, could be based on Constraint Programming [100], which is a widely applied method to solve decision and optimization problems. Kajtazovic [24] presents in his doctoral thesis, which has been carried out in a research project with the same company, an approach that transforms contracts for software components to a constraint satisfaction problem. Similarly, such an approach could be applied for finding optimal task deployments.

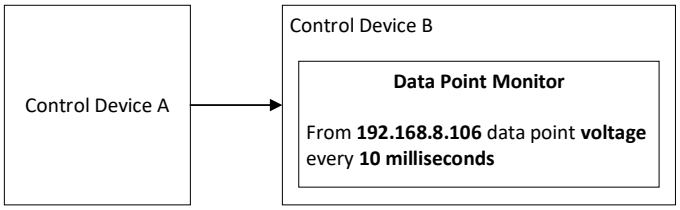


Figure 5.26.: Exemplary run-time constraint derived from design time.

Input for Run Time: A system configuration created at design time can also be useful at run time. For instance, based on the task graph specifying the exchanged data points, a monitor can observe whether data is received by a PLC in time. Figure 5.26 illustrates the case that a monitor observes if the data point "voltage" is sent every 10 milliseconds to the control device. Such observations and run-time constraints can also be based on non-functional properties such as timing, security and redundancy. In Chapter 6, we propose a self-adaptive software system that is based on this idea of utilizing models and contracts at run time.

5.6. Meeting Requirements

REQ1 tasks and FUPs: We provide a dedicated modeling language for specifying POUs, FUPs, and tasks.

REQ2 interface modules and available data points: The resource modeling language allows defining interface module types containing data points. Also instantiations of interface modules can define available data points.

REQ3 interconnections of control devices and tasks: With the resource modeling language it is possible to specify physical connections between ports, e.g., Ethernet. It is possible to define connections between data points contained in tasks with the component modeling language.

REQ4 loose coupling between task and control device: The deployment modeling language ensures that tasks are loosely coupled with control devices.

REQ5 generic modeling concept for non-functional properties: The contract modeling language allows us to define arbitrary non-functional properties. Through contract types we can ensure that contracts adhere to the expected semantics.

5.7. Discussion of Limitations

For utilizing the proposed modeling languages, a tight integration into the development of the single hardware components and software applications would be necessary. For instance, contracts for capturing the timing behavior would need to be defined based on automatic test setups. This would reduce the need for engineers who have a deep understanding of contract-based design.

With our proposed resource modeling language we tried to be generic for enabling future PLC hardware configurations. However, it is still possible that a future PLC requires an upgrade of the modeling language.

The decoupling of control logic and resources is different to the existing approach taken by our project partner. With their programming tool, FUPs are directly programmed for the target PLC. With our approach a deployment is necessary to be defined by a user. Because of the decoupling, it would need more assistance from the tooling in order to guide the user for defining valid input and output data points leveraged by FUPs in our approach.

5.8. Design Patterns

We found four design patterns for designing configurability into domain-specific language elements. We derived them from our work on modeling languages. Table 5.1 shows an overview of these design patterns with a short description of them. These patterns target to solve the general problem of how to embody a domain-specific concept in language elements that a language user is able to configure the concept to a certain degree. Therefore, they share the same problem and context but are subject to different forces.

Each of these patterns adds another level of configurability to a concept. The detailed explanations including associated forces and consequences can be found in **Paper B** [101]. We demonstrate the patterns by utilizing a simple finite state machine. In addition to these state machine examples, it contains an experience report of how we applied these

patterns in our research project. Concerning Scari (run time), these patterns can be applied for designing the modeling language of a world model.

Pattern	Description	Known Uses
ATOMIC CONCEPT	Instantiations of a language concept only differ by the values of their properties. The solution is to represent a concept as one language element and to add desired configurable attributes and references. This pattern is the simplest kind of representing a concept within a domain-specific language.	• JSON. • Can be found in virtually every domain-specific language (e.g. control structures).
ENTITY TYPE	Instantiations of a language concept differ concerning their available properties. Additionally, it is desired to reuse the definition of available properties. The solution is to add an Entity language element and an EntityType element. These two different kinds of elements can only represent the target concept together. The Entity element must reference at least one EntityType element. The EntityType element can define properties for the Entity element. Further, it can specify reusable data. Entities can be differentiated by their referenced EntityTypes.	• The elements *Class* and *InstanceSpecification* found in UML. • Interfaces and concrete components found in component-based modeling languages. • The patterns TYPE-OBJECT and ITEM-DESCRIPTOR.
DEEP CONFIGURATION	The reuse of already specified Entities is desired and the number of reused Entities varies. The solution is that Entities can reference other Entities in order to add, modify or remove configuration information. There can be an arbitrary long chain of them. Each Entity represents an instance of the language concept and may be semantically complete.	• Inheritance of classes. • Packages contain data and can reuse data from other packages. • UML state machines can be redefined and extended.
CONCEPT TAILORING	A language concept needs to be adjustable for slightly different cases inside a domain. Additionally, these adjustments may configure the language infrastructure. There exist two solutions for this need. The first is to add a Declaration element within the domain-specific language for constraining the concept. The second is to specify the Declaration outside of the language and to load it by the tooling infrastructure around a language. The settings of the Declaration can either restrict a language concept or enable features.	• Profiles found in UML. • XML Schemas. • The patterns KNOWLEDGE LEVEL and ADAPTIVE OBJECT MODELS.

Table 5.1.: Patterns for designing configurability into domain-specific language elements.

6. Run Time

This Chapter is devoted to the phase a control system is in operation. We identify the potential of a self-adaptive software system in our industrial context for detecting and adapting to hardware faults, security attacks, misconfiguration of the control logic, software bugs and changes in the environment (**C2**). Based on this analysis, we propose a decentralized hierarchical and model-based self-adaptive software system named Scari (**C3**). From our work on decentralized and distributed systems, we derived four design patterns (**C4**). Three design patterns try to grasp the trade-off between distributing data and information. The fourth design pattern provides a solution for separating processing and coordination in computer systems.

In Section 6.1, we motivate our work. In Section 6.2, we analyze the potential of self-adaptive software systems in our industrial setting. Section 6.3 discusses requirements on a self-adaptive software system that are derived from our analysis. Section 6.4 provides an overview of how Scari works in general and the distinct parts in detail. Section 6.5 explains how we implemented Scari technically. Section 6.6 showcases distinct situations and how these can be handled with Scari. In Section 6.7, we explain how Scari fulfills the requirements discussed in Section 6.3. Section 6.8 enumerates limitations of our approach. Finally, Section 6.9 enumerates design patterns that we found during our work on self-adaptive software systems.

6.1. Motivation

Industrial control systems have three advantages for adding resilience compared to other kinds of systems such as enterprise or even consumer systems. The first advantage of industrial control systems is that they are highly deterministic in their behavior. Data is processed and distributed at fixed points in time. Adjustments of physical processes yield predictable output, except if something is broken. The second one is that all devices used, including hardware, software, network topology, and communication patterns etc. are in principle known and only changed under strict change management. The third advantage is that a lot of industrial systems are designed in a redundant way in order to deal with permanent faults.

A self-adaptive system that takes these three advantages into account could defend the hardware/software stack of a system against a variety of threats. The determinism can be leveraged for detecting odd behaviors. The knowledge about existing devices can be

utilized for detecting wrong arrangements and connections. The redundancy of data allows to detect and circumvent faulty parts of a system.

6.2. Potential of Self-Adaptive Software Systems

Table 6.1 is an analysis of our industrial setting and illustrates anomalies, detection and adaptation mechanisms for the different system threats *hardware fault*, *security attack*, *software bug* and *misconfiguration* from the perspective of control devices. The latter threat *misconfiguration* refers to the design of FUPs and how control devices are configured to distribute data points. A different version of this analysis is presented in **Paper E** [102], which also includes a fifth potential application namely *faults in the environment*. In the following, we skip this part in order to simplify the presented analysis. We created the analysis presented in **Paper E** based on the combined expertise of the four authors and an expert from the cooperating company. The first draft of this analysis was created by the author of this thesis. Then the draft was discussed with the three co-authors of **Paper E**. Finally, the four authors discussed the analysis together with the head of the software development dealing with the control device from the cooperating company.

Table 6.1 uses three sets of columns for describing threats. First of all, concerning anomalies, then detection, and last adaptation methods. The left-hand side shows the location where a problem originating from the different kind of threats resides. *Sensor* and *Actuator* are part of the hydropower unit and generalize the various devices connecting a control device with a turbine. Concerning the control device, interface modules are connected with the sensors and actuators of a hydropower unit. They transmit their measured values as data points to the *ACPU* and receive parameters as data points. The *ACPU* is part of the central module and the place where the actual control logic resides. The *CCPU* is responsible for the communication with the supervisory network and controls the *ACPU*. An *ACPU* can be connected with the *ACPU* of another device in order to share data points. We analyzed the network level concerning two kinds of devices. *Connected device* represents a control device that sends or receives data points. *Network resource* is a very abstract representation of any other device in a network such as a switch or supervisory computer. The enumerations of anomalies, detection and adaptation methods are not complete and intended to show what is possible in our industrial setting.

Anomalies: *Dead* is an anomaly describing that the affected location is not responding to interaction attempts. This can be caused by a hardware fault, security attack, or software bug. We do not consider the misconfiguration of FUPs as a cause, because FUPs are merely changing parameters of actuators. This should not affect the actuator itself. *Performance* describes how well a system executes a task. It consists of CPU load, memory consumption or latency. When it comes to the hydropower unit and the control device level, we consider the performance to be potentially affected by hardware faults,

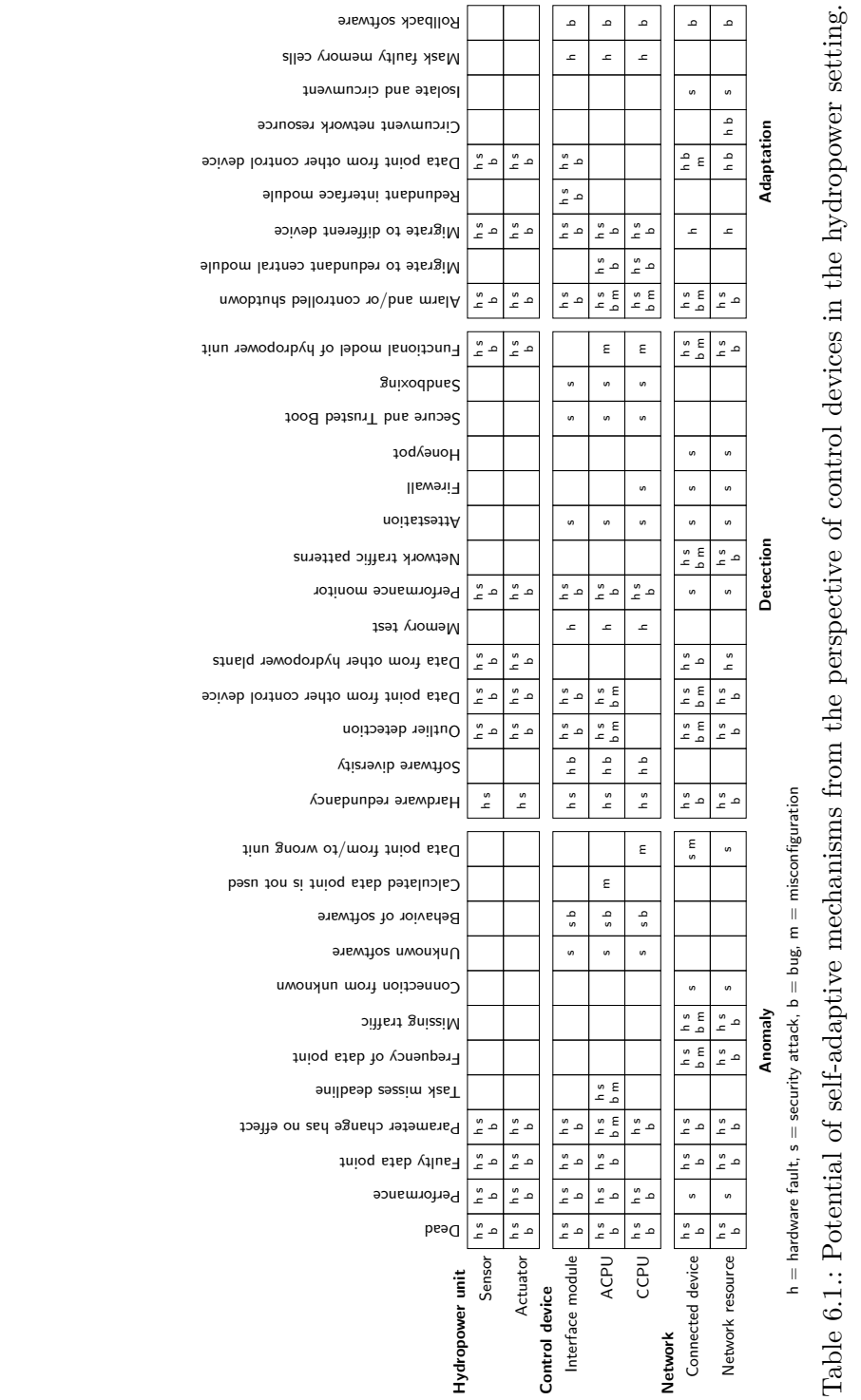


Table 6.1.: Potential of self-adaptive mechanisms from the perspective of control devices in the hydropower setting.

software bugs or security attacks. The intention for a security attack may be to manipulate the real-time control loop or to execute malware. From a security perspective, interface modules can be hacked because their software can be updated through the *CCPU*. On network level we relate the term performance to the transmission speed of data from one device to another. We treat performance losses concerning the transmission of data as a security-related attack, e.g., caused by a Denial-of-Service attack (DoS attack). *Faulty data point* refers to a data point that is wrong but not through a systematic error. We regard this to be caused by hardware faults, security attacks or software bugs on all enumerated locations except the *CCPU*. The *CCPU* can be the origin of such an anomaly if it transmits data points between an *ACPU* and an engineer or supervisory system accessing a PLC. However, in this analysis we relate *Faulty data point* to the exchange of data points between *ACPUs*. *Parameter change has no effect* describes the situation that a FUP or supervisory computer wants to change an actuator through a data point, but the sensed data points do not indicate that anything is behaving differently. The cause of such a problem can reside on all locations and originate from all mentioned threats. *Task misses deadline* refers to the situation that a FUP, contained in a cyclic task, needs to finish its work before the cycle starts again. This can only happen on the *ACPU* and we consider this to be caused by a hardware fault, software bug, security attack or misconfiguration. *Frequency of data point* describes the distribution of data points within a network. A faulty frequency can be caused through a hardware fault, security attack, bug or misconfiguration residing on a connected device or on a network resource. A similar anomaly is *Missing traffic*, which stands for the absence of distributed data points and messages. *Connection from unknown* refers to the case of a hacked control device or unknown network resource that suddenly attempts to communicate. *Unknown software* means that foreign software is started or executed on the interface module, *ACPU* or *CCPU*. *Behavior of software* represents hacked or faulty software that accesses data or communicates in an unexpected way or frequency. *Calculated data point is not used* states a misconfiguration of a FUP. *Data point from/to wrong unit* can either be caused by an unintentional misconfiguration of the *CCPU* or a security attack on network level.

Detection methods: *Hardware redundancy* is about duplication of components or devices in order to increase the reliability. A voter can then detect hardware faults or security attacks if only one subsystem is affected by such a threat. We added the detection of software bugs on network level because the control devices could be updated at different points in time. *Software diversity* refers to the idea of realizing a software functionality in two or more distinct ways. One way is by developing the functionality of a software several times in different ways by using independent development teams and technologies. A second approach is to compile software with different settings and to automatically create diverse software [103]. Since diverse software works in slightly different ways, there is a chance that permanent hardware faults can be detected in a manner similar to *Hardware redundancy*. Furthermore, it allows to detect software bugs residing at different parts of the control device. *Outlier Detection* refers to mechanisms that detect patterns within data

points that do not correspond to an expected behavior. This can be achieved by thresholds, historic data, or advanced mechanisms such as machine learning. It should be possible to detect hardware faults, security attacks, bugs and misconfigurations of FUPs that corrupt data points on all locations. Concerning *Network resource*, *Outlier Detection* is supposed to detect transmission faults. *Data point from other control device* can be used for redundancy and plausibility checks. *Data from other hydropower plants* may be used for plausibility checks of the sensed data concerning the river, e.g., height of water, and how hydropower units are behaving in general. *Memory test* is about checking the memory regularly and detecting permanent faults in memory cells. A *Performance monitor* observes a system regarding CPU usage, temperature, available memory or latency. This can be used on all locations for detecting *Performance* anomalies. *Network traffic patterns* describes the idea to observe the timing and sequence of network messages in order to detect incorrect behavior originating from connected devices or network resources. Hadeli et al. [104] demonstrate how such information can be derived from an industrial configuration and expert knowledge in order to detect anomalies. Similarly, Anton et al. [105] evaluate four different machine learning algorithms for detecting network traffic anomalies in industrial networks. *Attestation* is a security method where an attestator proves to a challenger, locally or remotely that used files and executed binaries correspond to their signatures [106]. *Firewall*, *Honeypot*, *Secure and Trusted Boot* and *Sandboxing* refer to mechanisms that are primarily applicable for detecting a security attack. *Functional model of the hydropower unit* describes the idea of a detailed software model in order to verify the change of data points with the expected outcome. This includes the possibility to verify if the configuration of the FUPs and distribution of data points is correct.

Adaptation methods: *Alarm and/or controlled shutdown* is a strategy applicable to all cases. *Migrate to redundant central module* is possible because the presented control device offers an open architecture where arbitrary modules can be attached. *Migrate to different device* can either be realized with a hot-standby device instantly taking over or through a costlier process where a device is selected for taking over the control activities. *Redundant interface module* enables to switch between interface modules and to circumvent hardware faults, security attacks or software bugs. *Data point from other control device* refers to the possibility to transmit data points, e.g., a data point for voltage, from one device to another. *Circumvent network resource* depends on the affected network device. A switch can be circumvented if another path exists. A supervisory computer can be replaced with a redundant one. *Isolate and circumvent* stands for blocking a *Connected device* or *Network resource* and to circumvent it by leveraging redundant devices or data points. *Mask faulty memory cells* leverages the capabilities of operating systems to blacklist memory areas. Modern control devices can periodically receive updates and new features. *Rollback software* assumes that a former version does not contain a freshly introduced software bug.

As we can see in Table 6.1, there are several shared anomalies, detection and adaptation

mechanisms. Some of the presented detection and adaptation mechanisms rely on having redundant or stand-by hardware. One could argue that combinations of detection and adaptation mechanisms can be implemented separately and focused on only one system threat. The downside of keeping combinations of detection and adaptation mechanisms strictly separate is that they might interfere with each other or reach wrong conclusions about the real cause of an anomaly. A self-adaptive software system offers means to deal with cross-cutting anomalies by orchestrating different mechanisms. It would have to reason about the combined outcome of several reasoning mechanisms in order to find the real cause. Based on the cause, a suitable adaptation mechanism would have to be selected. It is important to note that adaptation mechanisms do need some execution time. This may introduce delays into control processes. The impact of a small delay depends on the domain and application. For example, in the hydropower plant context one could argue that the benefit of fixing a permanent hardware fault, but at the cost of introducing a small delay is preferable to a broken or faulty control activity. Furthermore, testing an adaptation before it is carried out is a complicated procedure. This means a self-adaptive system must have a precise representation of the real system in order to ensure that an adaptation works as intended.

All in all, it is our considered opinion that the combination of presented detection and adaptation methods would add a valuable improvement to industrial control systems. The novelty is that these methods are glued together in order to enable a self-adaptive software system dealing with distinct system threats. It would allow the control devices to repair and protect themselves while they support the physical process.

6.3. Requirements

In the following, we discuss requirements derived from the analysis above. Requirements are marked with a "REQ" plus a number in square brackets.

A system for lifting the potential from above needs to be *open for arbitrary and existing detection mechanisms [REQ1].* For instance, the detection method "Sandboxing" can be based on existing technologies, e.g., mandatory access control for Linux platforms. Furthermore, functionalities such as the detection and adaptation method "Data point from other control device" may already exist in PLC software. It should require a *minimal effort for adding detection mechanisms [REQ2]* to the system. At the same time, a self-adaptive software system needs to allow the *parallel coexistence of distinct reasoning mechanisms [REQ3].* As an example, the detection method "Performance monitor" can be used for detecting anomalies originating from hardware faults, security attacks or software bugs. However, it needs entities that interpret an anomaly and try to repair or prevent the root cause. Such an interpretation can be based on the feedback from multiple detection methods. Therefore, if reasoning mechanisms coexist in parallel they can track and interpret distinct feedbacks from detection methods over time. In order to support

this coexistence of distinct reasoning mechanisms, it is a requirement that a detection mechanism can *notify distinct reasoning mechanisms at once [REQ4]*. Furthermore, if a reasoning mechanism interprets a situation it should be possible to *recommend a plan for dealing with a situation [REQ5]*. If a reasoning mechanism immediately carries out a plan it would prevent other, possibly more important, strategies recommended by other reasoning mechanism. Because we are aiming at distinct reasoning mechanisms that are executed in parallel, it is possible that at least two mechanisms are recommending plans at two close points in time. Therefore, it is a requirement on the self-adaptive system to *choose one recommendation [REQ6]*. For instance, this allows the competition of two distinct reasoning mechanisms targeting security attacks or hardware faults.

We believe that recommended plans should consist of *atomic operations [REQ7]* because it enables the reuse of adaptation activities. For instance, an atomic operation can be the deployment of a FUP or the stop of a PLC. It should be possible to *combine atomic operations arbitrarily [REQ8]*.

It is crucial that *adaptations must not interfere with each other [REQ9]*. Otherwise, the outcome of two interfering adaptations would be unpredictable and not as intended by single plans. As we want to leverage diverse kinds of reasoning mechanisms that lead to adaptations, it would be beneficial if *plans are reuseable [REQ10]* and their *applicability is checked during plan creation [REQ11]*. The latter requirement is important to prevent an adaptation that harms the industrial system.

In the introduction of this thesis, we outlined that control devices are deployed in various settings and interconnections. Therefore, we consider it to be a requirement that a device is able to *react to threats on its own [REQ12]*. If a fault affects functionally independent devices, they should be able to *react at the same time [REQ13]*. Depending on the devices part of the network topology, it may be possible to *escalate to higher layers with more capabilities [REQ14]*. Typically, PLC devices are part of a SCADA system that is organized hierarchically [107]. The higher layers of such systems control the lower ones. Thus, it is a requirement that *lower layers should not be able to adapt higher layers [REQ15]* in order to respect this control hierarchy and to avoid loops.

For supporting the various detection methods, reasoning mechanisms and plan creators it would be beneficial to provide the architectural information via one consistent *knowledge base [REQ16]*. This knowledge base can contain information from a system's design time such as tasks, function plans, hardware metadata, contracts or topologies. Furthermore, such a knowledge base can *reflect the current state [REQ17]* of a device. If an adaptation is carried out that changes a significant parameter or performs a structural change, the outcome can be reflected inside the knowledge base. Subsequently, detection methods and reasoning mechanisms could reconfigure themselves if the structure and connections between devices change.

One of the above requirements is that a device can react to threats on its own because such devices can be deployed in various settings. Thus, *a consistent knowledge base should be deployed on each device [REQ18]*. Another requirement from above is that a self-

adaptive system should be organized hierarchically and higher layers are typically more capable than lowers. Therefore, it is necessary that *knowledge bases get synchronized between layers [REQ19]* for operating on the current state of the overall system.

Different layers may need different representations of their functionality and structure. Therefore, it should be possible to use *distinct kinds of models [REQ20]* in order to allow specialized representations. It should also be possible to *combine models arbitrarily [REQ21]* for representing a hierarchy with its distinct specialized representations.

6.4. Scari

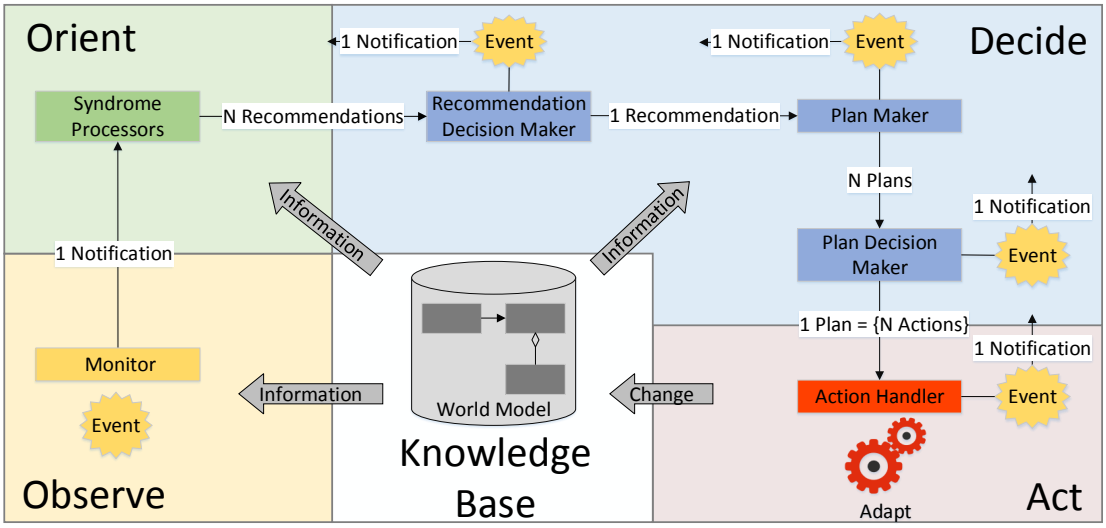


Figure 6.1.: Scari loop.

Scari (**Sec**ure **a**nd **r**eliable **i**nfrastructure) is our proposed solution based on self-adaptive software systems. With the term *infrastructure* we mean the hardware, e.g., CPU, physical network; and software, e.g., operating system, applications; stack providing the facilities for running industrial control logic. A self-adaptive software system is a system that is able to change its own behavior in response to changes in the operational environment [31]. We do not aim to change the control logic itself. Instead, we want to ensure that devices and networks last longer, operate in the presence of hardware faults, mitigate security attacks and can detect bugs and misconfigurations. Roughly speaking, the primary goal of Scari is to provide a generic and easily to extend infrastructure that allows the establishment and orchestration of different kinds of anomaly detection mechanisms with corresponding adaptation mechanisms. Scari itself follows an external approach, which means it observes

the system and adapts it through interfaces and communication channels.

The underlying principle of Scari is an event-driven loop that combines the well-known MAPE-K [28] with the OODA loop of John Boyd [40] [42]. As illustrated in Figure 6.1, it consists of 5 parts, namely *Observe*, *Orient*, *Decide*, *Act*, and a common *Knowledge Base*. The entities of the five parts are loosely coupled through a message bus.

The *Observe* part consists of *monitors* that are specialized in discovering and measuring specific anomalies, for instance a drift in data. These monitors notify an arbitrary number of interested *syndrome processors*, residing in the *Orient* part. The *syndrome processors* are implementing a specific detection mechanism, e.g., for hardware faults or security attacks. Technically, they can use any viable method for detection such as machine-learning or simple thresholds. If one or several *syndrome processors* diagnose a problem, they recommend plan types for handling a situation. For instance, a hardware fault *syndrome processor* may recommend circumventing a damaged module, while a security *syndrome processor* may recommend isolating a device. Next, the *recommendation decision maker* selects the best recommendation on the basis of a definable prioritization for the covered events and chosen plan types. The selected recommendation is then forwarded to the *plan maker* that creates the actions for the plan type. Some plan types may be realized in different ways, we thus added a *plan decision maker* that selects the plan with the least affected systems/resources and the lowest number of used actions. In the final part, the plan is executed by a *action handler*, and the system is adapted to the situation diagnosed by a *syndrome processor*. Each of the entities of the *Decide* and *Act* parts feed back their states as events. This enables the *syndrome processors* to log the state of their recommendations and to be notified in turn that the system has adapted.

The *knowledge base* serves as a source of knowledge for the *Observe*, *Orient* and *Decide* parts, while the *Act* part stores the executed changes of the system there. It contains the deployed models, including contracts, from design time and additional run-time information. Examples of such models are control logic (Tasks, FUPs, POUs), installed software applications, hardware (RAM, CPU, etc.), network connections and so on. We engineer these architecture run-time models according to the model-driven engineering principles [10]. So we have metamodels defining domain-specific languages and one meta-metamodel serving as a common technical base for the metamodels. One reason for applying model-driven engineering techniques is that we can precisely describe a part of a system with a specialized language that only the interested entities need to understand. A generalized schema for representing data would make it more difficult to grasp the semantics and less effective to support the *Observe*, *Orient*, and *Decide* steps. Another reason is that models are manageable reflections that abstract from unnecessary details of the system [108].

In addition to the models, the *knowledge base* incorporates a log of the distributed messages (*notification*, *recommendation*, *plan*, *action*) and a world version that changes if the models change. The world version is used by the recommendation and plan messages in order to ensure that they refer to the current state. If a message does not refer to the current state it is rejected by the *Decide* and *Act* entities because it comes from an out-

dated state of the world. Furthermore, *syndrome processors* need to reevaluate diagnosed problems and recommendations that were not taken if the *knowledge base* changes. It is crucial that the models can only be changed with an action while others can just access the *knowledge base* for deriving information.

The combination of MAPE-K with OODA leads in our opinion to the best of both concepts. MAPE-K introduces the *knowledge base* as a common information source for the different steps. OODA adds an explicit *Decide* part, which is useful for the selection of recommendations. The *Plan* step of MAPE-K is distributed over several loosely coupled entities. We also adopt from OODA that each step gives feedback to *monitors* and *syndrome processors*. This allows them to consider what happened with their notifications and recommendations.

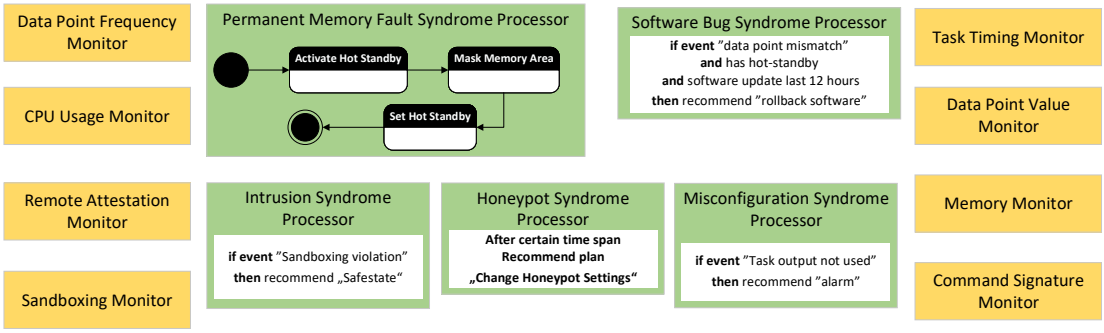


Figure 6.2.: Exemplary syndrome processors.

Figure 6.2 shows examples of possible syndrome processors surrounded by monitors and how they can be implemented. We experimented with state machines, rule engines and time-triggered syndrome processors.

A syndrome processor implemented as state machine is useful for adaptations that span over several plans. For instance the permanent memory fault syndrome processor in Figure 6.2 first recommends the plan to activate a hot standby device, then recommends the plan to mask a memory area and finally changes the control device itself to a hot standby device.

A software bug syndrome processor may be realized as a rule in order to recommend a rollback of software if the conditions are satisfied. The architecture of Scari allows the single syndrome processors to use their own database separated from the knowledge base for saving syndrome-specific characteristics. This further eases the development of syndrome processors.

The honeypot syndrome processor illustrates a time-triggered use case. After a certain time span it recommends to change the honeypot settings. This may include the rerouting of data points or the change of networking settings.

Situation	Plan	Actions
Emergency	Safe State	• Activate Hot-Standby PLC • Stop PLC • Notify Higher Layer
Data Point Mismatch	Increase Data Point Monitoring	• Activate Hot-Standby PLC • Increase Monitoring
Located Faulty Memory Area	Mask Memory	• Mask Memory • Reboot
Hacked PLC	Isolation	• Stop PLC • Add Network Connection • Replace Data Points • Add Attestation Monitor • Start PLC
Interface Module Failure	Switch Module	• Replace Data Points • Stop Module

Table 6.2.: Exemplary situations, plans and actions.

Table 6.2 lists exemplary situations, plans and actions. The term situation refers to cases where the corresponding plan would be suitable. The distinct plan types would be recommended by syndrome processors. The enumerated actions would be dynamically generated by the plan maker based on the models found in the knowledge base. The advantage of this reusability of plan types is that syndrome processors do not need to possess any logic of how a plan is implemented. Also, the plan maker can ensure that the recommended plan is actually possible. An action is supposed to be atomic in order to combine it in distinct plans and to increase its utility.

In our industrial setting, Scari is supposed to run on all devices and is organized in a hierarchical manner. Figure 6.3 illustrates how we intend to deploy Scari in our hydropower plant setting. An event-driven loop would be deployed on each ACPU and CCPU. The CCPU loops are observed by single loops responsible for a hydropower unit. A hydropower unit is basically a turbine controlled by several control devices responsible for excitation, synchronization, protection and turbine control. It is imaginable that these unit loops are goverened by one hydropower plant loop. Such a hydropower plant loop could be observed by a Scari loop responsible for different hydropower plants. In the literature, organizing adaptive loops in the presented way is known as *Hierarchical Control* pattern [33].

As shown in Figure 6.3, the information that flows upwards in the hierarchy are *events* and *knowledge*, while the information going down are actions adjusting lower nodes. It is important to note that a Scari loop residing on a higher layer needs to lock all knowledge bases and implicitly the corresponding action handlers of the lower loops. Otherwise, one

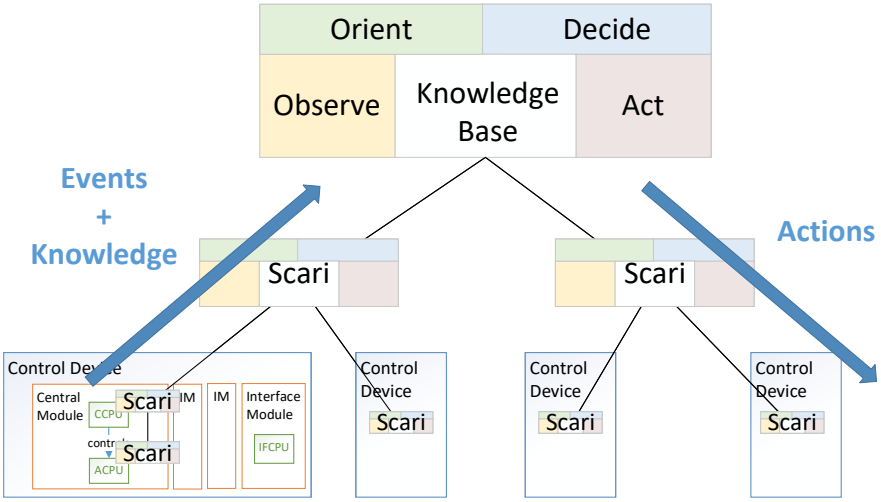


Figure 6.3.: Scari deployed in layers.

lower Scari loop could be faster with an own plan and the result of two interfering actions would be unpredictable. Furthermore, lock notifications from higher loops include the world version of the target loop in order to ensure that the higher loop operates on the current state of the world.

This hierarchical organization of Scari has two advantages: One is that a knowledge base only needs to know its subgraphs. If a knowledge base on a node changes, information is only propagated up to the subsequent higher nodes. The second advantage is that an adaptive loop only needs to handle its subgraph. A loop does not need to manage other parts of the overall industrial system, which also eases the configuration of Scari. If it is not possible to adapt to an event occurring on a node, it can be escalated to a parent node that has more knowledge, more resources, and can therefore leverage more powerful adaptation mechanisms. A drawback is that each higher layer would need to act on a much greater time scale.

In the following, we discuss the responsibility, rationale, and structure and behavior of the knowledge base, monitor, syndrome processor, recommendation decision maker, plan maker, plan decision maker, and action handler.

6.4.1. Knowledge Base

Responsibility: The knowledge base manages a so-called world model of a device and its associated child entities. This model serves as information source for the distinct participants of the Scari loop. It consists of relevant architectural and parametrization information obtained from design time and updated through run-time information. The knowledge bases are responsible for synchronizing information across layers. In addition,

a knowledge base provides a world version for ensuring that entities are referring to the current model. This world version is used by notifications, recommendations, plans and actions. Furthermore, it contains a history of changes in order to enable the assessment of adaptations through human operators.

Rationale: The design rationale for the knowledge base is to provide a consistent source of information about the structure and parameterization to monitors, syndrome processors, plan creators and actions. This is important for ensuring that participants are operating on the same state of the information. Thus, we added a world version that indicates whether something changed. Every change of the models contained in a knowledge base leads to a new world version. A knowledge base uses the world version of child knowledge bases in order to lock them. This is necessary for ensuring that a higher-layer Scari instance operates on a fully synchronized knowledge base. We decided that each Scari instance uses an own knowledge base in order to support distinct deployments of control devices. A locking mechanism is provided by the knowledge bases because they reflect the current states. Only an action handler holding the lock is able to perform changes in order to prevent influences from other action handlers.

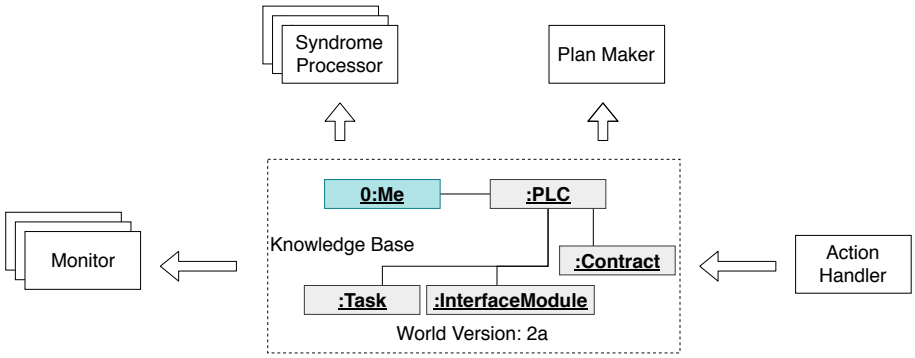


Figure 6.4.: Purpose of the knowledge base.

Structure & Behavior: Figure 6.4 shows how the knowledge base provides the contained models to the distinct entities. The object of type *Me* is known to all participants and serves as entry point for exploring the distinct kinds of models. It references an object, which represents the corresponding device a Scari instance is running on. This generic entry object of type *Me* is useful to monitors, syndrome processors, and plan makers for configuring themselves. Based on the referenced object, they can decide how they behave.

Figure 6.5 shows three hierarchically organized knowledge bases. This is also reflected by the contained models. Again, objects of type *Me* play a crucial role as generic entry points. Such objects also reference child objects of type *Me*. In Scari, the models are fully synchronized upwards in order to provide all information to the participants. The world versions are crucial for ensuring that adaptations are operating on the current state of the world. They are also used for locking knowledge bases by plan decision makers.

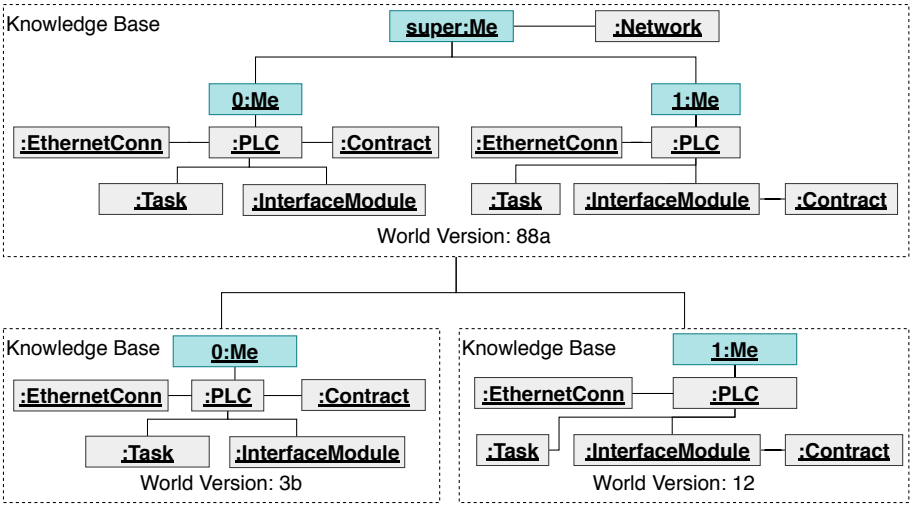


Figure 6.5.: Object diagram of a hierarchy of knowledge bases.

As can be seen in Figures 6.4 and 6.5, the knowledge bases contain the design-time information at run time.

6.4.2. Monitor

Responsibility: A monitor observes one property of a system, detects anomalies and notifies interested syndrome processors. The detection methods enumerated in Table 6.1 can be realized as single monitors.
Rationale: One requirement concerning Scari is to be able to incorporate existing detection methods. Another one is that it should require a minimal effort to incorporate them. Monitors fulfill these requirements. They can represent existing detection methods. For incorporating an existing method into Scari, a monitor can obtain the results from a detection method through an existing interface. Then a monitor distributes a notification containing the event and the obtained information. We decided that a monitor does not directly update a knowledge base for the purpose of making monitors independent from the specific world models. The notifications represent events currently happening on the systems.
Structure & Behavior: The basic step for turning a detection method into a monitor is to add the ability to distribute notifications about an event. A notification contains the event type and necessary details for syndrome processors. If necessary, a monitor can be configurable by Scari actions for instance to increase the monitoring of data points.

A monitor does not update the knowledge base, which reflects the structure and parameterization of a system. However, it is still possible that a monitor configures itself based on models and contracts from the knowledge base.

6.4.3. Syndrome Processor

Responsibility: A syndrome processor reasons about detected anomalies and relates them to information provided by the knowledge base. Based on this interpretation, a syndrome processor may recommend a possible plan type for dealing with the situation.
Rationale: At one point in an adaptive system it is necessary that someone analyzes how to handle a situation. Syndrome processors are supposed to do that based on notifications. We decided to allow the existence of distinct syndrome processors for easing the development of reasoning mechanisms that are specialized on the revelation of a certain fault. Such a reasoning mechanism can react on arbitrary received notifications of interest. Additionally, a syndrome processor can store notifications for analyses over time. We decided that a syndrome processor can only recommend a plan type and cannot create the single actions because of three considerations. The first consideration is that syndrome processors would need to implement concrete algorithms for creating specific adaptations. By only recommending a plan type, the algorithm for creating the adaptations can be reused. The second consideration is that the creation of adaptation plans takes time. Additionally, it may be necessary to retrieve information about the individual devices from the knowledge base. This extra processing time and resource consumption would limit the possibility to compare recommended adaptations from distinct syndrome processors because one syndrome processor may take too long for generating adaptation plans. The third consideration is that reusable plan creators are simpler to test during development.
Structure & Behavior: Figure 6.6 illustrates the distinct states of a syndrome processor and what a syndrome processor observes regarding its recommendation.

In the *Configure* state, a syndrome processor obtains structure and parameters about the corresponding entity from the knowledge base. The information of interest can include for instance whether the syndrome processor runs on a control device, what interface modules are available, what software applications are executed, which data points are received, etc. Based on this information, a syndrome processor can select possible plan types.

Then, a syndrome processor moves into the *Orient* state where it evaluates notifications and relates them to information about the system. In the simplest form, this evaluation of notifications and system information can be realized as a rule-based system. Basically, a rule in a rule-based system consists of an if-then condition [109]. For instance, a security syndrome processor that receives notifications from a monitor that observes PLC commands can implement the following simple rule:

if event "command not signed by XY" **then** recommend plan "block ip address"

Similarly, a syndrome processor repairing software bugs could implement the following rule:

if event "data point mismatch" **and** has hot-standby **and** software update last 12 hours **then** recommend plan "rollback software"

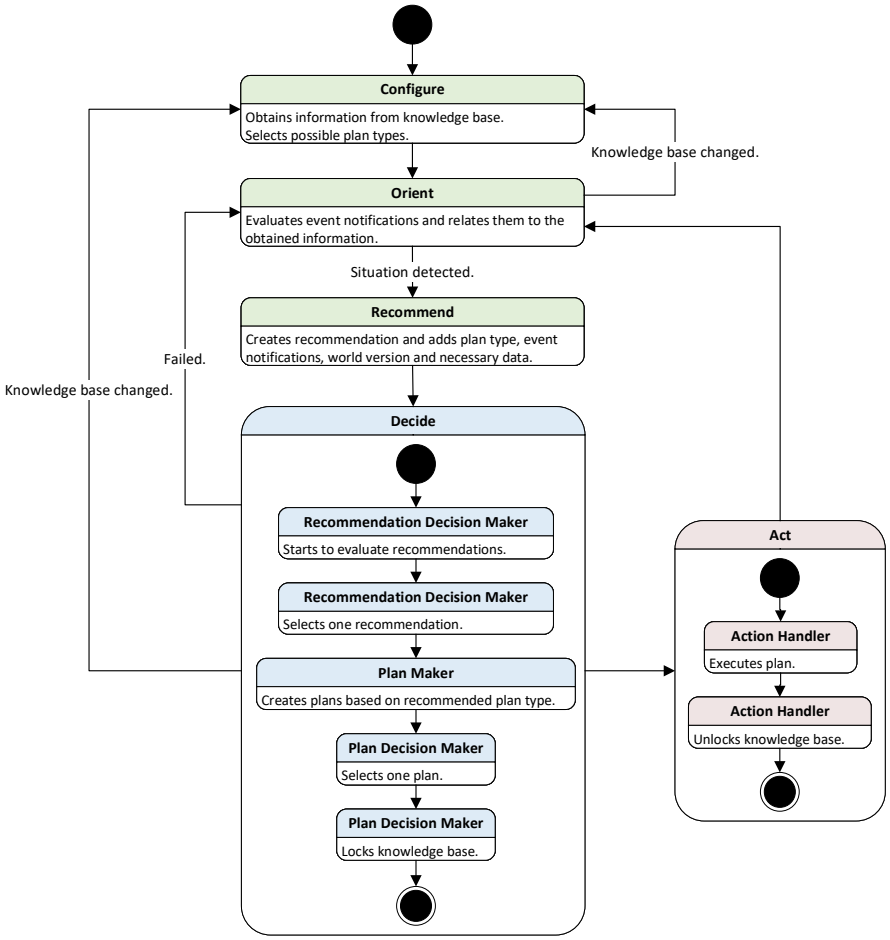


Figure 6.6.: State machine diagram of a syndrome processor.

We also experimented with syndrome processors which are time-triggered or based on state machines. A time-triggered syndrome processor is useful for recurring maintenance tasks. A state machine is useful for syndrome processors that need to execute distinct plans in sequence and decide on their outcomes.

During the *Recommend* state a syndrome processor creates the recommendation with the selected plan type, adds the corresponding event notifications, the world version and necessary data for the plan. Then the syndrome processor distributes the recommendation to the Scari infrastructure.

Now, a syndrome processor needs to track its recommendation if the syndrome processor is interested in the result of a plan. Note that a recommendation does not need to perform an adaptation. For instance, a plan could execute a hardware test that interrupts the

control logic. In this case, a syndrome processor may go through multiple states where it recommends plans based on the outcome of previously executed tests. As depicted in Figure 6.6, a syndrome processor can observe the phases *Decide* and *Act* of the Scari loop. Concerning the *Decide* phase, it is important to track whether the recommendation failed or if the knowledge base changed and the recommendation becomes outdated. After the recommendation passed through the *Decide* phase successfully, the syndrome processor can observe the results of single actions and a plan in general.

Figure 6.6 shows that a syndrome processor usually reverts to the state *Configure* if the knowledge base changed. This is necessary for updating information about the corresponding device and to select available plan options.

6.4.4. Recommendation Decision Maker

Responsibility: A recommendation decision maker chooses between recommendations after a certain timeout. It selects the recommendation based on the covered event with the highest priority. If there are equal recommendations it selects the one with the plan type of the highest priority.
Rationale: Because we propose to leverage distinct syndrome processors for reasoning about a situation, it is necessary that their recommendations are compared to each other. A recommendation consists of the covered events and a plan type. The recommendation decision maker selects a recommendation based on the highest prioritized event. For instance, in our industrial domain, events concerned with the correctness of the control activities are more important than temporary high CPU utilizations. If there are several recommendations covering events with the same priority, we choose the recommendation with the highest prioritized plan type. For instance, a plan for putting a device into a safe state would have a higher priority than a plan selecting a different interface module as input for data points. We decided to realize the recommendation decision maker with a timeout for giving various syndrome processors a chance to recommend a plan. The timeout starts with the first received recommendation. The drawback of this approach is that there is a fixed delay between a recommendation and the creation of plans. An alternative would be to create plans based on the first recommendation and to withdraw the plans if a higher priority recommendation is received before the final plan is executed.
Structure & Behavior: Algorithm 1 illustrates the steps taken by the recommendation decision maker. First, it selects the recommendations with the highest event priority. If there are several equal recommendations, it takes one with the highest plan type priority. The distinct prioritizations are determined beforehand.

6.4.5. Plan Maker

Responsibility: A plan maker creates plans, containing atomic actions for a given plan type. Concerning the distinct plan types, there exist corresponding plan creators that

```
Data: Recommendation[] recs
Result: Recommendation
Recommendation[] eventCandidates
int highestPriority = -1
foreach r in recs do
    if eventPriority(r) > highestPriority then
        eventCandidates.clear()
        eventCandidates += r
        highestPriority = eventPriority(r)
    else if eventPriority(r) == highestPriority then
        eventCandidates += r
    end
end
if eventCandidates.size() == 1 then
    return eventCandidates[0]
end
highestPriority = -1
Recommendation rec
foreach r in eventCandidates do
    if planPriority(r) > highestPriority then
        rec = r
        highestPriority = planPriority(r)
    end
end
return rec
```

Algorithm 1: Recommendation Decision Maker

are called by the plan maker. The plan creators leverage information provided by the recommendation and knowledge base.

Rationale: We added the plan maker to extract the creation of adaptations from the syndrome processors. This has the advantage that plans are reusable and that the corresponding plan creators verify whether an adaptation is possible. We decided that a plan maker can create more than one plan in order to enable algorithms that can create several distinct solutions.

Structure & Behavior: First, the plan maker chooses the corresponding plan creator. Plan creators are dedicated to one plan type. Then, the plan creator verifies the data part of the recommendation. Finally, the plan creator generates plans containing a set of actions. It should rely on the knowledge base for generating valid plans. The plan maker passes the created plans to the next phase of the Scari loop.

6.4.6. Plan Decision Maker

Responsibility: A plan maker can create distinct plans that realize a plan type in different ways. The plan decision maker selects one plan with the least affected Scari instances. If there are several candidate plans, it selects the one with the least amount of actions. Additionally, the plan decision maker locks the knowledge base.

```
Data: Plan[] plans
Result: Plan
Plan[] planCandidates
int leastAffectedScari = MAX_INT
foreach p in plans do
    if affectedScari(p) < leastAffectedScari then
        planCandidates.clear()
        planCandidates += p
        leastAffectedScari = affectedScari(p)
    else if affectedScari(p) == leastAffectedScari then
        planCandidates += p
    end
end
if planCandidates.size() == 1 then
    return planCandidates[0]
end
leastActions = MAX_INT
Plan plan
foreach p in planCandidates do
    if numberOfActions(p) < leastActions then
        plan = p
        leastActions = numberOfActions(p)
    end
end
return plan
```

Algorithm 2: Plan Decision Maker

Rationale: We extracted the comparison of plans from the plan maker in order to simplify the distinct plan creators. We decided to compare plans based on the affected Scari instances in order to change the least number of control devices. Similar, if there are several plan candidates we take the plan with the least amount of actions in order to change as little as possible. The locking of the knowledge base is necessary in order to prevent interferences from higher ranking Scari instances. Furthermore, the locking mechanism of the knowledge base ensures that it is synchronized with the lower layers. If the knowledge base is not synchronized, the plan decision maker withdraws the plan.
Structure & Behavior: Algorithm 2 illustrates the steps taken by the plan decision maker. First, it selects the plans which affect the least number of Scari instances. If there are several equal plans, it takes one with the smallest possible number of actions. The distinct prioritizations are determined beforehand.

6.4.7. Action Handler

Responsibility: The action handler executes plans and contained atomic actions. If an action targets a lower layer action handler, the action handler sends the action to the target Scari instance and waits until the action finished. All actions within a plan are executed

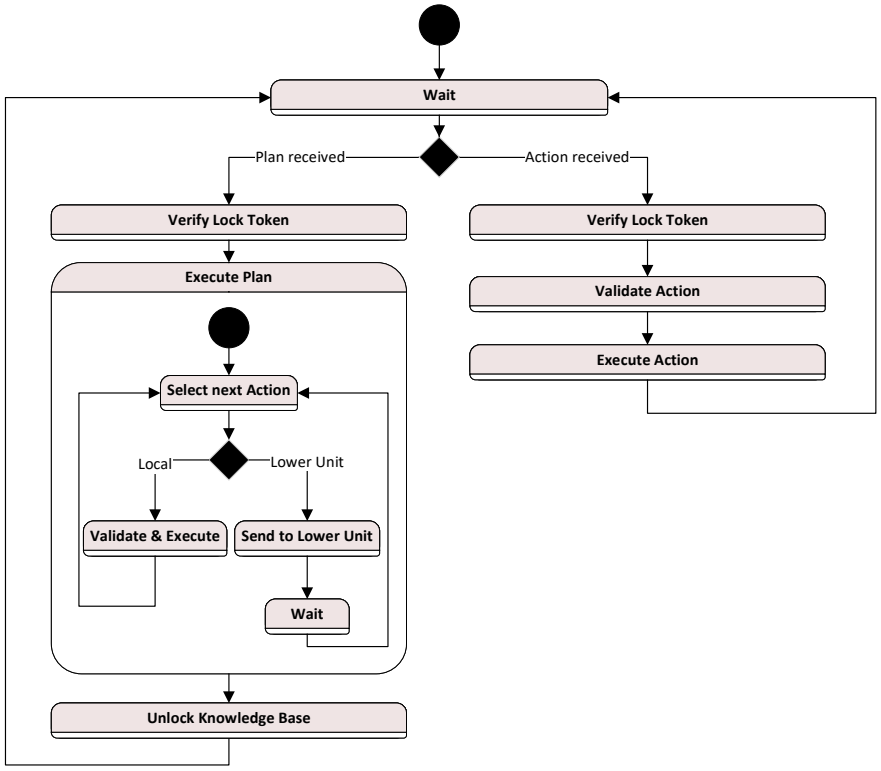


Figure 6.7.: State machine diagram of an action handler.

in a serial sequence. The single actions are responsible for changing the knowledge base. If a plan finished successfully, the action handler unlocks the knowledge base.

Rationale: This is the only entity part of Scari capable of actively changing the structure of devices. We deploy one action handler on each Scari instance. We realized the action handler in this way in order to make the execution of actions predictable and to prevent interferences.

Structure & Behavior: Figure 6.7 shows the distinct states of an action handler. An action handler reacts on plan and action messages. If it receives a plan, it verifies the lock token by asking the knowledge base. Afterwards, the action handler executes contained actions sequentially. If an action targets a lower unit, it is sent to a lower action handler and the sending action handler waits until the action finished. When an action handler receives an action to execute, it first verifies the lock token. Then it verifies the action, which means that the corresponding action logic revises whether all necessary data exists. Then the action handler executes the action. At the end of a plan, an action handler unlocks the knowledge base. If an action fails, the action handler aborts the plan and leaves the knowledge base locked.

6.5. Technical Implementation

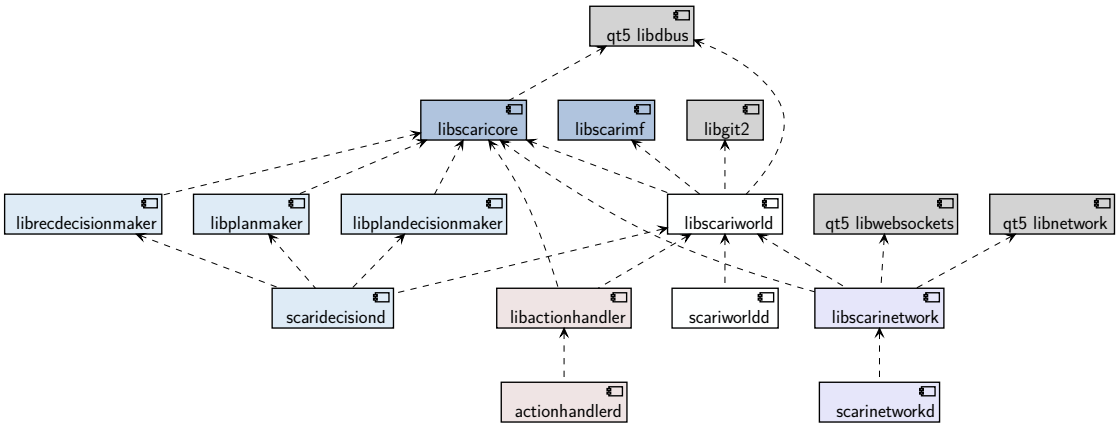


Figure 6.8.: Component diagram of the Scari software architecture.

We implemented our prototype of Scari in C++ using the Qt framework[1] version 5.7.1. We decided to do so because it is also the preferred way of developing software by our project partner. Figure 6.8 illustrates a component diagram of the Scari libraries and infrastructure daemon applications. We organized the application logic in libraries in order to have the opportunity to realize Scari within one or split into several independent applications. In the following we shortly present these libraries and the corresponding daemon applications. First we explain the core library, which is used by all other components except the Scari modeling framework. Then we present the components part of the *Decide* phase and of the *Act* phase. After that we describe the Scari modeling framework. Next, we explain the realized knowledge base containing the world model. Last, we shortly explain the networking between Scari loops.

6.5.1. Scari Core Library

libscaricore is the central library, which specifies the minimum number of classes necessary for communicating with the Scari infrastructure. It contains the classes and serialization functions for notification, recommendation, plan, and action messages. Figure 6.9 shows a class diagram of the different message types, which inherit from the parent abstract class *CausalElement*. The name of this abstract class comes from the containment *causes*. It enables to create a chain of messages in order to reconstruct why a notification, recommendation, plan, or action has been carried out. This information can be useful for syndrome processors. Furthermore, a recommendation decision maker decides based on the caused notifications. The *CausalElement* class contains different attributes that are

[1] https://www.qt.io/

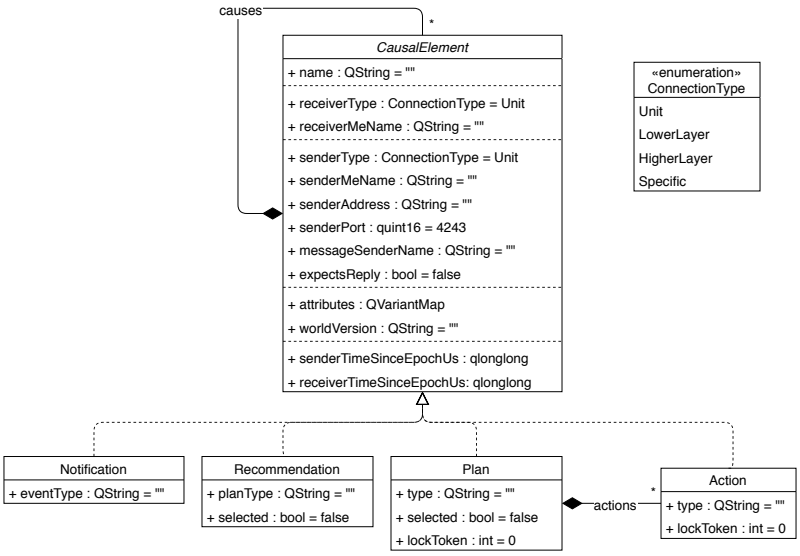


Figure 6.9.: Class diagram of the messages used by Scari.

used for specifying the receiver and sender in order to ease the delivery of messages. The field *attributes* can be used for attaching arbitrary data. The field *worldVersion* is mandatory for recommendations and plans for ensuring that they refer to the current world. At the bottom of Figure 6.9 are the different message types *Notification*, *Recommendation*, *Plan* and *Action*. Note that the classes *Plan* and *Action* contain a lockToken, which is necessary for changing the knowledge base.

In addition to these message types, *libscaricore* contains two classes that are used for receiving and sending messages over the message bus. We utilize Desktop Bus (D-Bus)[2], which is available and enabled in many Linux distributions by default. Essentially, it provides many-to-many communication. We use the qt5 dbus library for accessing the message bus. This allows us to loosely couple monitors, syndrome processors and the Scari infrastructure. At design time syndrome processors can be developed with respect to the available event and plan types. At any point during run time it is possible to start additional monitors and syndrome processors that connect seamlessly to the message bus. A downside of using D-Bus is that it does not guarantee any real-time constraints.

6.5.2. Decide Phase

librecdecisionmaker, *libplanmaker*, and *libplandecisionmaker* implement the different stages of the *Decide* phase. The event and plan type priorities of the recommendation decision maker are definable with JSON. It selects the recommendation based on the covered event

[2]https://dbus.freedesktop.org

with the highest priority. If there are equal recommendations it selects the one with the plan type of the highest priority. The plan maker uses the Qt5 plugin mechanism in order to load plugins that contain the logic for constructing a given plan type. The plan decision maker selects the created plan with the least amount of actions. These three libraries are used by the application *scaridecisiond*. Further it depends on *libscariworld* for locking the knowledge base. Table 6.3 enumerates events that can be distributed as notifications by *scaridecisiond* during the *Decide* phase. They are important for tracking recommendations and to react if something fails.

Event	**Description**
World Cannot Connect	*Scaridecisiond* cannot connect to the knowledge base.
Rec Invalid World Version	World version of the recommendation is outdated.
Rec Gets Evaluated	Attached recommendation gets evaluated.
Rec Selected	Attached recommendation is selected.
Plan Creator Does Not Exist	Plan creator for the plan type does not exist
Plan Invalid Rec	Recommendation does not provide the required metadata.
Plan Creator No Plans	Plan creator did not create plans.
Plan Creator Created Plans	Plan creator created plans.
Plan None Selected	No plan selected by the plan decision maker.
Plan Selected	A plan is selected by the plan decision maker.
Plan Lock Failed	Plan decision maker could not lock the knowledge base.

Table 6.3.: Event notifications distributed by the Decide phase.

6.5.3. Act Phase

libactionhandler implements the *Act* phase of the Scari loop and is utilized by the application *actionhandlerd*. Its purpose is to execute selected plans, manage the distribution of contained actions and finally execute actions. Every action is a single plugin and leverages the Qt5 plugin mechanism. Table 6.4 enumerates events possibly distributed during the execution of a plan.

Event	**Description**
World Cannot Connect	*Actionhandlerd* cannot connect to the knowledge base.
Plan Lock Token Mismatch	Lock token of the plan does not match the lock token of the knowledge base.
Action Does Not Exist	Action does not exist.
Action Lock Token Mismatch	Lock token of the action does not match the lock token of the knowledge base.
Action Invalid	The meta data of the action is invalid.
Action Failed	The execution of the action failed.
Action Successful	The action finished successful.
World Unlock Failed	*Actionhandlerd* could not unlock the knowledge base.
World Unlocked	*Actionhandlerd* unlocked the knowledge base.
Plan Failed	The execution of the plan failed.
Plan Successful	The plan finished successful.

Table 6.4.: Event notifications distributed by the Act phase.

6.5.4. Scari Modeling Framework

libscarimf realizes our modeling framework based on C++ and Qt5. We leverage the Qt framework because it adds reflection and meta information about objects to C++ in a platform-independent way.

We added macros that we use for defining attributes, references, and compositions to our modeling framework. Listing 6.1 illustrates an example definition of a ScariObject. Attributes are used for basic datatypes provided by the Qt framework. References are used for pointing to other ScariObjects. Containments make a ScariObject part of another. An object can only be owned by one other object. Each macro definition expands to setter and getter methods. A ScariObject is inherited from QObject and possesses all the features provided by the Qt framework. ScariObject definitions get automatically registered in a factory, realized as singleton, at the time the dynamic linker loads the surrounding library. Every ScariObject has a UUID as name in order to make them unique. We save ScariObjects as JSON. This is the point where the Qt reflection mechanisms come in handy. They allow to discover and call all attributes and methods of an object at run time in a generic way.

Listing 6.1: Definition of a ScariObject.

```
#include "scariobjectdefs.h"

SCARI_OBJECT(Entity)

    Q_OBJECT

    ATTRIBUTE_ONE(bool, active, false)
    ATTRIBUTE_N(int, values)

    REFERENCE_ONE(Entity, otherEntity)
    REFERENCE_N(Entity, otherEntities)

    COMPOSITION_ONE(Entity, ownedEntity)
    COMPOSITION_N(Entity, ownedEntities)

SCARI_OBJECT_END(Entity)
```

Listing 6.2: Instantiations of ScariObjects.

```
ScariObjectSet set;

Entity *e = set.createObject<Entity>();
Entity *e1 = set.createObject<Entity>();
Entity *e2 = set.createObject<Entity>();
Entity *e3 = set.createObject<Entity>();

e->set_active(true);
e->add_to_values(42);
e->set_otherEntity(e1);
e->add_to_otherEntities(e1);
e->set_ownedEntity(e2);
e->add_to_ownedEntities(e3);

ScarimfUtil::saveToDir(e, QDir::current());
```

Listing 6.2 shows how a ScariObject can be used by an application. *ScariObjectSet* takes care of the memory management and resolves references or containments. Listing A.1, found in the Appendix, contains the resulting JSON file. Note that we did not optimize the file size.

The main advantage of our modeling framework is that applications only need to reference modeling language libraries that they need to know. This allows to deploy different specialized modeling languages at different layers of Scari.

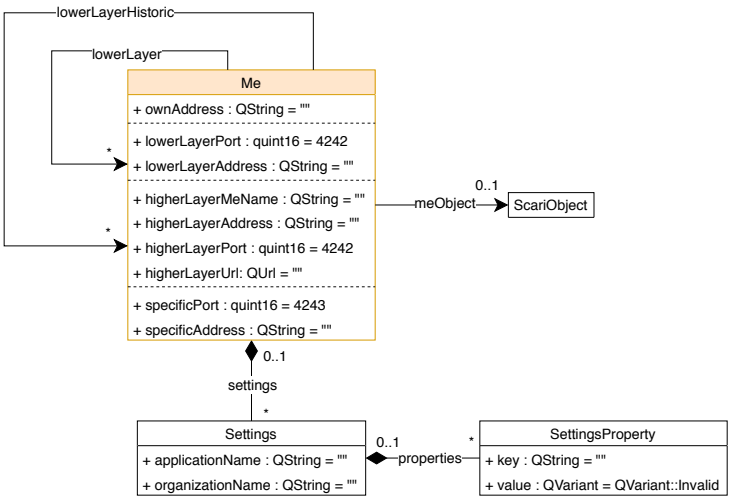


Figure 6.10.: Class diagram of the world metamodel.

6.5.5. Knowledge Base

The library *libscariworld* and the application *scariworldd* implement the knowledge base of the Scari loop.

Figure 6.10 illustrates the modeling language worldml utilized by *libscariworld*. The central element and starting point of every model exploration is the class *Me* representing the entity the corresponding Scari loop is running on. The attribute *ownAddress* contains the general IPv4 or IPv6 address of the entity supposed to be used by applications or monitors. *lowerLayerPort* and *lowerLayerAddress* contain the network settings for listening to connections from lower-layer Scari instances. The *higherLayer** attributes are the counterpart to the lower-layer settings and are used by lower-layer loops for connecting to a higher-layer Scari. The necessary settings for scarinetworkd are *higherLayerPort*, and *higherLayerAddress* or *higherLayerUrl*. The *higherLayerMeName* is automatically set by scarinetworkd and informs syndrome processors if there exists a higher layer. Additionally, it allows to detect whether the higher entity changed. The attributes *specificPort* and *specificAddress* can be used by higher layers in order to directly send actions to a Scari instance. The reference *lowerLayer* points to the current lower Me instances. This is a crucial reference for syndrome processors and plan makers in order to find lower-layer entities. The reference *lowerLayerHistoric* contains lower layers which have been connected once but are now disconnected. The containment *settings* can be used for storing settings specific to applications, monitors, syndrome processors, or Scari specific applications. We added the attribute *organizationName* to the class *Settings* to be compatible with operation systems such as Windows where application settings are supposed to be stored in the system registry. Last but not least, the reference *meObject* points to the current object

that represents the entity in detail. By referencing the class *ScariObject* it is possible to utilize any type defined with the Scari modeling framework. The *meObject* is not used by *scariworldd*. Therefore, it does not need to be aware of other modeling languages than worldml.

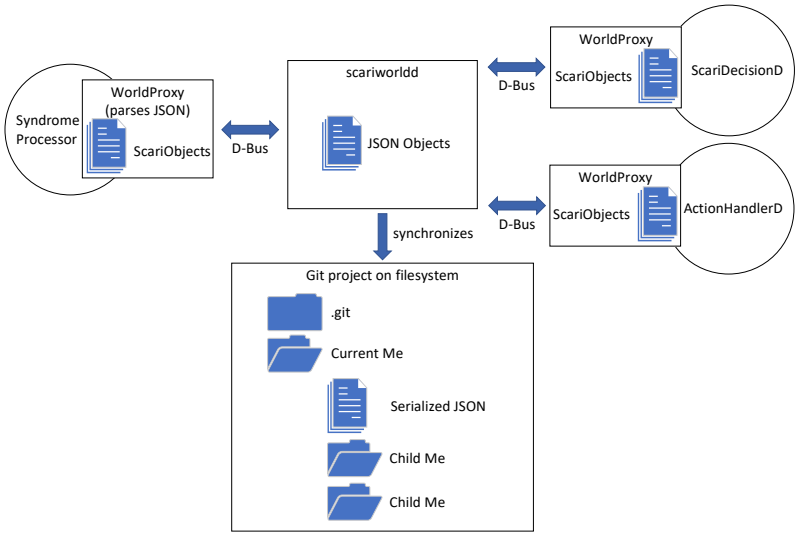


Figure 6.11.: Overview of a scariworldd instance.

Figure 6.11 illustrates an overview of what *scariworldd* implements. All *scariworldd* operations are accessible over D-Bus as a service. Syndrome processors, plan creators and actions can retrieve arbitrary model elements managed by *scariworldd*. Usually the model discovery starts with an object of type *Me* illustrated above in Figure 6.10. *scariworldd* holds ScariObjects as JSON objects in memory. The JSON objects get transformed to ScariObjects at the entities which are interested in the content. *scariworldd* stores the JSON objects as files in a git repository.

scariworldd manages the contained models, updates the world version, restricts access, and synchronizes the models between different layers. The access can be restricted by locking a world and its lower layer worlds. *scariworldd* takes care of that by sending lock notifications containing a lock number and the world version of the target world to the lower layer. The target world version is important because otherwise it could happen that a parent loop operates on an outdated world and would try to apply a destructive adaption.

We synchronize the worlds with git, leveraging the libgit2[3] implementation. Serialized models are stored as JSON files and as compressed objects in a git repository. Each change of the world is reflected by a commit and it is possible to analyze the history of

[3]https://libgit2.org/

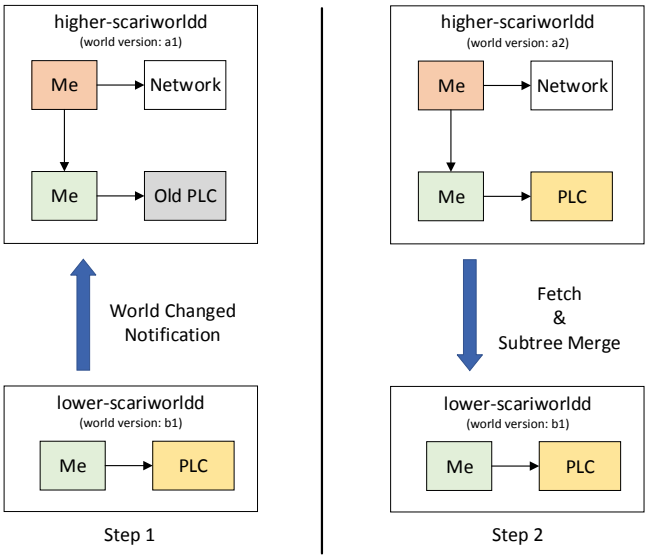


Figure 6.12.: Synchronization of the world models.

a Scari loop. Furthermore, it would even be possible to rollback a world. In addition to the model elements, the repository stores the messages sent over D-Bus. Each repository obtains a *world* tag which is incremented to the latest commit if a model element changes. The commit id targeted by the *world* tag equals the world version used by the Scari framework. Figure 6.12 shows how worlds become synchronized between layers. A change of the world leads to a notification that is sent to the higher layer world. The higher-layer world is responsible for fetching and merging the data. We perform a subtree merge, which incorporates the child tree into the master tree of the merging repository. The resulting directory structure of the repository reflects the hierarchical organization of Scari as illustrated in Figure 6.11. Furthermore, *scariworldd* parses the changed JSON files and manages the current *Me* object.

It is possible that the synchronization of a world takes longer than a syndrome processor would need to react to a notification from a lower layer. In our prototype implementation, *scaridecisiond* would not be able to lock the target world and its children if the world version of the recommendation does not match the version of the world. If the synchronization has not been carried out at the time *scaridecisiond* locks the world, then the unsynchronized child world would reject the receiving lock from the higher layer. The drawback of our prototype implementation is that a syndrome processor needs to reevalute the situation if it has been faster than the git synchronization mechanism. Table 6.5 enumerates event notifications distributed by *scariworldd*.

Event	Description
World Changed	The knowledge base was changed. This is important for other Scari entities.
World Lock	Used internally by *scariworldd* for locking the child knowledge bases.
World Unlock	Used internally by *scariworldd* for unlocking the child knowledge bases.

Table 6.5.: Event notifications distributed by the knowledge base.

6.5.6. Networking

libscarinetwork implements the functionality of distributing messages between layers and Scari loops. It is utilized by *scarinetworkd*. Figure 6.9, shown above, illustrates that we distinguish between *Unit*, *LowerLayer*, *HigherLayer* and *Specific* as message receiver types. *Unit* targets the local D-Bus. All other receiver types are managed by *scarinetworkd* in order to support the loose coupling of Scari components. *scarinetworkd* obtains the network settings from the local *Me* object. It communicates with other Scari loops over WebSockets. The WebSocket protocol is located at layer 7 in the OSI model and depends on TCP at layer 4 [110]. We decided to leverage it because it is small, simple to use, and well supported by Qt5 and many other frameworks. Table 6.6 enumerates events distributed by *scarinetworkd*.

Event	Description
Network Send Failed	*Scarinetworkd* could not send the message.
Network Websocket Error	A websocket error occurred.
Network Websocket SSL Error	A websocket SSL error occured.
Network Disconnect Socket	A websocket disconnected.
Network Connected Socket	*Scarinetworkd* connected a websocket to another Scari instance.

Table 6.6.: Event notifications distributed by the network entity.

6.6. Experiments

For evaluating and examining different mechanisms with Scari, we built a testbed consisting of eight Raspberry Pi 3 Model B devices running the Raspbian distribution based on Linux kernel version 4.9.59-v7+. Figure 6.13 illustrates the test setup.

Each side of the testbed consists of four Raspberries with a central power supply located in the middle. On top of the hardware stack resides a Netgear FS116 10/100 MBit/s Ethernet switch. Utilizing affordable Raspberry Pis for our testbed has the advantage that other researchers have the opportunity to yield comparable results for the following scenarios. Furthermore, the Raspberry Pi 3 Model B runs a 1.2 GHz Quad-Core ARM Cortex-A53 processor, offers an extended 40-pin GPIO header, one Ethernet port, four USB 2.0 ports, Bluetooth 4.1 and 802.11 b/g/n Wireless LAN. This allows us to simulate a network of PLCs that could communicate over various redundant channels and control simulated physical mechanisms.

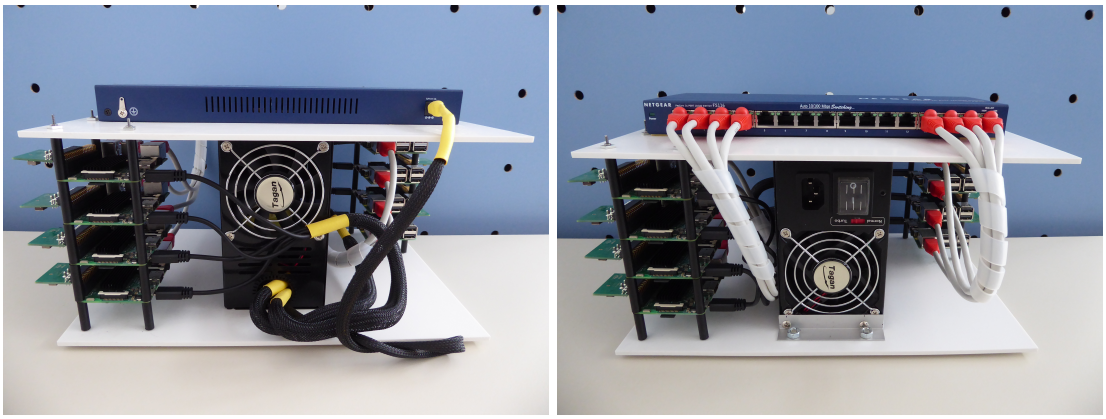


Figure 6.13.: Raspberry Pi Model B testbed equipped with Infineon Iridium 9670 TPM add-on boards.

In the following four subsections we give details of scenarios where Scari is applied. Subsection 6.6.1 utilizes the proprietary PLC software from our project partner and we artificially inject permanent DRAM memory faults through an extended version of the emulator software Quick Emulator (QEMU). Subsection 6.6.2 presents a security-related scenario where PLCs mutually attest themselves whether an unknown software is started. Subsection 6.6.3 shows the remote attestation case in an extended situation where a higher layer needs to adapt. Subsection 6.6.4 illustrates the case that one PLC experiences a data point mismatch and another PLC also has to react.

We measure the execution time of all steps, the number and sizes of all messages, and the different git repository sizes. We selected these metrics because they give a good indication of the performance overhead of Scari. The provided execution times are based on the arithmetic mean of ten single unrelated experiments.

6.6.1. Proprietary PLC Memory Fault

Figure 6.14 illustrates an experiment of Scari in our industrial setting with the proprietary software from our project partner. We implemented a memory-injection mechanism into QEMU[4] to change memory content during runtime. Our implementation is based on previous work from our research group [111]. We use this manipulated QEMU to simulate stuck-at memory errors at known memory locations.

Now, the scenario is that a permanent stuck-at memory error happens at the memory location of a data point. Table 6.7 illustrates the used plans and actions.

A monitor performs a cyclic check of the input and output data points of a FUP. If it encounters a wrong data point value, it notifies the Scari applications. Then a syndrome

[4]https://github.com/jib218/scari-qemu

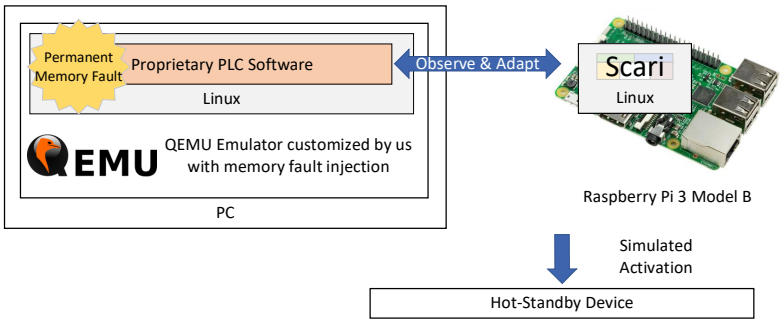


Figure 6.14.: Proprietary PLC memory fault.

Situation	Plan	Actions
Data Point Mismatch	Increase Data Point Monitoring	• Activate Hot-Standby PLC • Increase Monitoring
Located Memory Area	Test Memory	• Test Memory
Located Faulty Memory Area	Mask Memory	• Mask Memory • Reboot ACPU
No Data Point Mismatch	Decrease Data Point Monitoring	• Set Self Hot-Standby • Decrease Monitoring
Emergency	Safe State	• Activate Hot-Standby PLC • Stop PLC • Notify Higher Layer

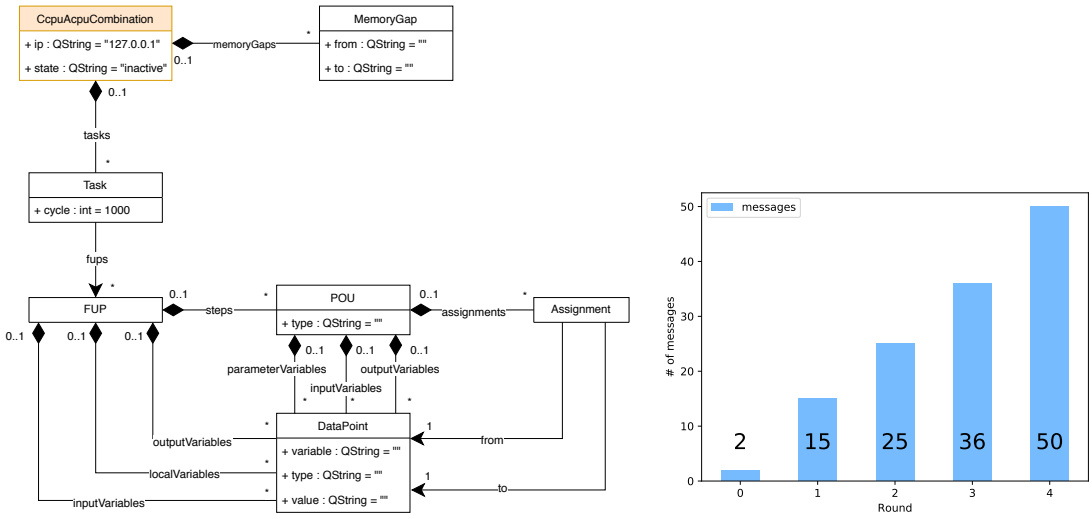
Table 6.7.: Available plans and actions of the PLC memory fault scenario.

processor recommends activating the hot-standby device and increasing the data point monitoring, which means that also data points inside of a FUP are monitored. Then the monitor compares all data points of a FUP with an oracle. Based on such an event, the syndrome processor recommends performing memory tests on the addresses of the affected data points. Depending on the results, the syndrome processor may recommend to mask the faulty memory areas with the Linux kernel boot parameter *memmap*[5]. *memmap* enables to mark specific memory addresses as reserved which are thus not assigned by Linux. The physical memory addresses can be obtained in Linux by using the *pagemap*[6] functionality. Masking a faulty memory area in Linux requires a reboot. After the reboot, the monitor can verify whether the data points are still incorrect. If not, the syndrome

[5]https://www.kernel.org/doc/html/v4.14/admin-guide/kernel-parameters.html
[6]https://www.kernel.org/doc/Documentation/vm/pagemap.txt

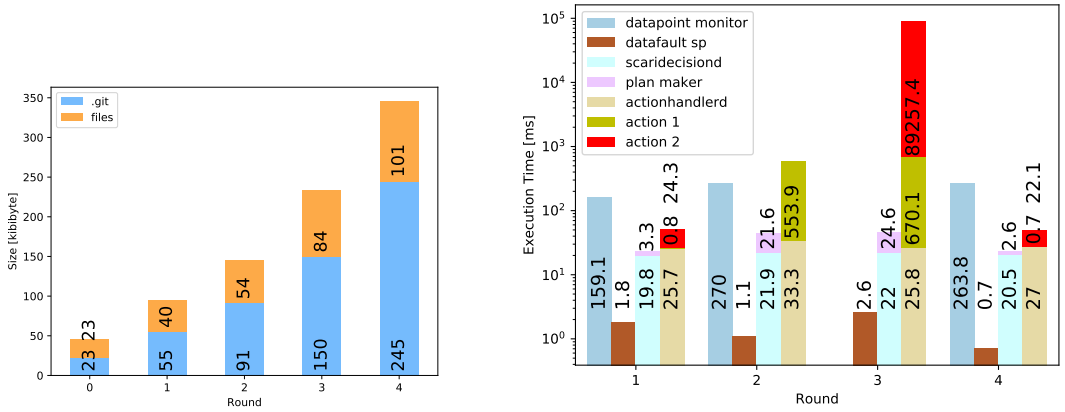
processor can recommend to set the device itself to hot-standby mode and decrease the monitoring.

All together these adaptions take four rounds. The detailed explanation of how the monitor, datafault syndrome processor, and plans work can be found in **Paper G** [112].



(a) Class diagram of the proprietary PLC metamodel.

(b) Amount of messages.



(c) Size of the git repository.

(d) Execution times of the different rounds.

Figure 6.15.: Class diagram and measurements of the proprietary PLC scenario.

Figure 6.15a illustrates the modeling language we created for this scenario. The class *CcpuAcpuCombination* represents our special situation where we are leveraging a Raspberry Pi to orchestrate the proprietary PLC software running on top of QEMU. The

attribute *ip* specifies the target address of the PLC software and the attribute *state* the current mode. The contained objects of type *MemoryGap* match the masked memory areas. We add one *MemoryGap* object during the execution of the third plan. The contained objects of type *Task* represent the executed logic of the PLC. The class *FUP* specifies the single steps represented by objects of type *POU*. Both classes can specify input-, local-, and outputVariables that are objects of type *DataPoint*. The class *Assignment* is used by *POU* objects for setting input and output variables. We designed this modeling language close to the IEC 61131 structure used by our project partner [92].

In our scenario we are using one *Task* and *Fup* object. The *Fup* object owns two steps consisting of two *POU* objects, each of them realizing a boolean *not*. The memory error happens at a linking data point between the two *POU* objects. Additionally, the *Me* object owns two *Settings* objects for the monitor and syndrome processor.

Figure 6.15b illustrates the amount of messages distributed over D-Bus during the execution of single rounds. Round 0 refers to the amount of messages before the first event happened. Figure 6.15c shows how the size of the git repository managed by scariworldd behaves. *.git* is the place inside a repository where git stores all configurations, commits, tags, trees, and objects. *files* refers to the json models and messages saved to the file system at the end of a round. Figure 6.15d contains the execution times of the single applications, plan plugins, and action plugins during each round on a logarithmic scale in milliseconds. The data point monitor needs to communicate with the proprietary PLC with XML commands send over TCP/IP. The execution times of scaridecisiond and actionhandlerd mainly consist of sending notifications and locking/unlocking the world. In round one, the first action consists of sending a notification to the data point monitor and the second action changes the state of the PLC object. In round two, the action performs a memory test at the target memory area. In round three, the first action executes a script at the target Linux instance in order to mask the memory area. The second action starts a reboot and waits until the PLC is reachable again. In the last round a notification is sent to the data point monitor and the state of the PLC object is changed to be *hotstandby*.

Note that these execution times do not include the delays introduced by D-Bus. We measured in this scenario, that a delay between the sender and receiver of a message can take up to 5 milliseconds. Also, Figure 6.15b does not include the time scaridecisiond waits until it evaluates the recommendations. This is definable by the user. In our scenario it would make sense to wait less than a second because the datafault syndrome processor only takes up to 2.6 milliseconds. However, a good waiting time depends on the number of syndrome processors and the longest calculation time.

With this experiment, we demonstrated that Scari is suitable for dealing with a permanent memory fault scenario. The execution time of the single Scari phases is quite low compared to the time a monitor needs for retrieving a data point from the observed ACPU. Figure 6.15b illustrates that at each round of Scari about ten to fourteen messages are distributed by the distinct parts. This is necessary for ensuring that a syndrome processor can log the state of its recommendation. Because of these messages, also the size of the

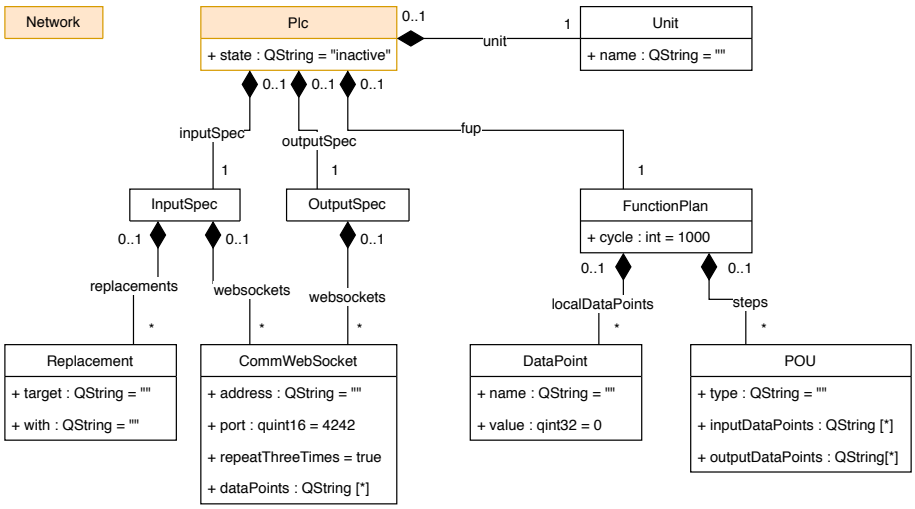


Figure 6.16.: Class diagram of the simplified PLC metamodel.

git repository grows from 46 kibibytes to 346 kibibytes. This growth is not as bad as it sounds because Scari only interacts in unusal situations.

6.6.2. Remote Attestation

Here, we present a remote attestation case of how Scari could deal with a security-related violation of the integrity. Attestation is a security method where an attestator proves to a challenger, locally or remotely, that used files and binaries correspond to their signatures [106]. In contrast to the scenario above, we use a self-built software implementing the behavior of a simple PLC. The software can execute function plans containing components that perform simple operations such as an *add* or *not* triggered by a definable cycle. We only support data points representable by 32-bit integers. Furthermore, our PLC simulator can send data points to other PLC simulators over the WebSocket protocol, which eases for us to conduct experiments. Because the PLC simulator is only focused on simple control logic, it lacks many features compared to the proprietary one from our project partner. The benefit is that this eases for us to inject failures and conduct experiments.

Figure 6.16 illustrates the used modeling language. The class *Network* represents the higher layer Scari instance. The class *Plc* contains an object of type *Unit* that represents a simplification of the connected hydropower unit. Objects of the types *InputSpec* and *OutputSpec* are contained within *Plc* and specify what data points are received and sent over websockets. If a *CommWebSocket* is used by an *InputSpec* the PLC application starts a websocket server that listens for incoming connections. If it is used by an *OutputSpec* the PLC application tries to connect to the target address and port. We included the option to send a data point three times because this behavior is required in the hydropower domain.

Additionally, the class *InputSpec* utilizes objects of type *Replacement* for substituting data points with redundant ones. We use this class in the first case for switching the used data points. The last type contained in the class *Plc* is the class *FunctionPlan*. It posesses an attribute *cycle*, a containment *localDataPoints* and *steps* consisting of *POU* objects. Data points are created with this modeling language by defining them inside *CommWebSocket* objects, through local data points of a *FunctionPlan*, or by naming *outDataPoints* in *POU* objects.

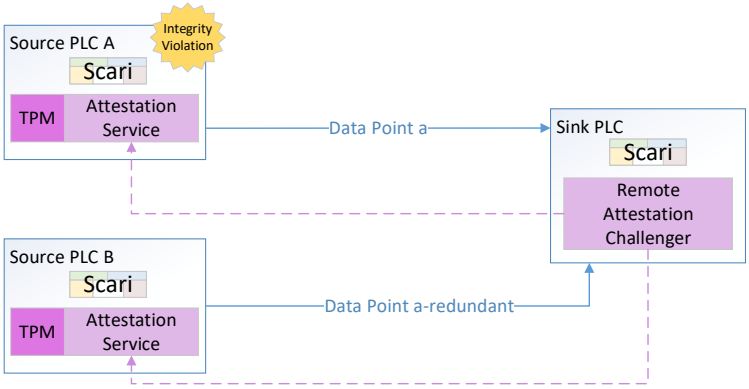


Figure 6.17.: Remote attestation scenario

Figure 6.17 illustrates the scenario where Source PLC A and Source PLC B are sending the data point *a* to a Sink PLC. The Source PLCs are proving to the Sink PLC that they are not manipulated by an attacker and therefore the data points are trustable. To do so we leverage a modified Linux Integrity Measurement Architecture subsystem that detects if files are altered. It hashes and verifies files by using an Infineon Iridium 9670 TPM. Figure 6.13 above shows these TPMs attached to our testbed. This ensures, together with a form of secure boot mechanism, that an attacker is not able to manipulate the security integrity mechanism. Now, if an untrusted software is started on a PLC, e.g., on Source PLC A, the integrity mechanism would inform the local attestation service. This service notifies the observing remote attestation challengers. The remote attestation challenger, located on the Sink PLC, notifies the local syndrome processors.

Table 6.8 enumerates the available plans. In the case illustrated in Figure 6.17, the syndrome processor recommends to replace the data point *a* with *a-redundant* which also adapts the model accordingly.

Figure 6.18a shows the amount of messages sent over D-Bus at the Sink PLC before and after the PLC got adapted. Figure 6.18b illustrates the size of the *.git* directory and the current files. The increase of the size is caused through the added object of type *Replacement* and the sent Scari messages.

Situation	Plan	Actions
Failed Attestation	Switch Variables	• Add Replacement
Failed Attestation	Notify Higher Layer	• Send Notification

Table 6.8.: Available plans and actions of the remote attestation scenario.

Figure 6.18c provides a detailed overview of the single execution times. *A attestationd* is the time from being notified by the integrity mechanism and the start to informing the attestation monitors. *action 1* involves the configuration of the PLC software and the addition of the *Replacement* object to the model. The action communicates with our PLC software over a socket connection. Note that the measured execution times are worse than in the proprietary PLC memory fault scenario above. This is due to the fact that the used *Plc* model element contains all configurations while the *CcpuAcpuCombination* model element from above references *FUP* objects. Therefore, the transferred model sizes over D-Bus are bigger. An additional possible reason is that there are more applications executed on the Raspberry Pi and it also needs to deal with several network connections.

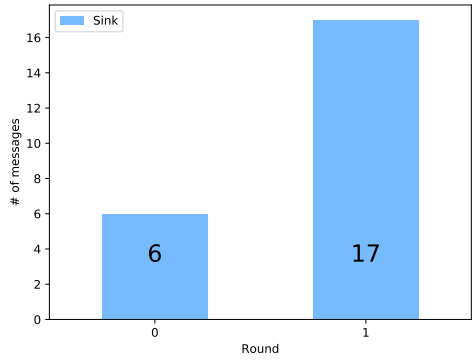
This experiment demonstrated how Scari can behave in a security related scenario. Note that a control device can deal with this situation on its own without a higher layer. The execution times of the single activities are neglectable but could be further optimized. The size of the git repository doubled. This is because the *Plc* object contains all other objects. A model where the objects are referenced and not contained would optimize this situation.

6.6.3. Remote Attestation Higher Layer

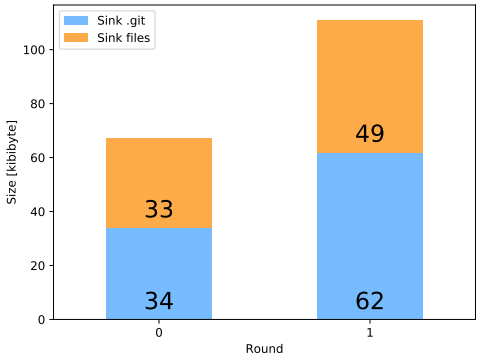
In this scenario we are leveraging a higher layer for adapting to a security-related violation of the integrity of a PLC. We are reusing the modeling language shown above in Figure 6.16.

Figure 6.19 shows the scenario. Source PLC B is now a control device, which happens to calculate the same data point similar to the way it is provided by Source PLC A. However, at the beginning of the scenario it is not sending its data point to the Sink PLC.

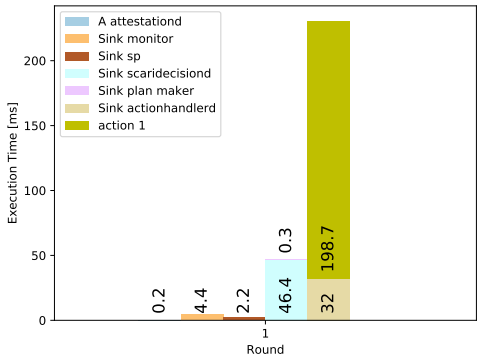
Table 6.9 enumerates the used plans and actions. Now, the examined situation is that Source PLC A experiences an integrity violation. The Sink PLC is not able to replace the affected data point on its own. Therefore, it notifies the higher layer. The syndrome processor residing on the Supervisory Computer recommends a plan that involves stopping the Source PLC A and the Sink PLC. After these first actions, the Source PLC B is configured by an action to send the data point a to the Sink PLC. Then an action starts a new remote attestestion monitor on the Sink PLC targeting Source PLC B. The last action is to start the Sink PLC again and to let the control system continue.



(a) Amount of messages.



(b) Size of the git repository.



(c) Execution times.

Figure 6.18.: Measurements of the remote attestation scenario.

Situation	**Plan**	**Actions**
Failed Attestation	Notify Higher Layer	• Send Notification
Failed Attestation	Isolation	• Stop PLC • Add OutputWebSocket • Add Attestation Monitor • Start PLC

Table 6.9.: Available plans and actions of the extended remote attestation scenario.

Figure 6.20a illustrates the messages sent over the single Scari loops. In round one the Sink PLC notifies the higher layer through a plan. Whether it is necessary to do this through a recommendation, is a design choice. In round two the higher layer recommends

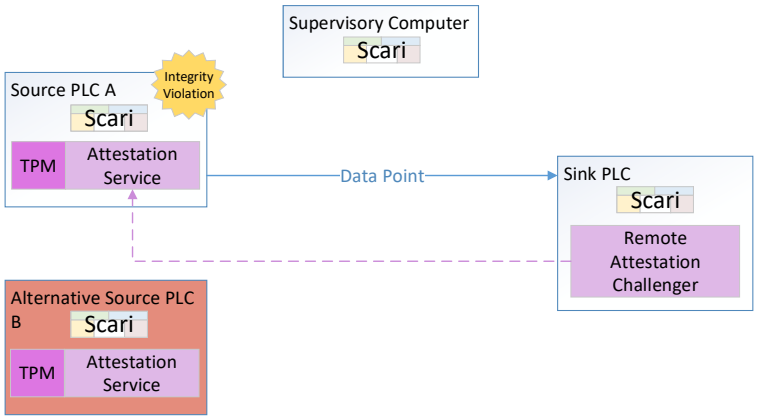


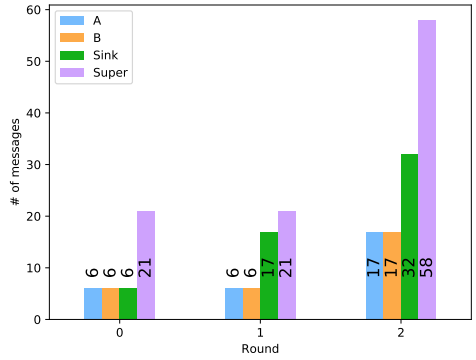
Figure 6.19.: Remote attestation scenario with a higher layer.

to isolate Source PLC A. The higher layer sequentially sends actions to the lower layer Scari instances. This leads to the same amount of messages on both Source PLCs because each executes one action. The Sink PLC executes three actions.

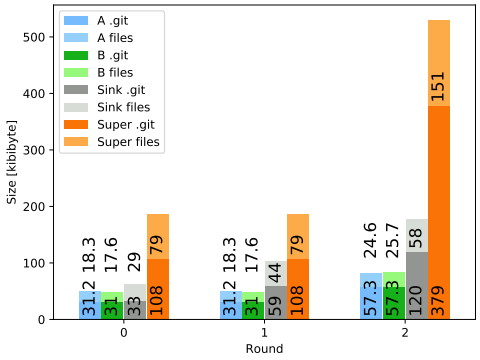
Figure 6.20b shows the behavior of the git repositories. In round zero and one the repository of the Source PLC A is slightly bigger than the repository of Source PLC B. This is because Source PLC A obtains an object of type *CommWebSocket*. It changes in round two because Source PLC B also receives an object of type *CommWebSocket*.

Figure 6.20c highlights the execution times of the first and second round. In the first round, *A attestationd* resides on Source PLC A and the Sink PLC responds to the security violation by sending a notification to the higher layer through *Sink action 1*. In the second round, *Super sp* located on the Supervisory Computer recommends to isolate Source PLC A. The whole adaption takes 3245,7 milliseconds including the network communication between the Scari loops. The plan consists of the following actions: First the Sink PLC is stopped with *Sink action 1*. Then the Source PLC A is stopped with *A action*. Next, Source PLC B receives a new output websocket with *B action*. Finally, the Sink PLC starts a new remote attestation monitor targeting Source PLC B with *Sink action 2* and the Sink PLC continues with *Sink action 3*.

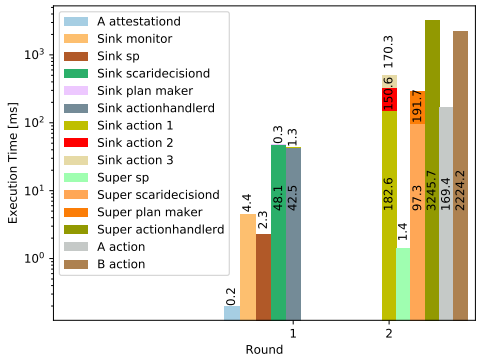
This experiment shows that the hierarchical organization of Scari enables to deal with situations that can only be resolved by an entity overlooking several PLCs. A drawback is that the git repository of the supervisory computer grows because it incorporates the information of the lower layers. In this case, a compressing or pruning algorithm for the repository could remedy the situation.



(a) Amount of messages.



(b) Size of the git repository.



(c) Execution times.

Figure 6.20.: Measurements of the extended remote attestation scenario.

6.6.4. Data Point Mismatch

In this scenario a data point mismatch between redundant PLCs happens. It shows how Scari can deal with parallel adaptations on distinct PLCs. We are reusing the modeling language shown above in Figure 6.16.

Figure 6.21 illustrates three Source PLCs and one Sink PLC. These PLCs are observed by one Supervisory computer. Each of the three Source PLCs is generating the same data point a, which are sent to the Sink PLC. Additionally, they send the data points to each other in order to verify if their calculated data point is correct.

Table 6.10 enumerates the available plans and actions. Now what happens in this scenario is that the Source PLC A recognices a mismatch from its own data point. At about the same time the Sink PLC recognices that the data point from Source PLC A is

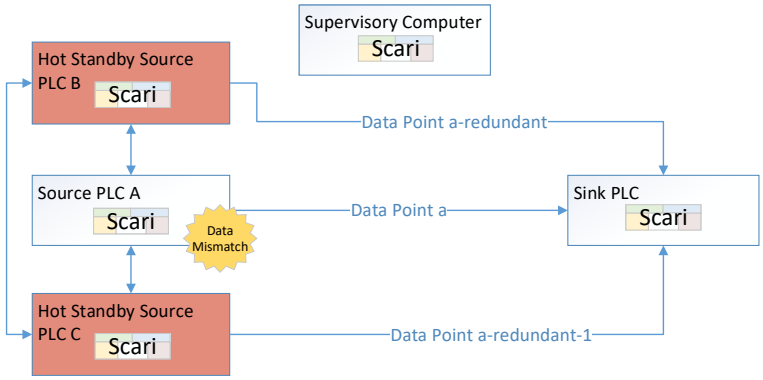


Figure 6.21.: Data mismatch happening on Source PLC A

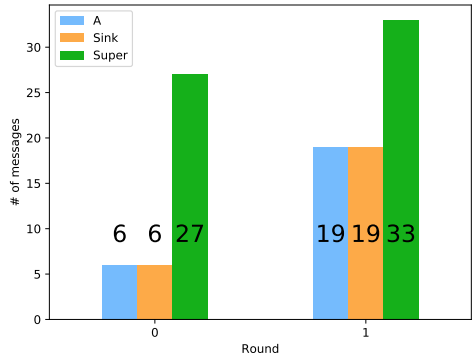
Situation	Plan	Actions
Data Point Mismatch	Safe Stop	• Stop PLC • Send Notification
Data Point Replaced	Add Model Object	• Add Replacement Object • Send Notification

Table 6.10.: Available plans and actions of the data point mismatch scenario.

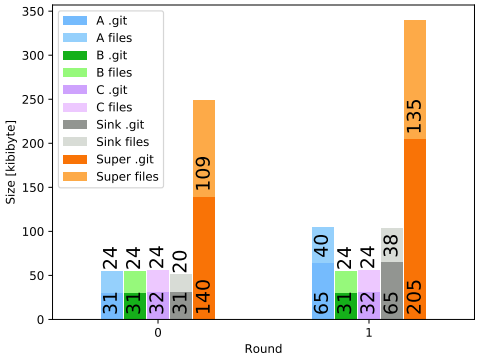
incorrect. These detection capabilities are realized within our PLC simulation software. Source PLC A distributes a notification to its local Scari instance that a data mismatch occurs. The syndrome processor recommends stopping the PLC and sending a notification to the higher layer. The Sink PLC immediately reacts on its own by replacing the faulty data point a with a redundant one. After this reaction, the Sink PLC sends a notification to its local Scari instance. The corresponding syndrome processor recommends adding the replacement to the world model, which also makes the replacement persistent over reboots. The plan also involves to notify the higher layer. At this point in the scenario we stop. The supervisory instance could now try to repair the Source PLC A through software and inform engineers.

Figure 6.22a shows the messages residing on the Source PLC A, the Sink PLC, and the Supervisory Computer. The two PLC Scari rounds take the same amount of messages. Six messages appear at the Supervisory Computer. Two are the notifcations from the PLCs. Another two are “World Changed” notifications from the lower layer. They lead to two more local “World Changed” notifications because the higher layer world needs to synchronize.

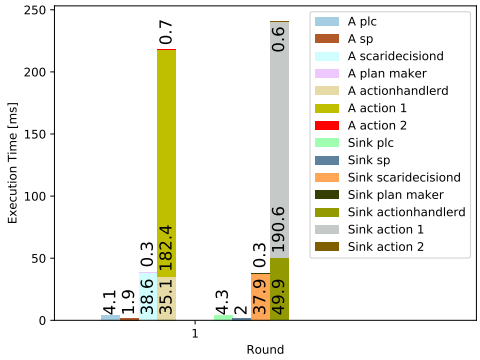
Figure 6.22b illustrates the behavior of the git repositories of all PLCs and the Super-



(a) Amount of messages.



(b) Size of the git repository.



(c) Execution times of the separate Scari loops.

Figure 6.22.: Measurements of the data point mismatch scenario.

visory Computer. Each PLC and the Supervisory Computer obtains models based on the modeling language shown in Figure 6.16 above. The Supervisory Computer is represented by an object of type *Network*. The PLCs contain function plans that send and receive the data points. The objects of type *Me* also contain settings for the syndrome processors and PLC software.

Figure 6.22c contains the execution times of the Scari loops residing on the Source PLC A and the Sink PLC. The left-hand bars describe the Scari loop of the Source PLC A. *A plc* is the time of the PLC software that it takes for recognicing the mismatch until the notification is sent. *A action 1* refers to stopping the Source PLC A and updating the model. *A action 2* executes the notification of the higher layer. The right-hand bars depict the Scari loop of the Sink PLC. *Sink plc* is the time of the PLC that it takes for adding the replacement and sending the notification. *Sink action 1* adds an object of type

Replacement to the model. *Sink action 2* notifies the higher layer.

This scenario shows that Scari is able to deal with a situation that needs to be handled at several places at the same time. We moved the direct reaction of the Sink PLC directly into the PLC software in order to deal with the situation immediately. This could also be achieved through a Scari loop round but would need much longer. The advantage compared to a centralized Scari instance handling all PLCs at the same time is, that a layer of separate Scari instances can adapt faster. If a problem can only be handled by one instance overlooking all PLCs, the individual Scari instances can escalate to a higher layer. In our case, this would be the Supervisory Computer Scari instance.

6.7. Meeting Requirements

REQ1 open for arbitrary and existing detection mechanisms: Detection mechanisms are called monitors in Scari. They do not need to use the knowledge base and do not rely on other entities.

REQ2 minimal effort for adding detection mechanisms: A detection mechanism just needs to be extended with the functionality to distribute an event notification for being part of Scari. These notifications should include the current world version, which is also distributed as notification by the knowledge base.

REQ3 parallel coexistance of distinct reasoning mechanisms: In Scari, an arbitrary number of syndrome processors with their own technology concerning reasoning can exist. This requirement is the main reason why we chose OODA as underlying concept. OODA offers an orientation phase which is represented by syndrome processors and a dedicate decide phase.

REQ4 notify distinct reasoning mechanisms at once: We support this one-to-many communication style through the use of a message bus.

REQ5 recommend a plan for dealing with a situation: With the explicit message type "Recommendation", a syndrome processor proposes a plan type and necessary meta data.

REQ6 choose one recommendation: The recommendation decision maker selects one recommendation based on predefinable priorities.

REQ7 atomic operations: The action handler executes single actions in a serial manner. Each action must fulfill only one purpose.

REQ8 combine atomic operations arbitrarily: A plan consists of arbitrary actions in order to fulfill the adaptation.

REQ9 adaptations must not interfere with each other: We ensure this requirement by locking the knowledge base and the children. Thus, lower Scari instances are not able to perform adaptations while a higher one is active.

REQ10 plans are reuseable: Syndrome processors can only recommend plans and plans are reusable.

REQ11 applicability is checked during plan creation: This is ensured by the implementation of the individual plan creators.

REQ12 react to threats on its own: We realize this requirement by proposing a decentralized and hierarchical system. Scari instances at the same layer or in distinct hierarchies can react on their own.

REQ13 react at the same time: This is possible for Scari instances at the same hierarchical layer while a higher Scari instance is not locking its graph.

REQ14 escalate to higher layers with more capabilities: This requirement is ensured by the hierarchical organization.

REQ15 lower layers should not be able to adapt higher layers: In Scari lower devices can only send notifications to higher layers.

REQ16 knowledge base: We propose decentralized and hierarchical knowledge bases that provide the architectural information as models.

REQ17 reflect the current state: Actions must adapt the knowledge base accordingly to their adaptations. The knowledge base does not ensure on its own that it contains the right information.

REQ18 a consistent knowledge base should be deployed on each device: Each device maintains its own knowledge base.

REQ19 knowledge bases get synchronized between layers: If a knowledge base is changed, it notifies the higher layer. The higher layer pulls the changed model data from the lower layer.

REQ20 distinct kinds of models: We ensure this in our implementation by proposing a modeling framework that allows to define specialized modeling languages for distinct kinds of devices or layers.

REQ21 combine models arbitrarily: Through the use of a world modeling language that is present in every knowledge base and a model object named *Me*, we can deploy any models on a device. A Scari entity can ensure on its own that it is compatible. Our proposed modeling framework allows to combine distinct modeling languages arbitrarily.

6.8. Discussion of Limitations

In this Section, we discuss limitations of Scari and reflect on important design decisions. We address the following issues and limitations:

- *Hierarchical control* as an important design decision for Scari.
- Pros and cons of *utilizing architectural models* as reflection of the device and configuration.
- Limitations concerning *syndrome processors*.
- Dealing with failed *adaptations* and the possible preemption of them.
- Setting up the *configuration of Scari*.
- *Waiting time* of the recommendation decision maker.
- Possible *real-time* behavior of the Scari framework.
- Dealing with *human intervention* at run time.
- *Security of Scari* itself.
- Discussion of our decision to take *git* as underlying mechanism for the knowledge base.

6.8.1. Hierarchical Control

We decided to engineer Scari in accordance with the idea that each device can react on its own or escalate to a superiour entity. This increases the utility of our system as control devices in our setting can be deployed in distinct combinations. Furthermore, we decided that each Scari loop maintains its own knowledge base. An alternative to our distributed knowledge bases would be a centralized knowledge base. One advantage of our architecture is that there can happen multiple adaptations of distinct devices on lower layers at the same time. Another advantage is that the adaptation time is potentially lower because there is less communication overhead. The major drawback of our approach is the synchronization needed for the knowledge bases. A higher layer cannot lock if it does not operate on a fully synchronized knowledge base. We consider this to be necessary to prevent wrong recommendations and incorrect creations of plans. With an engineering effort, the knowledge bases could be centralized and the single Scari instances would leverage cached information. This would eliminate the synchronization steps between each layer but would require synchronizing caches. Also, the centralized knowledge base would need to maintain different world versions for keeping the feature of multiple adaptations of

distinct devices. Thus, it essentially would accelerate the synchronization between layers but would still include a management overhead.

A complete opposite of our proposed hierarchical organization of adaptations loops would be one centralized loop. In theory, this is possible with Scari. The single devices could just host monitors and syndrome processors, while a central recommendation decision maker decides what to do. However, it would require quite an engineering effort of enabling multiple adaptations at different locations at the same time. This is also the reason why centralized self-adaptive systems found in the literature only allow one adaptation at a time.

6.8.2. Utilizing Architectural Models

A model is always a model of something. In our case, the architectural models represent hardware/software components, their interconnections and settings. A challenge these aspects propose is keeping them in sync with the device itself. In the proposed version of Scari, we only synchronize the models through adaptations. This can lead to short time periods where the world model does not reflect the true state of the device. For instance, in the experiment in Subsection 6.6.4 the PLC software of the Sink PLC itself resolves the problem of a faulty data point. This makes sense because it is the fastest way of dealing with the situation. We utilize a special syndrome processor that recommends updating the world model. From the recommendation to the adaptation it takes 285,6 milliseconds plus the waiting time of the recommendation decision maker. Within this time the world model is practically not consistent with the device. We think this is acceptable because of two reasons. The first reason is that no other adaptation can be carried out while the model is updated. The recommendation of updating a model would need a higher priority than others. The second reason is that Scari adaptations do not happen very often. Scari targets to extend the availability of devices but does not directly control hydropower turbines.

An alternative to utilizing an architectural model of the target device would be querying the responsible software components themselves. For instance, when a syndrome processor receives a notification from a monitor, it could ask the PLC software and other software components what their states are. The problem is that one could not be sure whether this reflects a consistent state. The architectural models are linked to a world version. If a model changes, this is subsumed by the new world version, which invalidates older architectural models.

6.8.3. Syndrome Processors

Syndrome processors are the entities that drive the Scari loop. They reason about events, recommend adaptations and utilize the world model for configuring themselves. A syndrome processor does not necessarily need to rely on notifications sent from monitors.

Essentially, the difference between a monitor and a syndrome processor is that a monitor observes a change and notifies other entities about it while a syndrome processor reasons about a situation and recommends an adaptation. If there is no need to separate the observation and reasoning of a situation then a syndrome processor could fulfill both roles in order to become faster.

A drawback is that syndrome processors need to track their recommendations. This is because another recommendation could be selected, a plan cannot be created, the action handler could be locked by the higher layer, or the recommendation refers to an outdated world version. In all these cases, a syndrome processor may reevaluate a situation.

Another drawback is that syndrome processors can only cope with situations and recommend adaptations anticipated at design time. For instance, a security attack, which has not been foreseen at design time cannot be recognized during run time. Also the available plan types are fixed during run time.

6.8.4. Adaptations

Actions are supposed to be atomic and used by plans in a sequential order. It may happen that an action fails to perform its adaptation. In that case, the action handler daemon stops the execution of the plan, leaves the system locked and distributes a special notification to the local unit and higher layer. With an engineering effort, this notification could be received by a special entity that triggers an emergency mechanism.

Alternatively, a roll back mechanism could be implemented that executes actions in reverse. However, this would just shift the problem as the corresponding roll back for an action could also fail.

As mentioned above, we execute actions in a sequential order. This is also true for plans as they cannot interfere each other. We do not consider it useful that a currently executed plan can become preempted by another plan. It would be complicated for a plan creator to create actions that preempt a running plan at the right point in time and do not cause some inconsistencies.

6.8.5. Configuration of Scari

In our prototype implementation, Scari is configured by initializing the *scariworldd* daemon with preconfigured models. At startup, if *scariworldd* does not find an object of type Me in its repository, it incorporates the preconfigured models provided by a configuration file. The monitors and syndrome processors are started by hand but obtain their configuration from the corresponding world model. With an engineering effort, monitors and syndrome processors could be started by a daemon observing the world model. We envision that the world model is preconfigured with an object of type Me and a device specific model during the production of control devices. The monitors and syndrome processors could then be added during the initial commissioning of devices.

6.8.6. Waiting Time

We omitted in our work guidelines of what is a good waiting time for the recommendation decision maker. This is because it depends on the syndrome processors that are competing around events. The syndrome processors in our experiments needed between 0.7 and 2.6 milliseconds for recommending a plan. This is because they are simple state machines that react on events. They precalculate the suitable recommendations based on the current world model. Potentially, such a syndrome processor could also be a costly algorithm that dynamically needs to check the current world model and data processed in the past. In our scenarios a small waiting time of several milliseconds would be enough.

A viable alternative to the waiting time of the recommendation decision maker could be the possibility to cancel the creation of a plan if a recommendation with a higher priority is received. In that case, if there is currently no plan created, the recommendation decision maker would process the first recommendation received. The priority of each subsequent recommendation would then be compared to the one currently processed. In our experiments, we measured execution times from the selected recommendation to the distributed selected plan ranging from 23,1 to 289 milliseconds. With an engineering effort, a plan can be canceled until it is distributed. This mechanism could replace the waiting time of the recommendation decision maker. The syndrome processor with the canceled plan could recommend the same plan later if the target fault still exists.

6.8.7. Real-Time

Detecting threats and adapting a system in real-time was not a requirement of Scari. In our opinion, several things would be needed for achieving a real-time behavior from the recommendation to the adaptation. D-Bus would need to be replaced with a message bus that offers hard real-time guarantees. In our experiments, we measured that a delay between the sender and receiver of a message sent over D-Bus ranges from less than one millisecond to five milliseconds. According to the D-Bus FAQ [7], a D-Bus one-to-one communication is 2.5 times slower than simply pushing data raw over a UNIX socket.

A real-time operating system can help substantially towards a predictable communication timing. Possible plans would need to be precalculated by the plan maker for each possible recommendation. With an engineering effort, the syndrome processors could register their possible recommendations at the plan maker. Further, the timing of world model changes, network latency and single actions need to be measured beforehand in order to precalculate the total execution time of plans.

If all these changes are applied and the software is extensively measured, one could guarantee distinct real-time deadlines for single recommendations with the corresponding adaptations.

[7] https://dbus.freedesktop.org/doc/dbus-faq.html

6.8.8. Human Intervention

It is crucial that engineers do not interfere with Scari while it is active. This could lead to an unsynchronized world model or counteracting adaptations. With an engineering effort, Scari could be paused while an architectural change is carried out. Such a mechanism should also include an update of the world model. Having a special state for a system during configuration is not uncommon. The commercial PLC we dealt with also has a special state named "load" which enables to configurate tasks. Correspondingly, Scari could be paused if the "regular" state of the PLC changes to "load".

6.8.9. Security of Scari

In our prototype implementation we trust all notifications, recommendations, plans, and actions sent over D-Bus and WebSockets. This is not secure. With an engineering effort, this behavior could be changed. By default, D-Bus authenticates the user under which an application is executed and not the application itself. A distinct user cannot connect to the session bus of another user. This can be improved by utilizing SELinux to assure that the correct applications access the message bus. As a downside, this does not prevent malicious applications running in the same security context. Concerning WebSockets, our implementation can be improved by utilizing Transport Layer Security (TLS) with company specific certificates. The same is true for the synchronization of git repositories.

However, even if the layers of Scari and the surrounding mechanisms, are properly secured and not maliciously modifiable, our approach would not be invulnerable against security attacks. For instance, an attacker, with enough knowledge and access to parts of a plant, could lead a Scari instance to believe that a fault occured and trigger the recommendation of a plan type. It is the responsibility of the plan creators to ensure that a plan does not damage the system. Under the assumption that plan creators are implemented properly, an attacker could therefore only change the structure of a system or trigger a fail-safe mechanism. In the worst case, this would stop the production of electricity but would not cause damage to the physical process.

6.8.10. Git

We rely on the version control sytem git for tracking changes and synchronizing knowledge bases between layers. This fits our requirements surprisingly well. In our experiments, one can observe that git repositories can grow up to three times the size of the round before. This is not as bad as it sounds because of two reasons. First, Scari only commits changes to the git repository if something is adapted. This should rarely happen because adaptations should not be carried out very often. Second, the software *git* provides a garbage collection routine[8], which packages objects and compresses them. By default,

[8] https://git-scm.com/docs/git-gc

changed model files are stored by git in new objects and we implement the same behavior with libgit2. By applying this routine on the repositories of the experiments, we could achieve up to four times smaller repository sizes.

What our combination of storing serialized JSON files on the filesystem and managing a git repository does not provide is some kind of query language such as SQL. In our experiments, we did not miss this feature because syndrome processors, plan creators and actions are directly operating on the models and know their specific structure. Where a query language could be useful, is on a higher layer which oversees several Scari instances. A plan creator could then use a simple query for finding viable data points. In our implementation, such a plan creator has to retrieve every child of type Me, check whether it points to a viable device and revise the provided data points. This would be easier to implement with a database.

6.9. Design Patterns

In distributed systems, we encountered three design patterns that try to grasp the trade-off between distributing data and information. Table 6.11 gives an overview of these patterns together with their known uses. A detailed explanation extended with a self-driving vehicle scenario in order to demonstrate the patterns can be found in **Paper D** [113].

Additionally, Table 6.11 lists the pattern *SEPARATION OF PROCESSING AND COORDINATION*, which is presented in **Paper H** [114]. It provides an architectural solution which shows how processing subsystems can be observed and adjusted by coordination subsystems. For instance in an industrial control system, a controller (processing) needs to control a physical process (resource) while also being observable and adjustable by a supervisory computer (coordination). The primary purpose of such a controller is to control the physical process within a defined time span. Additionally, the controller needs to distribute data to other devices or receive adjustments from higher ranking supervisory computers. All of these coordination tasks need to be done while supporting the physical process in real time. The supervisory computer may not be bound to specific real time requirements. This situation roughly describes the problem the presented pattern solves with a solution consisting of three parts.

Pattern	Description	Known Uses
LOCAL DATA PROCESSING	Entities, part of a distributed system, need to provide services. For doing that, they have to exchange states about the environment or themselves. The presented solution is that each entity is responsible for analyzing its sensed raw data to create higher level information. The entities exchange higher level information instead of raw data streams.	• SCADA systems. • Vehicle-to-X technologies. • Wireless sensor networks.
CENTRAL DATA PROCESSING	A service provided by an entity needs raw data sensed by other entities. Collecting higher level information from others would hide patterns and dependencies which would lead to a worse service. The solution is to gather raw data at one central entity responsible for the intended service and utilize powerful analysis methods (e.g. machine learning) for calculating appropriate actions. Other entities in the network are only responsible for sensing the environment, transmitting raw data to the accumulating entity and executing actions send from that entity.	• Internet of Things together with Cloud Computing. • The pattern AGGREGATING DEVICE GATEWAY in the domain of Internet of Things. • Master Terminal Units found in SCADA systems. • The pattern LOG AGGREGATION in the domain of cloud computing.
MIXED DATA PROCESSING	An entity needs to provide a service even if network connections fail. The intended service can be improved if raw data and information from other entities is available. The solution is that the entity realizes an analyzing mechanism in order to provide the service in a degraded form with the stored or currently sensed data and refined information. If a network is available, the entity may collect information or raw data from others. Another possibility would be that the entity connects to an other realizing the CENTRAL DATA PROCESSING pattern as soon as a network is available.	• Navigation applications • The pattern LOCAL PROCESSING GATEWAY in the domain of Internet of Things. • Edge-centric computing
SEPARATION OF PROCESSING AND COORDINATION	Systems are built for a purpose. The purpose transacted is usually handled by the processing part of a system and is observed and adjusted by coordination parts. In principle, these two kinds of system parts share the same target resource; the thing that is controlled by processing and indirectly by coordination subsystems. This leads to mutual influences, which can result in timing and priorities violations as well as performance degradations. This pattern provides an architectural solution which shows how processing subsystems can be observed and adjusted by coordination subsystems.	• Linux Xenomai • Self-Adaptive Software Systems (Scari) • Bolt processor interconnect

Table 6.11.: Three design patterns grasping the trade-off between distributing data and information. The fourth design patterns is about separating processing and coordination in computer systems.

7. Conclusion and Future Work

As outlined in the introduction, our work confirms the following hypothesis: *System knowledge enables automated resilience in industrial control systems.* We verify this hypothesis by exploring the possibilities of utilizing system knowledge at design and especially at run time for adding resilience.

Concerning design time we contribute modeling languages for describing the system knowledge (**C1**). Particularly, we propose metamodeling fragments for describing contracts and state machines in order to capture non-functional behavior of components. Supporting this, we propose a configurable constraint language, which can be leveraged by contract type specifications. The contract types limit the design space of a contract and therefore ease the transformation to verification tools. We show that our modeling languages can be utilized for verification, search of combinations and as input for run time. Based on our design-time work we derived design patterns for designing configurability into domain-specific modeling languages (**C4**).

We believe that contract-based design can play a key role in future systems for capturing non-functional properties. We envision that the construction of contracts should be done automatically during development. We think that manually capturing non-functional contracts, such as the timing of a software component on a certain operating system and processor, is too tedious for an engineer. In order to realize automatic mechanisms, simulation environments would be needed where an engineer deploys, for instance, a software component, and the system measures the behavior. However, certain non-functional contracts, such as contracts expecting hardware/software components to be signed with specific certificates, would probably still be manually configured. The contract state machine that we propose can play an important role in the composition of systems out of contracts. If at run time a certain state in a system changes the contract state machine can enable new contracts and invalidate or override old contracts.

Regarding the presented metamodel fragments for contracts, we do not present composition, refinement, and conjunction of contracts as described theoretical by Benveniste et al. [23]. This may be a worthwile gap for future directions of the modeling language. Furthermore, the presented metamodel could be ported to UML as a thin generic UML profile. This profile could be aligned with the existing OMG specifications MARTE [58] and SysML [55]. As mentioned by Selić and Gérard [115], a natural complementarity exists between MARTE and SysML. We have the view that a UML profile for contract-based design would benefit from concepts such as the physical types of MARTE or the constraint blocks of SysML. Not using such existing and standardized modeling concepts would be

similar to reinventing the wheel. The advantages of such a UML profile for contracts could be manifold. The most important one is the fact that it would allow the rise of specialized analyzing tools of different vendors which target single non-functional properties. The input of such tools would depend, in such an ideal ecosystem, on the same UML profile for contract-based design.

Concerning run time, we identified the potential of a self-adaptive software system for tackling hardware faults, security attacks, software bugs, misconfiguration, and faults in the physical environment, in our industrial setting (**C2**). To the best of our knowledge, we are the first ones identifying the potential of such a system in order to defend the hardware/software stack of industrial control systems. Based on this identified potential, we contribute a self-adaptive software system (Scari) that combines MAPE-K with the OODA loop (**C3**). We organize instances of this loop decentralized and hierarchical with own knowledge bases. This architecture allows us to execute adaptations on distinct devices in parallel. Based on these concepts we implemented a prototype implementation. We conducted realistic experiments in four scenarios. In order to evaluate our approach, we compared it to our requirements and in detail to state-of-the-art self-adaptive software systems. We discussed the issues and limitations of our approach and the technical implementation. This discussion can provide helpful hints and wisdom for other researchers. Based on our work at run time, we identified three design patterns grasping the trade-off between distributing data and information at run time. Furthermore, we found a design pattern for separating processing and coordination in computer systems (**C4**).

Scari is an ambitious attempt of taming harmful events in future industrial control systems. It is by no means a silver bullet and can only be part of a holistic approach that includes a plethora of development processes, verification methods, configuration methods, simulation environments etc.

The next step of the technical implementation of Scari would be to make it real-time capable for guaranteeing adaptation times. This would make it applicable for safety use cases. Furthermore, timing information could also be incorporated in the decision steps of the Scari loop. From the security perspective, the access to the self-adaptive system itself needs to be restricted.

A fundamental issue of self-adaptive software systems is that their monitors can be tricked by attackers that control the environment. At least, the planning part of Scari ensures that an adaptation is possible and tries to prevent harm.

With our testbed, we went into the direction of simulating "good" Scari configurations. Here, on could extend this testbed for simulating hydropower settings more realistic. From a scientific point of view, one could built a generic testbed, out of the proposed one, for comparing future decentralized self-adaptive software systems.

Regarding the configuration and development of monitors, syndrome processors and plans, it would be worthwhile goal to derive them from analysis methods such as Hazard Analysis and Risk Assessment (HARA), Failure Mode and Effects Analysis (FMEA), and STRIDE (Security).

Concerning the hierarchical structure of Scari, different structures could be explored. For instance, the decide, act, and knowledge parts of the loop could be centralized on higher layers.

Scari targets the properties self-configuring, self-healing and self-protecting. The property self-optimization is out of scope of this thesis and it would involve an understanding of the control logic. In our opinion, a centralized approach would be better suited for targeting self-optimization because it needs the complete knowledge about a hydropower unit.

The potential of self-adaptive systems raised in this thesis is a worthwhile goal to achieve. The presented self-adaptive software system named Scari could serve as reference architecture and starting point for future resilient system. We are confident that other domains, such as IoT and edge computing, can learn from this approach and that many aspects are reusable.

The proposed combination of design time modeling and a self-adaptive system at run time morphs the implicit system knowledge about an industrial control system into explicit "living" domain-specific knowledge contained in a structured, machine-processable way. This may enable new possibilities for designing reliable and secure industrial control systems in the future.

8. Publications

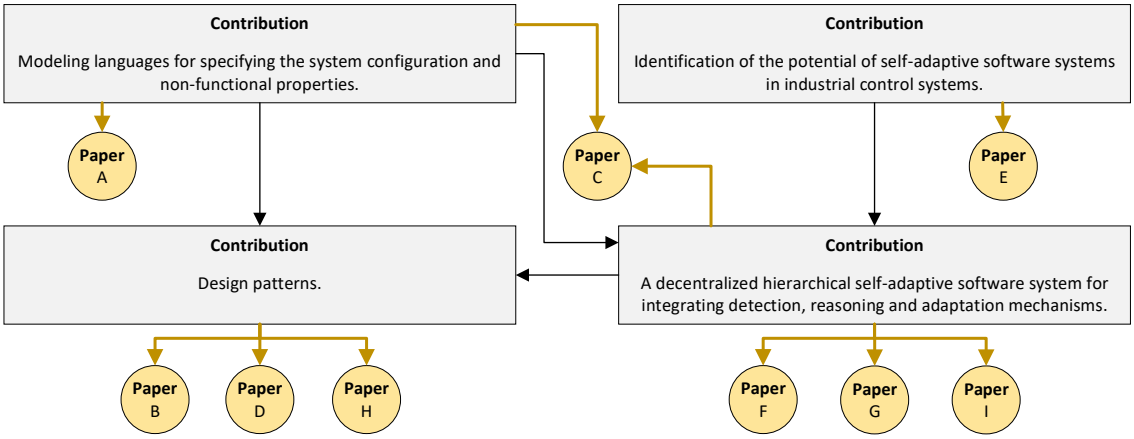


Figure 8.1.: Overview of contributions and related papers.

This thesis has been presented in parts in peer-reviewed workshop papers, conference papers and a book chapter (ordered by publication date). Figure 8.1 shows how the papers are related to our four contributions. In **Paper A** we propose pragmatic modeling concepts that are supposed to pave the way for integrating contract-based design into component models of systems. **Paper B** contains four design patterns for designing configurability into domain-specific language elements. **Paper C** proposes our approach of describing and verifying an industrial system at design time and utilizing this information at run time for our self-adaptive software system Scari. **Paper D** discusses three design patterns that try to grasp the trade-off between distributing data and information in distributed systems. **Paper E** shows in detail the potential of a self-adaptive system in our industrial setting. **Paper F** provides an overview of Scari and discusses design considerations and future research challenges. **Paper G** applies Scari in order to detect and repair permanent memory faults. **Paper H** presents a pattern for separating processing and coordination in computer systems.

A. J Iber, A Höller, T Rauter, and C Kreiner. "Towards a Generic Modeling Language for Contract-Based Design." In: *Proceedings of the 2nd International Workshop on Model-Driven Engineering for Component-Based Software Systems co-located with*

ACM/IEEE 18th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2015). ModComp@MoDELS '15. CEUR-WS.org, 2015

B. J Iber, A Höller, T Rauter, and C Kreiner. "Patterns for Designing Configurability into Domain-Specific Language Elements." In: *Proceedings of the 21st European Conference on Pattern Languages of Programs*. EuroPlop '16. ACM, 2016. ISBN: 978-1-4503-4074-8. DOI: 10.1145/3011784.3011785

C. J Iber, T Rauter, M Krisper, and C Kreiner. "An Integrated Approach for Resilience in Industrial Control Systems." In: *Proceedings of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops*. DSN-W '17. IEEE, 2017. DOI: 10.1109/DSN-W.2017.23

D. J Iber, T Rauter, M Krisper, and C Kreiner. "Patterns Grasping the Trade-off Between Distributing Data and Information." In: *Proceedings of the 22nd European Conference on Pattern Languages of Programs*. EuroPLoP '17. ACM, 2017. ISBN: 978-1-4503-4848-5. DOI: 10.1145/3147704.3147724

E. J Iber, T Rauter, M Krisper, and C Kreiner. "The Potential of Self-Adaptive Software Systems in Industrial Control Systems." In: *Proceedings of the 24th European Conference on Software Process Improvement*. EuroAsiaSPI '17. Springer International Publishing, 2017. ISBN: 978-3-319-64218-5

F. J Iber, T Rauter, and C Kreiner. "A Self-Adaptive Software System for Increasing the Reliability and Security of Cyber-Physical Systems." In: *Solutions for Cyber-Physical Systems Ubiquity*. IGI Global, 2018. DOI: 10.4018/978-1-5225-2845-6.ch009

G. J Iber, M Krisper, J Dobaj, and C Kreiner. "Dynamic Adaption to Permanent Memory Faults in Industrial Control Systems." In: *Proceedings of the 9th International Conference on Ambient Systems, Networks and Technologies*. ANT '18. Elsevier, 2018. DOI: 10.1016/j.procs.2018.04.058

H. J Iber, M Krisper, J Dobaj, and C Kreiner. "Separation of processing and coordination in computer systems." In: *Proceedings of the 23rd European Conference on Pattern Languages of Programs*. EuroPLoP '18. ACM, 2018. ISBN: 978-1-4503-6387-7/18/07. DOI: 10.1145/3282308.3282322

Additionally, this thesis is supplemented by the following peer-reviewed poster paper which was anticipating Scari:

I. A Höller, J Iber, T Rauter, and C Kreiner. "Poster: Towards a Secure, Resilient, and Distributed Infrastructure for Hydropower Plant Unit Control." In: *Proceedings of the 2016 International Conference on Embedded Wireless Systems and Networks*. EWSN '16. Graz, Austria: Junction Publishing, 2016. ISBN: 978-0-9949886-0-7

Peer-reviewed papers authored by the author of this thesis and presented at international conferences that are not included in this thesis (only principal authorship papers):

1. J Iber, N Kajtazović, A Höller, T Rauter, and C Kreiner. "Ubtl - UML Testing Profile based Testing Language." In: *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development*. Modelsward '15. SciTePress, 2015. ISBN: 978-989-758-083-3. DOI: 10.5220/0005241300990110

2. J Iber, N Kajtazović, G Macher, A Höller, T Rauter, and C Kreiner. "A Textual Domain-Specific Language Based on the UML Testing Profile." In: *Communications in Computer and Information Science*. Vol. 580. Springer International Publishing, 2015. ISBN: 9783319278681. DOI: 10.1007/978-3-319-27869-8_9

Towards a Generic Modeling Language for Contract-Based Design

Johannes Iber, Andrea Höller, Tobias Rauter, and Christian Kreiner
Institute for Technical Informatics
Graz University of Technology
Inffeldgasse 16, Graz, Austria
{johannes.iber, andrea.hoeller, tobias.rauter, christian.kreiner}@tugraz.at

***Abstract*—Component-based and model-driven engineering are key paradigms for handling the ever-increasing complexity of technical systems. Surprisingly few component models consider extra-functional properties as first class entities.**

Contract-based design is a promising paradigm, which has the potential to fill this shortage of methods for dealing with extra-functional properties. By defining the concept of using assumptions in order to determine the environment, and by using the concept of guarantees to state what a component provides to the environment, it enables the analyzability of components and compositions in advance and during system execution.

With this work, we aim to create the base for a pragmatic model-driven method that provides reusable modeling concepts for contracts targeting arbitrary extra-functional properties. Furthermore, we expand the current state-of-the-art of contract-based design by introducing the concept of a finite state machine, where single states consist of several valid contracts. It is also assumed that these modeling language features will ease the use of contract-based design. Additionally, we demonstrate the applicability of the presented modeling concepts on an exemplary use case from the automotive domain.

***Index Terms*—Metamodeling, contract-based design, extra-functional properties, component models**

I. Introduction

Numerous industrial sectors are currently confronted with massive difficulties originating from managing the increasing complexity of systems. The automotive industry, for instance, has an annual increase rate of software-implemented functions of about 30% [1]. This rate is even higher for avionics systems [2]. Additionally, this development of systems is not restricted to software, as we are facing a so-called Internet of Things, where the number of physical devices is expected to expansively explode [3]. New challenges regarding complexity of systems emerge caused by this dramatic increase of diverse hardware/software, possible interactions and distributed intelligences [4].

Component-based engineering is today a widely recognized and well-established paradigm for tackling complexity of systems [5]. Together with model-driven engineering, it forms a potentially powerful union to construct, analyze, and deploy systems.

But still, modern component models are flawed. As shown by Crnković et al. [5], astonishingly few (software) component models are addressing extra-functional properties (e.g. timing, safety, memory consumption, etc.) as first class entities. However, these properties are essential for composing a component-based system predictable and safe. Management of extra-functional properties is thus still one of the core challenges faced by component-based design [6].

Contract-based design is a promising paradigm for filling or narrowing this gap, [7]. It captures the behavior of a specific functional or extra-functional property in relationship with the environment of a component. Despite the existence of a mathematical groundwork [7] [8] and exemplary applications, a standard and generic metamodel for contract-based design does not yet exist.

With this work, we provide pragmatic modeling concepts that pave the way for integrating contract-based design into component models of systems. We present a metamodel fragment for contracts which target arbitrary single extra-functional properties. Furthermore, we introduce the concept of a finite state machine, where single states constitute valid contracts. This concept extends the current state-of-the-art regarding contract-based design. We show the applicability of these modeling concepts by using an example from the automotive domain. The target component of the use case is a simplified electronic steering column lock, which we examine with respect to the extra-functional properties safety and timing.

The remainder of this paper is structured as follows: the next Section provides a brief overview of the background to this work. In Section III the proposed modeling concepts are introduced. Subsequently, a use case demonstrating the applicability of these concepts is described in Section IV. Finally, concluding remarks and future research opportunities are given in Section V.

II. Background and Related Work

Here, we give an overview of system abstractions and properties. After this, we briefly explain contract-based design. Finally, we summarize the related work concerning contract-based design, which is also the motivation setting for this work.

A. System Abstractions and Properties

According to Jantsch [9], there are four main different abstraction models or views concerning embedded system engineering. First is the *computational model*, which describes the observable behavior of a system or of its single parts

(hardware, software components), i.e. the relationship between inputs and outputs [10]. Second, a *data model* exists that provides notations for information (e.g. integer, boolean). Third, a *time model* is needed to constitute the causality of events. Fourth, a *communication model* is established to specify how components interact. This model forms the top-level system behavior.

In the context of the properties of systems the literature distinguishes between functional and extra-functional (also known as non-functional) properties. Functional properties describe the function of a system or component, i.e. behavior, input or output data types. Extra-functional properties provide additional information and give a better insight into the behavior and capability of a system or component [6]. A wide range of such properties exists, e.g. safety, security, portability, performance. Since these issue from humans, there is no method to determine a priori which extra-functional properties exist in a system [6] [11].

B. Contract-based Design

Contract-based design usually sees a component as an abstraction, a hierarchical entity that represents a single unit of design [8] [12]. Therefore in the context of contract-based design a component can represent, for instance a module, a composition, a complex system or even a physical device.

The essence of this paradigm is to decompose a component into different independent views referred to as contracts, which capture the behavior of a target functional or extra-functional property under certain conditions [12] [13]. This approach significantly reduces the complexity of design and verification, because the single properties become manageable.

Informally, a contract is a set of assumptions and guarantees.

An assumption asserts what a contract expects from the component environment (this can include interactions with other components). Additionally, an assumption provides a certain context for the guarantees. The condition contained in an assumption can reference for instance input data, events or system properties. In general, the available variables are set or inferred by the analysis environment.

A guarantee describes what a component provides to the environment if the corresponding assumptions become valid. In the simplest case a guarantee states that a component just works under the constrained context. More complex contracts define limits for instance for output data, environment characteristics or extra-functional properties such as timing.

Historically, contract-based design is influenced by Meyer's design-by-contract principle [14] for object-oriented software [7]. The main difference is that contract-based design goes much further and provides means to integrate components in the design hierarchy [10]. This is achieved through capturing the context by assumptions (which may include platforms, other components, etc.), under which a component behaves as specified by the guarantees. Furthermore, a system can be viewed by selecting only appropriate contracts of interest.

Fig.1 illustrates that contract-based design not only allows the analyzing of components on a horizontal design level (e.g. interaction between software modules, hardware devices, etc.). It also enables analyzing to take place on a vertical level between different kinds of abstraction [7].

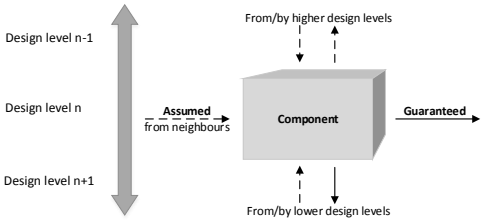


Fig. 1. Contract assumptions and guarantees for a component (Adapted from [15])

A solid mathematical groundwork already exists for this as provided by several authors, including Benveniste et al. [7], and Sangiovanni-Vincentelli et al. [8].

Promising applications of contract-based design have been shown for several domains. For instance, this paradigm has been demonstrated for smart integrated energy management systems [16], aircraft electric power systems [12], mixed-signal integrated circuits [17], and automotive [18] [7]. Despite these examples, contract-based design is still at its infancy [19].

Little work has been done towards establishing a generic standard metamodel for contract-based design. Warg et al. [20] presents a prototype modeling tool for contracts, but their work solely focuses on safety integrity levels.

C. Summary of Contract-Based Design

There exist a few approaches for realizing contract-based design, for instance the contract-based model developed in the framework of the SPEEDS project [13]. The problem is that state-of-the-art approaches either tackle single extra-functional properties, or take a relatively theoretical approach without concrete modeling examples or tool implementations. A survey concerning the certification of safety-relevant systems, carried out by the SafeCer consortium [21], shows that only a few companies are actually using contracts for components. And where this is the case they are relying on Meyer's design-by-contract principle on a programming language level.

III. Proposed Modeling Language Concepts

In this section, we explain concepts which are necessary for a pragmatic modeling language that targets contract-based design.

A. Target System Abstractions and Properties

With the following concepts, we aim at enriching the computational, time, and communication models of a system. Furthermore, the data model plays an important role, as it provides data types and notations, which could be used by contracts.

In the context of properties, our intention is to capture extra-functional properties and not necessarily functional behavior. We take the view that functional behavior is better described by other well-established methods than by the use of many different contracts.

The issue of what extra-functional properties we are aiming at, is dependents on the specific use case or context under which the following language features are used. These concepts may be applied for a wide range of different extra-functional properties (e.g. security, safety, timing, expected hardware/platform, memory consumption, many-core environment, etc.). But certainly not for all of them, since no silver bullet exists for dealing with every extra-functional property [11].

B. Pragmatic Modeling Langugage Features

In the following, we present a modeling concept for contracts. Additionally, we introduce the concept of a finite state machine for contracts.

1) Contract: Fig.2 illustrates our proposed metamodel for contracts. We separate a contract into two parts. A *Contract Declaration* represents a type for *Contract Definitions*. It states the available parameters, assumptions and guarantees. Furthermore, it represents the target extra-functional property. A *Contract Definition* captures the unique behavior concerning the target extra-functional property of a component in relationship to its environment.

Parameters can represent properties of the execution environment, data ports or events. They can be used by *Constraint Definitions* in order to set the specific assumption or guarantee. *Parameter Declarations* are used to specify that a variable of a specific data type may exist, but the concrete value has to be defined by the realizing *Contract Definition*. This can be useful for data arrays where the data points contained are individual for each component.

In the context of assumptions and guarantees, it is possible for a *Constraint Declaration* to set expected data types. The associated *Constraint Definition* must provide an expression where the resulting data type equals one of the expected types.

As we can see in Fig.2, we use the placeholders *Variable* for parameters, *DataType* for data types, and *Expression* for constraint expressions. These elements should be provided by a suitable constraint language or referable by the language that is used for the *Constraint Definition* expressions.

2) Finite State Machine for Contracts: Single contracts are sometimes not adequate for representing extra-functional properties. As we explain with our presentation in the following Section IV, cases exist where the behavior of a component - including extra-functional properties - changes over time or as a result of specific events. We thus expand the theory of contract-based design and capture such differences concerning contracts by applying the concept of a finite state machine. The idea is to have a finite state machine, where the single states may contain several currently valid contracts. The state machine itself operates on parameters provided by the environment or the internal states of a component.

Fig.3 illustrates our proposed metamodel for such a state machine. We again use the concept of declaration and definition in order to separate the specification and actual instance of a so-called contract state machine.

A *Contract State Machine Declaration* constitutes allowed *Contract Declarations*, concrete parameters and declarations of parameters which need to be defined by corresponding *Contract State Machine Definitions*.

Parameters are supposed to be used by *Contract State Machine Events* within constraint expressions, which trigger transitions to other *Contract State Machine States*. Such a state contains zero to infinite *Contract Definitions*.

Again, the metamodel elements *Variable*, *DataType* and *Expression*, refer to an arbitrary constraint language.

The actual semantics of a contract state machine depends on the target extra-functional properties and is determined by convention. It may be that entering a state implies that only those *Contract Definitions* it contains are valid. An alternative convention would be, that all visited *Contract Definitions* are valid except that a current *Contract Definition* overrides a former visited one by using the same *Contract Declaration*.

IV. Use Case

In this Section we show the application of our modeling concepts as presented on an exemplary use case from the automotive domain. First we give an overview of the target component and system. After that, we apply contracts together with a contract state machine. Finally, we discuss the use case presented.

A. Example - Electronic Steering Column Lock

Fig.4 illustrates a simplified electronic steering column lock (ESCL). Such locks are mandatory for cars in many countries. The Electronic Control Unit (ECU) decides whether to lock the steering column based on the input signals *Key State* and *Velocity*. These signals may be transmitted by a CAN bus or separate connections. If the ECU decides to lock the steering column, an actuator is activated which inserts the bolt into the steering column. Otherwise, the ECU decides to hold or eject the bolt.

There are several extra-functional properties which are worth considering in a system of this kind. In the following, we apply the modeling concepts presented for the extra-functional properties safety and timing. In the safety context we capture the data on whether the component ESCL is performing *normally*, is in a *failure* state, or *recovering* from a failure state. A failure state can be induced for instance by faulty transmitted data or other misbehaving components. Further to this we capture the data on how long it takes to execute the lock or unlock mechanism in two separate contract definitions.

B. Declarations

According to our metamodel concepts, the first step is to specify general declarations for components. Such declarations are known to contract checkers, interpreters or model transformers in advance. Fig.5 illustrates declarations for a

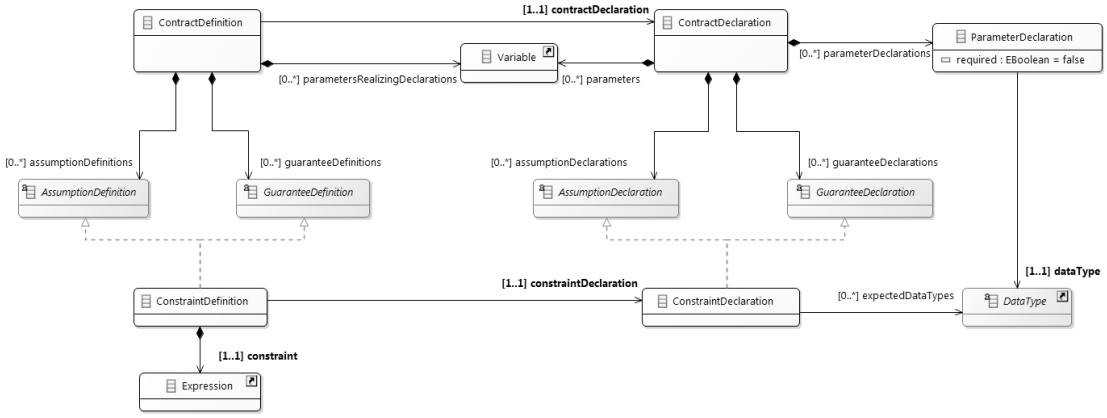


Fig. 2. Proposed Metamodel for Contract Declarations and Definitions

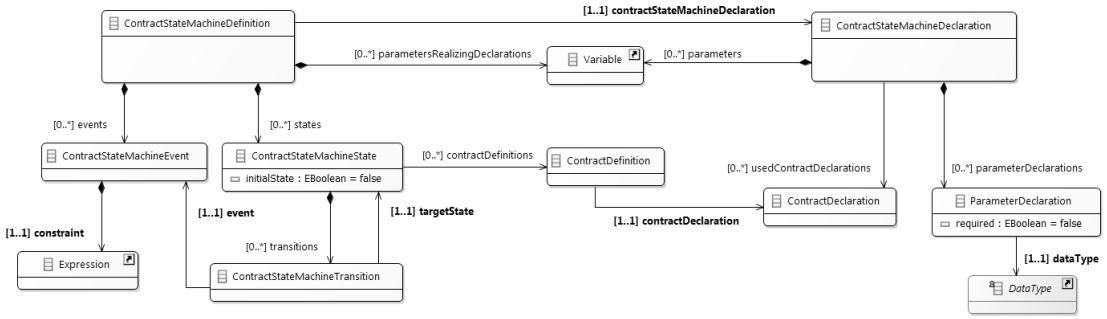


Fig. 3. Proposed Metamodel for Contract State Machine Declarations and Definitions

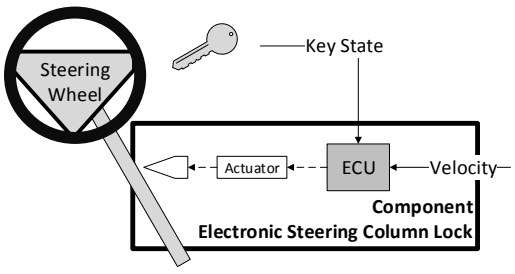


Fig. 4. Example Component - Electronic Steering Column Lock

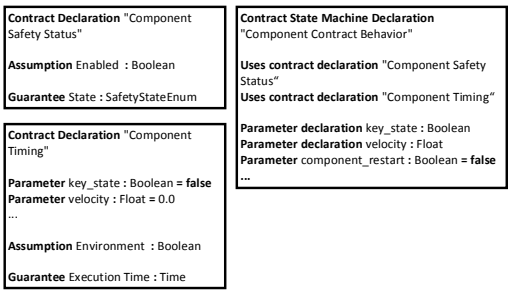


Fig. 5. Use Case Declarations for the target extra-functional properties

component's safety status and timing. Additionally, we specify a contract state machine declaration that is used to capture the behavior of a component in order to set valid contracts.

The contract declaration *Component Safety Status* assumes whether the component of interest is enabled and guarantees a certain safety state to the environment. The available types for this guarantee are restricted by the data type *SafetyStateEnum*, which contains the literals *NORMAL*, *FAILURE*, and *RECOVER* (not shown in Figure 5).

The contract declaration *Component Timing* is used to

guarantee a specific execution time for certain assumed environments. The parameters *key_state* and *velocity* are provided by the analysis environment. The boolean parameter *key_state* indicates whether the ignition system is activated (boolean value true), while the parameter *velocity* states the current speed of the car. A comprehensive contract declaration would provide several other parameters, which may be obtained for instance by a CAN bus or observed from the condition of a system. The issue of which of these parameters are actually used by the assumption *Environment* depends on the component. When this assumption results in a boolean true, the guarantee *Execution Time* becomes valid.

Furthermore, contract definitions of these declarations can be used by the single states of the contract state machine *Component Contract Behavior*. Here again the parameters contained are obtained by the analysis environment or transmitted by the available connections. For instance, the parameter *component_restart* must be set by the analysis environment or by the described component. These parameters are used by a contract state machine definition in order to specify the events for state transitions.

C. Definitions

We now present how the declarations from above are used.

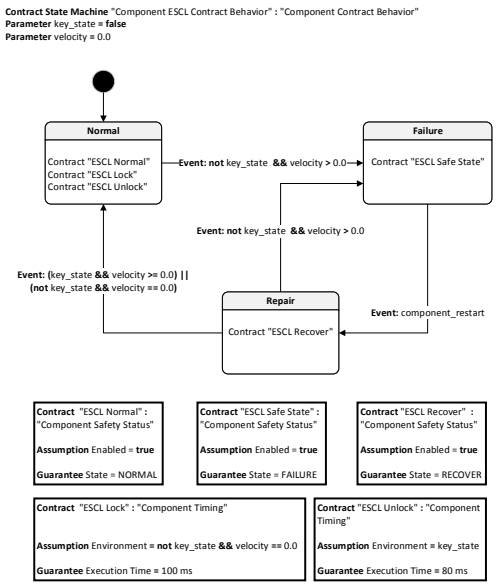


Fig. 6. Contract State Machine and Contract Definitions of the ESCL Example

Fig.6 illustrates a contract state machine definition which sets the valid contract definitions according to the current state. The parameters are realizations of the parameter declarations declared by the contract state machine declaration *Component Contract Behavior* and are initialized to default values.

The initial state of this example is state *Normal*. Within this state, we can guarantee the execution time in respect to the locking and releasing mechanism. Furthermore, the contract *ESCL Normal* determines the safety state *NORMAL* to the environment. Whenever an abnormal event occurs such as there is no key but the car is moving, the contract state machine changes to the state *Failure*. In this state we cannot constitute the execution time of the ECSL and the contract *ESCL Safe State* becomes valid. After the component ESCL restarts, the state machine changes to the state *Repair*, which is reflected by the contract *ESCL Recover*. When the recover procedure was successful, the state machine changes to the state *Normal*, where the contained contracts become valid again, otherwise the state machine switches back to state *Failure*.

D. Discussion of the Use Case

We have shown how our contract modeling features can be used as presented on a simplified use case. It is imaginable that this example can be further advanced to capture the target and other extra-functional properties in more detail.

Note that we do not capture the actual functional behavior of the component ESCL. We rather use the functional behavior of the environment in order to determine how the target extra-functional properties timing and safety status of the component are changing and what guarantees are valid in that state. The semantics of the contract state machine we present is such that a new state invalidates the former visited contracts. The assumptions and guarantees of the *Contract Definitions* must be either automatically gathered by a measurement software or issued by humans.

Such a contract state machine can be used for two purposes.

One purpose is that a system becomes analyzable in advance, also with respect to composability. A model checker could simulate such a system and calculate the different expected safety states. Another model checker would be able to estimate the overall timing of a system.

The second purpose would be that a detection mechanism observes and constitutes the single states during runtime of a system and takes appropriate action based on predetermined contracts.

V. Conclusion and Future Work

In this paper, we presented concepts for modeling contracts and showed in a use case how these concepts can be applied.

The vision is to have a generic modeling language for specifying contract types and contract instances. By using the term generic we mean contracts that are suitable for at least a substantial number of extra-functional properties.

We introduced the concept of splitting a contract into a declaration and a definition. For analysis purposes a specific contract declaration would be known by a model checker or code generator beforehand. It declares the available parameters, assumptions and guarantees, while a contract definition uses such a declaration to define the actual behavior of a target extra-functional property.

Furthermore, we introduced the concept of a contract state machine which is basically a finite state machine where the single states represent different contract definitions. This concept is necessary, because a component may behave in different ways depending on the input data, environment properties or specific events. For instance, the timing of a component may be different depending on its previous processed data. It may also be different if the environment has changed. Such changes may require different valid contracts.

Concerning our future work, we are currently working on a configurable constraint modeling language, inspired by OCL [22], which we want to use for setting assumptions and guarantees. The idea is to have a constraint language where language elements, such as an if expression or a boolean operation, can be disabled and is afterwards not usable by an assumption or guarantee. This is useful, in our opinion, to simplify the construction of contract checkers or interpreters, because not all concepts of an expression language need to be considered and handled properly. It would also provide a user with direct feedback concerning what language elements are allowed for use.

Additionally, the presented modeling features for contracts do not consider composition, refinement, and conjunction of contracts as described theoretical by Benveniste et al. [7]. We are still working on finding pragmatic and usable metamodel solutions for these concepts.

After building this in a form suited to our use case metamodel for contract-based design, we are planning to develop a thin generic UML profile [23] for contracts and contract state machines.

This profile will be aligned with the existing OMG specifications MARTE [24] and SysML [25]. As mentioned by Selić and Gérard [26], a natural complementarity exists between these two profiles. We are of the view that a UML profile for contract-based design would benefit from concepts such as the physical types of MARTE or the constraint blocks of SysML. Not using such existing and standardized modeling concepts would be like reinventing the wheel.

The advantages of such a UML profile for contracts could be manifold. The most important one is, that it would allow the rise of specialized analyzing tools of different vendors which target single extra-functional properties. The input of such tools would depend, in such an ideal ecosystem, on the same UML profile for contract-based design.

REFERENCES

[1] C. Ebert and C. Jones, "Embedded Software: Facts, Figures, and Future," *Computer*, vol. 42, no. 4, Apr. 2009.

[2] P. Feiler, J. Hansson, D. de Niz, and L. Wrage, "System Architecture Virtual Integration: An Industrial Case Study," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, CMU/SEI-2009-TR-017, 2009.

[3] M. Miller, *The Internet of Things: How Smart TVs, Smart Cars, Smart Homes, and Smart Cities Are Changing the World*. Pearson Education, 2015.

[4] D. Miorandi, S. Sicari, F. De Pellegrini, and I. Chlamtac, "Internet of things: Vision, applications and research challenges," *Ad Hoc Networks*, vol. 10, no. 7, Sep. 2012.

[5] I. Crnkovic, S. Sentilles, V. Aneta, and M. R. Chaudron, "A Classification Framework for Software Component Models," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, Sep. 2011.

[6] S. Sentilles, P. Štěpán, J. Carlson, and I. Crnković, "Integration of Extra-Functional Properties in Component Models," in *Component-Based Software Engineering*. Springer Berlin Heidelberg, 2009.

[7] A. Benveniste, B. Caillaud, D. Nickovic, R. Passerone, J.-B. Raclet, P. Reinkemeier, A. L. Sangiovanni-Vincentelli, W. Damm, T. Henzinger, and K. Larsen, "Contracts for Systems Design," INRIA, Rennes, France, Tech. Rep., 2012.

[8] A. Sangiovanni-Vincentelli, W. Damm, and R. Passerone, "Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems," *European Journal of Control*, vol. 18, no. 3, Jan. 2012.

[9] A. Jantsch, *Modeling Embedded Systems and SoCs: Concurrency and Time in Models of Computation.* San Francisco, Amsterdam: Morgan Kaufmann, 2004.

[10] N. Kajtazovic, "A Component-based Approach for Managing Changes in the Engineering of Safety-critical Embedded Systems," Ph.D. dissertation, Graz University of Technology, 2014.

[11] I. Crnkovic, M. Larsson, and O. Preiss, "Concerning Predictability in Dependable Component-Based Systems: Classification of Quality Attributes," in *Architecting Dependable Systems III*. Springer Berlin Heidelberg, 2005.

[12] P. Nuzzo, Huan Xu, N. Ozay, J. B. Finn, A. L. Sangiovanni-Vincentelli, R. M. Murray, A. Donze, and S. A. Seshia, "A Contract-Based Methodology for Aircraft Electric Power System Design," *IEEE Access*, vol. 2, 2014.

[13] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis, "Multiple Viewpoint Contract-Based Specification and Design," 2008.

[14] B. Meyer, "Applying 'design by contract'," *Computer*, vol. 25, no. 10, Oct. 1992.

[15] A. Rajan and T. Wahl, Eds., *CESAR - Cost-efficient Methods and Processes for Safety-relevant Embedded Systems*. Vienna: Springer Vienna, 2013.

[16] M. Maasoumy, P. Nuzzo, and A. Sangiovanni-Vincentelli, "Smart Buildings in the Smart Grid: Contract-Based Design of an Integrated Energy Management System," 2015.

[17] P. Nuzzo, A. Sangiovanni-Vincentelli, Xuening Sun, and A. Puggelli, "Methodology for the Design of Analog Integrated Interfaces Using Contracts," *IEEE Sensors Journal*, vol. 12, no. 12, Dec. 2012.

[18] N. Kajtazovic, C. Preschern, A. Höller, and C. Kreiner, "Constraint-Based Verification of Compositions in Safety-Critical Component-Based Systems," in *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, ser. Studies in Computational Intelligence. Springer International Publishing, 2015.

[19] P. Nuzzo and A. Sangiovanni-Vincentelli, "Lets Get Physical: Computer Science Meets Systems," in *From Programs to Systems. The Systems perspective in Computing*. Springer Berlin Heidelberg, 2014.

[20] F. Warg, B. Vedder, M. Skoglund, and A. Soderberg, "Safety ADD: A Tool for Safety-Contract Based Design," in *2014 IEEE International Symposium on Software Reliability Engineering Workshops*, Nov. 2014.

[21] O. Bridal, R. Mader, A. Geven, E. Schoitsch, H. Martin, M. Larramendi, A. Aristimuno, A. Fritsch, E. Vaumorin, M. Bordin, A. Solinas, A. Martelli, I. Korago, A. Levcenkovs, F. Joakim, R. Land, A. Söderberg, P. Conmy, and M. Illarramendi, "State-of-practice and state-of-the-art agreed over workgroup," Tech. Rep., 2011. [Online]. Available: http://www.safecer.eu/images/pdf/pSafeCer_D1.0.1StateOfThePracticeAndTheArt.pdf

[22] Object Management Group (OMG), "Object Constraint Language Version 2.4," 2014. [Online]. Available: http://www.omg.org/spec/OCL/2.4/

[23] ——, "Unified Modeling Language (UML)," 2015. [Online]. Available: http://www.omg.org/spec/UML/Current

[24] ——, "UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems Version 1.1," 2011. [Online]. Available: http://www.omg.org/spec/MARTE/

[25] ——, "OMG Systems Modeling Language (OMG SysML) Version 1.3," 2012. [Online]. Available: http://www.omg.org/spec/SysML/1.3/

[26] B. Selić and S. Gérard, *Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE*, 2014.

Patterns for Designing Configurability into Domain-Specific Language Elements

JOHANNES IBER, ANDREA HÖLLER, TOBIAS RAUTER and CHRISTIAN KREINER, Institute of Technical Informatics, Graz University of Technology

Nowadays, designing a domain-specific language is easier than ever before. Nevertheless, finding the right balance concerning the configurability of concepts represented by language elements is a complicated design task. With this paper we provide four patterns that discuss different kinds of configurability that can also be combined. In the end, we show the application of these patterns using an example we had to deal with in our research project.

CCS Concepts: •**Software and its engineering** → **Design patterns;**

Additional Key Words and Phrases: configurability, domain-specific language, language element

ACM Reference Format:

Johannes Iber, Andrea Höller, Tobias Rauter, and Christian Kreiner. 2016. Patterns for Designing Configurability into Domain-Specific Language Elements , 14 pages.
DOI: http://dx.doi.org/10.1145/3011784.3011785

1. INTRODUCTION

New domain-specific languages are designed by engineers and researchers all the time. Especially through the rise of so-called language workbenches this task becomes subsequently easier to accomplish. Language Workbenches are tools that help to build a domain-specific language [Fowler 2010]. We claim that with the help of these tools, the number of languages rises much higher in the future.

There exists a lot of literature on how to technically design a domain-specific language and corresponding infrastructure. But, as far we know, little has been mentioned about the inherent configuration nature concerning the representation of concepts through language elements. By using the term *concept* we mean a domain-specific thing that is represented inside a language. *Language elements* are for us the building blocks to configure a concept and to bring it to life with data. With this work, we aim to give language designers useful patterns in order to foresee the consequences of their language element design decisions. These patterns are not canceling each other out and can be combined in order to make a concept more flexible.

We use the term domain-specific language not only for textual languages based on grammars (e.g. build by using an Extended-Backus Naur Form grammar), but also for model-based languages (e.g. facilitating the metamodeling architecture defined by the Object Management Group), and internal languages (e.g. defined inside a host programming language like Java). The following patterns can be applied with all these techniques.

EuroPLoP '16, July 06 - 10, 2016, Kaufbeuren, Germany

ACM 978-1-4503-4074-8/16/07 $15.00
DOI: http://dx.doi.org/10.1145/3011784.3011785

EuroPLoP '16, Publication date: 2016.

The remainder of this work is structured as follows: the next Section gives an overview of this work. Section 3 describes the Atomic Concept pattern, which can be seen as the base pattern for the other ones. Section 4 describes the second pattern named EntityType. Section 5 outlines the Deep Configuration pattern. Section 6 presents the last pattern named Concept Tailoring. Section 7 gives an experience report and documents our journey of applying these patterns. Finally, Section 8 concludes this work.

2. OVERVIEW

The presented patterns are targeting to solve the general problem of how to embody a domain-specific concept in language elements that a language user is able to configure the concept to a certain degree. Therefore, they share the same problem and context. The explicit highlighting of these two parts is omitted inside the single patterns.

The first pattern named Atomic Concept represents the simplest one and serves as base for the others. It can be found in probably every domain-specific language and represents concepts where reusing data is not needed. The second pattern adds a layer that allows the reuse of information and enables the definition of variants of concepts. The third pattern named Deep Configuration reuses data from existing instantiations of a concept. The fourth pattern targets to unhinge information from the infrastructure around a language. It makes parts of it adjustable by a language user.

In all patterns, we use the term *Entity* for a language element that represents the referable instance of a language concept. *EntityType* is, for us, a kind of language element that defines the behavior and attributes of corresponding *Entities*. A *Declaration* element unhinges information from the language infrastructure and deeply influences *EntityType* and *Entity* elements. The element kinds *EntityType* and *Declaration* are usually partly or completely hidden for other language elements not part of the target language concept.

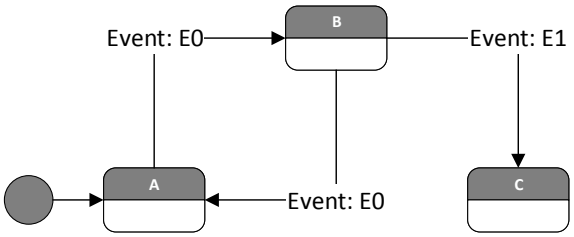


Fig. 1: A simple finite state machine that serves as an example for the following patterns.

The running example demonstrating the patterns is a simple finite state machine that consists of events, states and transitions. Figure 1 visualizes the exemplary state machine. The expressed behavior remains in all examples the same. Main differences between the examples are how a state machine is organized and, more importantly, what is configurable and reusable by language users. The examples itself are presented in two parts. An abstract part illustrates a high level description of the example, while the other part shows how the example could be defined in a host general purpose programming language like Java.

EuroPLoP '16, Publication date: 2016.

3. ATOMIC CONCEPT PATTERN

Forces

—Each *Entity* is similar to others.

—Different variants can, but not necessarily be expressed by contained attributes.

—Serves as building block for other language concepts.

Solution

Add one language element that represents the language concept. Add desired configurable attributes and references. The possible variants of this language element are defined by these attributes and references.

This pattern is used for *Entities* that realize a language concept where parts of the concept itself are not supposed to be reused. The configuration nature is not split between separate elements. This is the simplest kind of representing a concept within a language. Examples are control or data structures that just exist.

Consequences

+ Easy to implement.
+ Easy to validate.
+ Easy to understand for language users.
+- Limits concept variability.
- Static semantics.
- May not be appropriate for all kind of language concepts.
- Complicated to evolve if language environment changes. Existing artifacts may break.

Abstract Example

Figure 2a illustrates an application of the Atomic Concept pattern on a state machine. Each state defines possible transitions to others, triggered by the occurrence of certain events. This is a realization of this pattern, because there does not exist a separation between the referable entity state machine and a defining or configuring kind of elements. Figure 2b shows an alternative application of the Atomic Concept pattern on a state machine, where the behavior is defined as rules.

Java Example

Listing 1 shows an implementation of an Atomic Concept state machine similar to Figure 2b. The class *FSMEntity* (lines 7-27) represents the whole instantiation of the language concept state machine. It offers a method *addTransition* (lines 11-17) that allows adding usable transitions, while the method *move* (lines 19-26) triggers the finite state machine and determines a new state. The class *Transition*

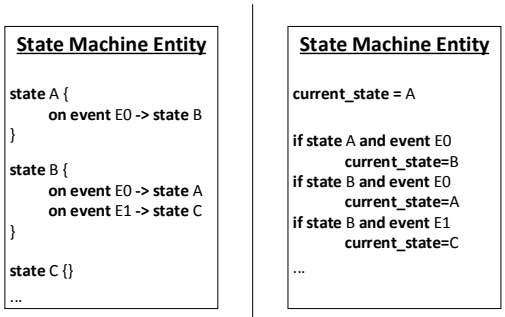


Fig. 2: Two different possible realizations of an Atomic Concept state machine.

(lines 1 - 5) is used for representing a transition from a current state to a target state based on an event.

Listing 1: Java realization of the second Atomic Concept example.

```
public class Transition {
 public String currentState = "";
 public String event = "";
 public String targetState = "";
}

public class FSMEntity {
 public String currentState = "";
 private Vector<Transition> transitionTable = new
      Vector<Transition>();

 public void addTransition(String currentState,
      String event, String targetState) {
  Transition t = new Transition();
  t.currentState = currentState;
  t.event = event;
  t.targetState = targetState;
  transitionTable.add(t);
 }

 public void move(String event) {
  for (Transition t : transitionTable) {
   if (t.currentState.equals(this.currentState) &&
        t.event.equals(event)) {
    this.currentState = t.targetState;
    break;
   }
  }
 }
}
```

Listing 2 demonstrates how the class *FSMEntity* is used. The states and events are defined inside transitions. The methode *move* triggers the state machine based on the event argument.

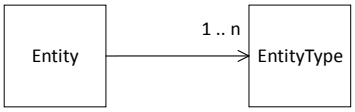
Listing 2: Application of the Java example

```
public static void main(String[] args) {
 FSMEntity entity = new FSMEntity();

 entity.addTransition("A", "E0", "B");
 entity.addTransition("B", "E0", "A");
 entity.addTransition("B", "E1", "C");

 entity.currentState = "A";

 entity.move("E0");
 // ...
}
```

Known Uses

—The Atomic Concept pattern is the simplest form of implementing a concept inside a language. It can be found in virtually every domain-specific language. Some languages like JSON [Crockford 2016] consist exclusive of concepts which are compositions of this pattern. Control structures, like a while loop, are usually realizations of this pattern. Another application is to use it for properties that capsule for instance a type, name and value.

4. ENTITYTYPE PATTERN



Forces

—*Entities* express the same concept but differ concerning their available properties.

—Different instantiations of a language concept are needed.

—Language infrastructure is probably fixed.

—A reuse of already defined data is needed.

Solution

Add an *Entity* language element and a *EntityType* element. These two different kinds of elements can only represent the target concept together. The *Entity* element must reference at least one *EntityType* element. The *EntityType* element can define properties for the *Entity* element. Further, it can specify reusable data. *Entities* can be differentiated by their referenced *EntityTypes*.

Consequences

+ *Entity* becomes flexible concerning the available properties.
+ The defining part of *Entities* is separate.
+ Appropriate for type-object relationships.
+ Concept known to users from common programming languages.
- A change of *EntityType* data affects all dependent *Entities*.
- The semantics of the language element *EntityType* is embedded into the language infrastructure.

Abstract Example

Figure 3 illustrates an exemplary application of this pattern on a state machine. Note, that a state machine is now split into two parts. The *State Machine Type* is used for specifying available states and events. The *State Machine Entity* describes the transitions and represents the usable state machine. This decoupling allows the specification of several state machines that share the same states and events. Further, *State Machine Entities* become distinguishable from each other based on the used *State Machine Type*.

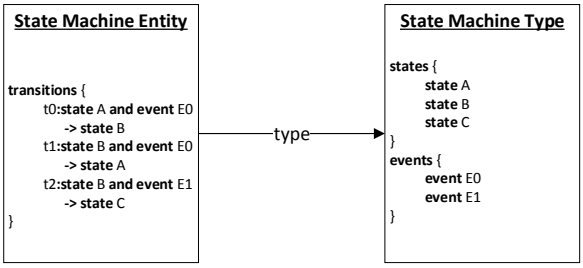


Fig. 3: EntityType example

Java Example

Listing 3 shows the EntityType state machine realized in Java code. The class *FSMType* (lines 1 - 4) is used for describing states and events, while the class *FSMEntity* (lines 10 - 30) holds the transition information. Further, the class *FSMEntity* owns a method *validateStateMachine* (lines 23 - 25) which ensures that transitions correspond to the states and events specified by the used *FSMType*. We skipped the implementation of this method as it can be realized in different ways.

Listing 3: Java realization of the EntityType example.

```
1  public class FSMType {
2   public Vector<String> states = new Vector<String>();
3   public Vector<String> events = new Vector<String>();
4  }
6  public class Transition {
7   // Equal to Atomic Concept
8  }
10 public class FSMEntity {
11  public FSMType type;
12  public String currentState = "";
13  private /* . */ transitionTable = // Equal to Atomic
        Concept
15  public FSMEntity(FSMType type) {
16   this.type = type;
17  }
19  public void addTransition(/* ... */) {
20   // Equal to Atomic Concept
21  }
23  public boolean validateStateMachine() {
24   // Implementation specific
25  }
27  public void move(String event) {
28   // Equal to Atomic Concept
29  }
30 }
```

Listing 4 illustrates the use of the classes *FSMType* and *FSMEntity*. What changed to the pattern above is, that defined states and events are encapsulated and such capsules are comparable and interchangeable. It is also imaginable that the *EntityType* and *Entity* parts are specified by different stakeholders. The validation of a state machine is necessary because otherwise invalid transitions would be specifiable.

Known Uses

—A prominent concept of UML that applies this pattern are the language elements *Class* and *InstanceSpecification* [Object Management Group (OMG) 2015]. A *Class* specifies features that characterize the structure and behavior of objects. An *InstanceSpecification* may describe an object with data

Listing 4: Application of the Java example

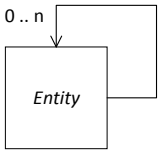
```
public static void main(String[] args) {
 FSMType def = new FSMType();
 def.states.add("A");
 def.states.add("B");
 def.states.add("C");
 def.events.add("E0");
 def.events.add("E1");

 FSMEntity entity = new FSMEntity(def);
 entity.addTransition("A", "E0", "B");
 entity.addTransition("B", "E0", "A");
 entity.addTransition("B", "E1", "C");
 entity.currentState = "A";

 if (entity.validateStateMachine()) { // Correct SM
  entity.move("E0");
  // ...
 }
}
```

that correspond to the features of the language element *Class*. Therefore, the *Class* element behaves like a *EntityType* and defines reusable properties, while *InstanceSpecification* is an Entity which is referred to for instance by Interaction Diagrams.

—Many component-based modeling languages apply this pattern by using interfaces as *EntityTypes* while the concrete components are *Entities* [Crnkovic et al. 2011].

—In literature this pattern is also known as *type-object* [Johnson and Woolf 1998] and *item-descriptor* [Coad 1992] pattern.

5. DEEP CONFIGURATION PATTERN



Forces

—Reuse of already specified *Entities* is desired.

—The number of reused *Entities* varies.

Solution

Entities add, modify, or remove configuration information from other *Entities*. There can be an arbitrary long chain of them. Each *Entity* represents an instance of the language concept and may be complete.

Consequences

+ Extensible *Entities*.
+ Reuse of already specified information.
- Changes of *Entities* have consequences on dependent *Entities*.
- Very fragile.

Abstract Example

Figure 4 illustrates a state machine where states, events and transitions are specifiable inside *State Machine Entities*. The difference to the patterns above is, that an arbitrary number of *Entities* can be added on top of each other. Each of these state machines is fully functional. The advantage of this pattern is that language users can efficiently reuse or modify already defined information.

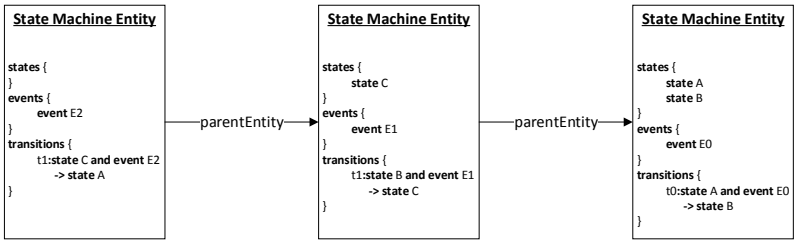


Fig. 4: Deep Configuration example

Java Example

Listing 5 represents a realization of the Deep Configuration pattern which allows defining a chain of state machines like it is depicted in Figure 4. The class *FSMEntity* (lines 5 - 40) provides an attribute *parentEntity* (line 6) that allows reusing an already defined *FSMEntity*. The getAll methods (lines 22 - 26) are used for collecting all states, events and transitions. The validateStateMachine (lines 28 -30) and move (lines 32 - 39) methods serve the same purpose like in the patterns above.

Listing 5: Java realization of the Deep Configuration example.

```
public class Transition {                                          1
 // Equal to Atomic Concept                                        2
}                                                                  3

public class FSMEntity {                                           5
 public FSMEntity parentEntity;                                    6
 public Vector<String> states = new Vector<String>();              7
 public Vector<String> events = new Vector<String>();              8
 public Vector<Transition> transitionTable = new                   9
      Vector<Transition>();
 public String currentState = "";                                  10

 public FSMEntity() {}                                             12

 public FSMEntity(FSMEntity parentEntity) {                        14
  this.parentEntity = parentEntity;                                15
 }                                                                 16

 public void addTransition(/* . */) {                              18
  // Equal to Atomic Concept                                       19
 }                                                                 20

 public Vector<String> getAllStates() {/* . */}                    22

 public Vector<String> getAllEvents() {/* . */}                    24

 public Vector<Transition> getAllTransitions() {/*.*/}             26

 public boolean validateStateMachine() {                           28
  // Implementation specific                                       29
 }                                                                 30

 public void move(String event) {                                  32
  Vector<Transition> allTransitions =                              33
       getAllTransitions();
  for (Transition t : allTransitions)                              34
   if (t.currentState.equals(this.currentState) &&                 35
        t.event.equals(event)) {
    this.currentState = t.targetState;                             36
    break;                                                         37
   }                                                               38
  }                                                                39
}                                                                  40
```

Listing 6 demonstrates how the chain of finite state machines of Figure 4 are specified with the Java class *FSMEntity* of Listing 5. The big downside of this pattern is that if a referenced entity is changed the consequences for dependent entities is not always simple to foresee.

Listing 6: Application of the Java example

```
public static void main(String[] args) {
 FSMEntity entity0 = new FSMEntity();
 entity0.states.add("A");
 entity0.states.add("B");
 entity0.events.add("E0");
 entity0.addTransition("A", "E0", "B");
 entity0.currentState = "A";

 if(entity0.validateStateMachine()) { // Correct SM
  entity0.move("E0");
 }

 FSMEntity entity1 = new FSMEntity(entity0);
 entity1.states.add("C");
 entity1.events.add("E1");
 entity1.addTransition("B", "E1", "C");
 entity1.currentState = "A";

 if(entity1.validateStateMachine()) { // Correct SM
  entity1.move("E0");
 }

 FSMEntity entity2 = new FSMEntity(entity1);
 entity2.events.add("E2");
 entity2.addTransition("C", "E2", "A");
 entity2.currentState = "B";

 if(entity2.validateStateMachine()) { // Correct SM
  entity2.move("E1");
 }
}
```

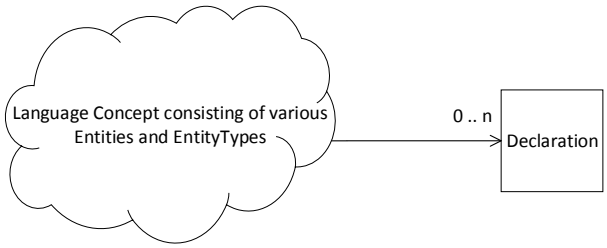
Known Uses

—The concept of inheritance of classes, known from programming and modeling languages like UML, is an application of this pattern. Derived classes reuse, add or modify information.

—The concept of a package is a realization of this pattern. A package contains data (e.g. classes) and can reuse data from other packages.

—UML offers the possibility to redefine a state machine [Object Management Group (OMG) 2015]. A specialized state machine can extend a general one. It can add or redefine states and transitions similar to the example of this pattern.

6. CONCEPT TAILORING PATTERN



Forces

—Domain-specific concept has to be applicable for slightly different cases inside a domain.

—Some parts of a concept need to be constrained. For instance, there could exist different target platforms with different support of features.

—Language infrastructure has to be dynamic and parts of it should be configurable by the language user.

Solution

In principle, there are two ways of applying this solution.

One is to add a *Declaration* element inside a language to the existing elements that represent a concept. Then it configures a context for *EntityType* and *Entity* elements and consists of various settings that adjust the possibilities of what can be done with a concept. At some point a language concept has to refer to a specific *Declaration* element.

The second way is to specify the *Declaration* outside of a domain-specific language with a different language, for instance with XML, and to load it by the tooling infrastructure around a language.

Basically, this pattern makes behavior explicit that would have been hard-coded into the language infrastructure. Further, it allows adjusting a domain-specific concept semantically. The settings of the *Declaration* can either restrict a language concept or enable features.

Consequences

+ Implicit configuration of language infrastructure becomes accessible and adjustable.
+ Allows adjusting domain concepts without breaking existing artifacts. Legacy artifacts simply use an already existing *Declaration*.
+ Supports fast but not disruptive changing domain concepts.
+ *Declaration* may be hidden for a language user, but accessible for the language infrastructure.
+ Validation happens more on language concepts and less on domain semantics.
- Complicated to understand for language users.
- Implementation effort higher than in the patterns above. The validation of an *Entity* or *EntityType* needs to take the optional *Declaration* element in account.
- Difficult do define the appropriate variability inside the language elements.
- No gain if the concept is fixed and behaves semantically always the same.
- Language infrastructure has to be aware of possible variants and consequences.
- May only be a bet on different uses of a language concept.

Abstract Example

Figure 5 illustrates a Concept Tailoring state machine. The difference to the EntityType state machine is that there exists a third kind of element named *State Machine Declaration*. We designed that a *Declaration* can only be referenced by a *State Machine Type* element, but this is basically up to the language designer. It defines a context for state machines and limits their specification possibilities. The benefit is that information about the context of state machines becomes explicit and simple to change while otherwise it would be implicit and part of the language infrastructure. The *State Machine Declaration* does not necessarily need to be accessible to language users. It may also be hidden by the language infrastructure and loaded besides the user defined content. This approach is useful because the language infrastructure becomes less hard coded.

EuroPLoP '16, Publication date: 2016.

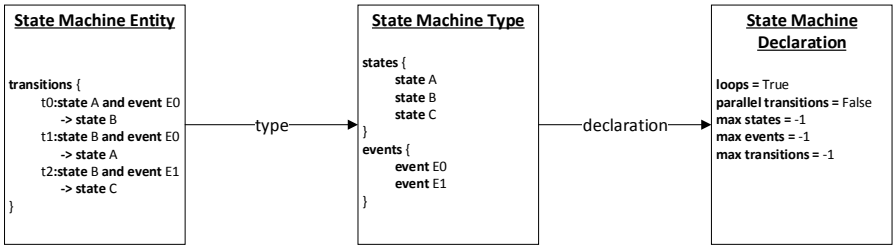


Fig. 5: Concept Tailoring example

Java Example

Listing 7 shows the Java realization. The class *FSMDeclaration* (lines 1-4) is responsible for making infrastructure information explicit. In this example only the maximum number of states is configurable, but it is imaginable that also other properties are available (i.e. see Figure 5). Note, that such a configuration influences the whole depicted concept and not only the *EntityType* part. The classes *FSMType* (lines 6 - 18) and *FSMEntity* (lines 24 - 45) are similar to the EntityType pattern. The class *FSMType* includes a method *validate* (lines 15 - 18) that carries out checks based on the used *FSMDeclaration*.

Listing 7: Java realization of the Concept Tailoring example.

```
1  public class FSMDeclaration {
2   public int maxNumberStates = -1;
3   // ...
4  }
5
6  public class FSMType {
7   public FSMDeclaration declaration;
8   public Vector<String> states = new Vector<String>();
9   public Vector<String> events = new Vector<String>();
10
11   public FSMType(FSMDeclaration declaration) {
12    this.declaration = declaration;
13   }
14
15   public boolean validate() {
16    // Implementation specific
17   }
18  }
19
20  public class Transition {
21   // Equal to Atomic Concept
22  }
23
24  public class FSMEntity {
25      public FSMType type;
26      public String currentState = "";
27
28      private Vector<Transition> transitionTable = new
            Vector<Transition>();
29
30      FSMEntity(FSMType type) {
31       this.type = type;
32      }
33
34      public void addTransition(/* ... */) {
35       // Equal to Atomic Concept
36      }
37
38      public boolean validateStateMachine() {
39       // Implementation specific
40      }
41
42      public void move(String event) {
43       // Equal to Atomic Concept
44      }
45  }
```

Listing 8 illustrates the application of the classes *FSMDeclaration*, *FSMType* and *FSMEntity*. It is similar to EntityType, except that a restricting configuration, expressed by the *FSMDeclaration*, is added.

Listing 8: Application of the Java example

```
public static void main(String[] args) {
 FSMDeclaration decl = new FSMDeclaration();
 decl.maxNumberStates = 5;

 FSMType def = new FSMType(decl);
 def.states.add("A");
 def.states.add("B");
 def.states.add("C");
 def.events.add("E0");
 def.events.add("E1");

 FSMEntity entity = new FSMEntity(def);

 entity.addTransition("A", "E0", "B");
 entity.addTransition("B", "E0", "A");
 entity.addTransition("B", "E1", "C");
 entity.currentState = "A";

 if (entity.validateStateMachine()) { // Correct SM
  entity.move("E0");
  // ...
 }
}
```

Known Uses

—Profiles in UML [Object Management Group (OMG) 2015] are an application of this pattern. They define a context for general UML concepts and can restrict how elements are used. Further, stereotypes can be used to make parts of the tooling infrastructure adjustable by language users.

—In principle, XML schemas are a realization of this pattern, as schemas define the legal building blocks of an XML document [W3C 2016]. A schema can be loaded by a tooling infrastructure for validating a document.

—In the context of object models, Fowler [Fowler 1997] names this pattern *Knowledge Level*. The author shortly describes it as *a group of objects that describe how another group of objects should behave (also known as a meta level)*.

—Also in the context of object models, Yoder et al. [Yoder et al. 2001] describe an object-oriented architecture style called *Adaptive Object Models* where this pattern plays a key role. There it is identified as a realization of the Strategy pattern where rules validate *Entities* and *EntityTypes*.

7. EXPERIENCE REPORT

In our research project we had the task to implement a modeling language for describing components and compositions that are used by a component-based infrastructure. In our case, a component represents an executable that offers datapoints which are set, read and parameterized during execution. A composition represents an entity that connects datapoints and executes existing components or other user-defined compositions. The whole architecture is inspired by the IEC 61131 standard for programmable logic controllers [John and Tiegelkamp 2010]. In the following we concentrate on the flexibility and configurability of the language concept *Component*.

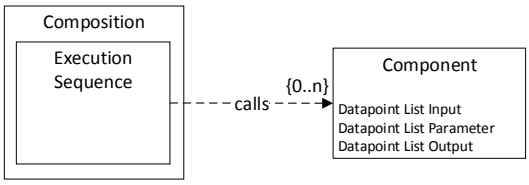


Fig. 6: Sketch of the first version we had in mind.

Figure 6 illustrates the solution we had first in mind. A composition owns an element called execution sequence. This execution sequence referes to components and calls them during the execution of the composition. The components contain datapoints inside the datapoint lists *Input*, *Parameter* and *Output*.

Having one language element representing the concept *Component* is not suitable for our case as we want to have different entities of components of the same definition. Therefor, we applied the EntityType pattern. Figure 7 illustrates this solution. The *Component Type* is used to define datapoints which are contained inside the three different types of datapoint lists. A *Component Entity* has to reference one *Component Type*. Further, a *Component Entity* owns a language element *Settings* that can be used to configure datapoints that are contained inside the datapoint list *Parameter*.

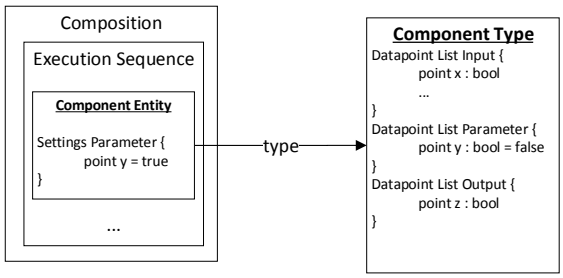


Fig. 7: Intermediate version that applies the EntityType pattern.

We were not happy with this version of our modeling language. The language compiler had to know a lot about the meaning of language elements. For instance we had separate language elements that described input, parameter and output datapoint lists. The semantics of each list was hard-coded inside the language compiler. We solved this by applying the Concept Tailoring pattern in order to make the behavior of datapoint lists configurable. Figure 8 illustrates the result of that iteration. We split the concept of a component into three different kinds of language elements. A *Component Declaration* is used to declare the available datapoint lists and to define their behavior. A *Component Type* is used to describe a type of component with its unique datapoints. A *Component Entity* is used by the execution sequence of a composition. It represents the instantiation of a component and offers the possibility to parameterize datapoint lists that are configured to be settable. Now our language compiler only has to know about the behavior of the language concepts. But it does not have to know what a datapoint list called *Input* means because this information is now defined by the *Component Declaration*. Note, that concerning the language concept *datapoint list* we apply the EntityType pattern. The concept *execution sequence* does not need any special treatment and is a realization of the Atomic Concept pattern.

We stopped at this version of the modeling language because we currently do not find it justifiable to apply more configuration patterns. Theoretically, the Deep Configuration pattern could be applied on the language element *Component Entity* in order to reuse defined settings. Of course, this additional change would make the concept *Component* more fragile.

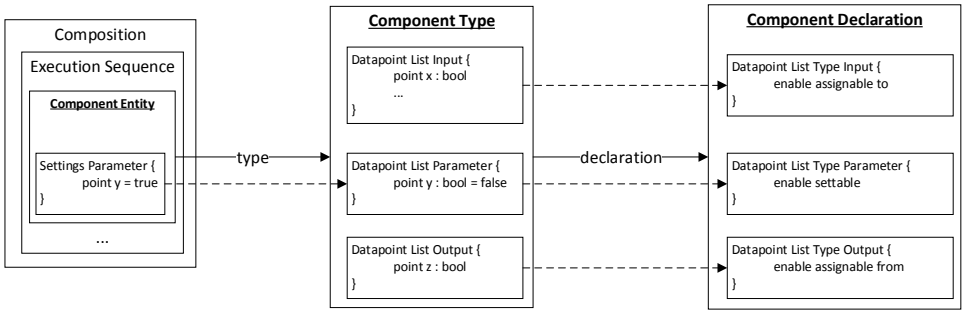


Fig. 8: Final version of the modeling language that applies the Concept Tailoring pattern on the language concept *component*.

8. CONCLUSION

To sum up, we presented four patterns describing different kinds of representing a concept through language elements. Each of these patterns has another effect on the configurability of a concept. Last, but not least, these patterns can be combined for describing a concept ever more flexible. If a language designer decides to apply all these patterns for a concept, the resulting language architecture probably looks like the *Adaptive Object Model* [Yoder et al. 2001].

Acknowledgments

We thank our shepherd Ralph Johnson for his inspiring advice. Further, we thank our EuroPLoP 2016 focus group for giving very helpful hints and suggestions.

REFERENCES

Peter Coad. 1992. Object-oriented patterns. *Commun. ACM* 35, 9 (1992), 152–159.

Ivica Crnkovic, Severine Sentilles, Vulgarakis Aneta, and Michel R.V. Chaudron. 2011. A Classification Framework for Software Component Models. *IEEE Transactions on Software Engineering* 37, 5 (sep 2011), 593–615. DOI:http://dx.doi.org/10.1109/TSE.2010.83

Douglas Crockford. 2016. Introduction to JSON. (2016). http://json.org/

Martin Fowler. 1997. *Analysis Patterns: Reusable Object Models*. Addison-Wesley.

Martin Fowler. 2010. *Domain-Specific Languages*. Pearson Education.

Karl Heinz John and Michael Tiegelkamp. 2010. *IEC 61131-3: Programming Industrial Automation Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg.

Ralph Johnson and Bobby Woolf. 1998. The Type Object Pattern. In *Pattern Languages of Program Design 3*. Addison-Wesley.

Object Management Group (OMG). 2015. OMG Unified Modeling Language (OMG UML), Version 2.5. (2015). http://www.omg.org/spec/UML/2.5/

W3C. 2016. XML Schema. (2016). https://www.w3.org/XML/Schema

Joseph W. Yoder, Federico Balaguer, and Ralph Johnson. 2001. Architecture and Design of Adaptive Object-models. *SIGPLAN Not.* 36, 12 (Dec. 2001), 50–60. DOI:http://dx.doi.org/10.1145/583960.583966

An Integrated Approach for Resilience in Industrial Control Systems

Johannes Iber, Tobias Rauter, Michael Krisper and Christian Kreiner
Institute of Technical Informatics, Graz University of Technology, Austria
{johannes.iber, tobias.rauter, michael.krisper, christian.kreiner}@tugraz.at

***Abstract*—New generations of industrial control systems offer higher performance, they are distributed, and it is very likely that they are internet connected in one way or another. These trends raise new challenges in the contexts of reliability and security. We propose a novel approach that tackles the complexity of industrial control systems at design time and run time. At design time our target is to ease the configuration and verification of controller configurations through model-driven engineering techniques together with the contract-based design paradigm. At run time the information from design time is reused in order to support a modular and distributed self-adaptive software system that aims to increase reliability and security. The industrial setting of the presented approach are control devices for hydropower plant units.**

I. Introduction

Cyber-physical systems (CPS) are the recent evolution of networked embedded systems that blend physical mechanisms with software. One part of the broad range of different kinds of CPS are industrial control systems. According to a National Institute of Standards and Technology workshop report [1] the key challenges of CPS development include what is needed to cost-effectively and rapidly build-in and assure safety, reliability, availability, security and performance of next generation CPS. Industry is using more and more commercial off-the-shelf hardware platforms, which are inexpensive and offer high performance. The downside of these platforms is that typically they only offer limited safety and fault tolerance features [2] [3]. This is a matter for concern since industrial control systems are increasingly becoming the targets of security attacks [4].

The inherent problem of CPS is complexity. This issue is going to escalate even more for industrial control systems, because they are becoming large-scale distributed systems. Systems of this kind must deal with uncertainty, change during operation and moreover be scalable and tolerant to threats [5].

With this work we contribute a novel integrated approach in which we aim to manage and deal with the negative effects of complexity at design time and run time. The industrial setting of our approach is that of control systems for hydro-electrical power plants, which also happens to be the context of our ongoing research project.

Concerning design time, we apply the principles of model-driven engineering and propose to utilize models that describe the available hardware resources, the control logic, and the deployment of the control logic onto specific hardware resources. Furthermore, we apply the paradigm contract-based design for enhancing models with arbitrary non-functional properties, e.g. timing, security, safety, and performance. To ensure that the configuration of a control system is valid we use verification tools that are specialized on one of these properties.

To ensure resilience to security attacks or hardware faults at run time, just a correct configuration of a distributed control system alone is not enough. Tackling such threats requires mechanisms that are specialized in detecting anomalies and adapting the system. Thus, we are proposing a self-adaptive software system, named Scari (Secure and reliable infrastructure), that provides the means to implement and orchestrate adaptive mechanisms. Scare offers a knowledge base that reflects the current state of a system as models. These models are constructed based on design time models and information available at run time. If a mechanism wants to change a system it can verify the changes with similar tools as those used for the design time models.

The remainder of this paper is structured as follows: Section II provides a brief overview of the related work. Section III explains our industrial setting. Section IV shows the approach in detail. Section V demonstrates a use case. In Section VI we shortly discuss the approach. Finally, concluding remarks and future work are given in Section VII.

II. Related Work

In the following subsections we describe three research areas which are significant for our approach. The first area is contract-based design, a well-known paradigm that we use for capturing non-functional properties at design time and run time. The second area is about self-adaptive software systems, a method we utilize at run time. In the last subsection, we present in brief Models@Run.time, a flavor of self-adaptive software systems where models play a key-role.

A. Contract-based Design

Contract-based design usually sees a component as an abstraction, a hierarchical entity that represents a single unit of design [6] [7]. In the context of contract-based design a component can represent e.g. a software module, a composition, a complex system or even a physical device.

The essence of this paradigm is to decompose a component into different independent views referred to as contracts, which capture the behavior of a functional or non-functional property under certain conditions [7] [8]. Functional properties describe

the function of a system or component, i.e. behavior, input, or output data. Non-functional properties (also known as extra-functional) provide additional information and give insight into the behavior and capability of a system or component [9]. Examples of non-functional properties are safety, security, portability, performance.

The goal of contract-based design is to significantly reduce the complexity of design and verification by making functional and non-functional properties manageable. Additionally, a system can be viewed by selecting only appropriate contracts of interest.

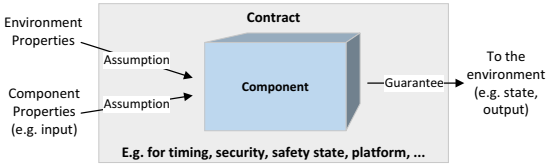


Figure 1. Assumptions and guarantees of a contract.

Informally, a contract is a set of assumptions and guarantees as illustrated in Figure 1.

An assumption asserts what a contract expects from the environment (including interactions with other components) or from the described component itself. Essentially it provides a specific context for the guarantees. The condition contained in an assumption can reference e.g. input data, events or system properties. In general, the available variables are set or inferred by the analysis environment.

A guarantee describes what a component provides to the environment as long as the corresponding assumptions hold. In the simplest case a guarantee states that a component just works under the constrained context. More complex contracts define limits for output data, environment characteristics or non-functional properties such as timing.

In another work we presented a generic modeling language for specifying contracts and contract-state machines [10] which we are also using for our approach in this work.

B. Self-Adaptive Software Systems

> *Self-adaptive software modifies its own behavior in response to changes in its operating environment. By operating environment, we mean anything observable by the software system, such as end-user input, external hardware devices and sensors, or program instrumentation. [11]*

Self-adaptive software systems can follow internal or external approaches; typically the external (architectural) approach is used [12]. An internal approach interweaves application and adaption logic based on programming language features like exceptions, conditions, and parametrization. The issue with that is that sensors, actuators, parallel adaption processes and actual purpose of an application are complicated to engineer within a single software design. This leads to notable drawbacks with respect to scalability, testability and maintainability.

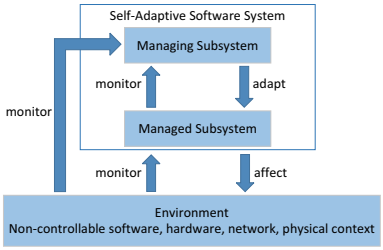


Figure 2. Parts of a self-adaptive software system (adapted from [13]).

In an external approach, as illustrated in Figure 2, the domain-specific application logic, which is termed *Managed Subsystem* is monitored by a *Managing Subsystem*. The *Managing Subsystem* is where the actual adaption logic resides. It additionally monitors the *Environment* that may consist of other software, hardware, network, or the physical context (including humans). On the basis of monitored data and analyzed problems, the *Managing Subsystem* decides whether and what to adapt inside the *Managed Subsystem*.

In general, a *Managing Subsystem* itself incorporates an adaptive loop that is responsible for reacting to problems originating from the monitors. Muccini et al. reveal in a systematic literature review [5] that, at least in literature, the MAPE-K loop is by far the dominant adaptive loop for CPS, with a share of 60%.

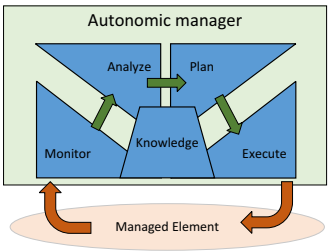


Figure 3. MAPE-K (adapted from [14]).

Figure 3 illustrates the MAPE-K adaptive loop [14]. It consists of the steps *Monitor*, *Analyze*, *Plan*, *Execute*, and a shared part representing *Knowledge*.

The *Monitor* step gathers information about the *Managed Element* that is usually related to the current performance and load of the system [15]. The *Analyze* step reasons about the data, identifies problems and attempts to find the source or cause of it. The *Plan* step reacts to the results of the *Analyze* step and creates a set of actions to solve the problem. The last step, termed *Execute*, implements these actions and changes the *Managed Element* through actuators.

Knowledge is the central point where all the information within a MAPE-K loop comes together. The *Monitor* step

stores its observed data there, the *Analyze* step uses it to find anomalies, the *Plan* step leverages it to create actions and gathers its policies and goals from there, and finally, the *Execute* step stores its record of executed actions in it.

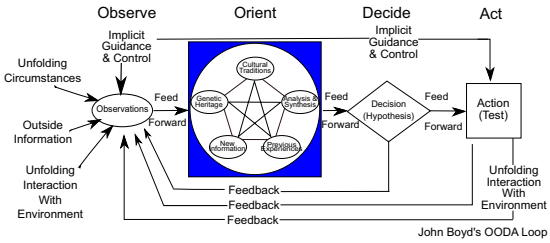


Figure 4. OODA-loop [16].

In our approach we use a combination of MAPE-K with John Boyd's OODA loop. Colonel John Boyd was a United States Air Force fighter pilot and Pentagon consultant who developed the first version of his OODA-loop for explaining how to achieve success in air-to-air combat in the 1950s. Later he expanded his groundbreaking work and hypothesized that it is the essence of winning and loosing of organizations and people [17]. As pointed out by other authors, this also suits well for self-adaptive system [18]. Figure 4 illustrates the type of data that can be observed. The first step in the OODA-loop, *Observe*, gathers, monitors, and filters data. Implicit guidance and control over this observations have significant influence for the following *Orient* step. In the *Orient* step a list of options is derived through analysis and synthesis, previous experience, new information, and genetic and cultural heritage (as the loop is intended for humans). The derived list of options is then forwarded to the *Decide* step where the best hypothesis is selected via a ranking. In the last step, the *Act*, the selected option is executed and tested whether the hypothesis was correct. As pointed out by John Boyd, "*orientation shapes observation, shapes decision, shapes action, and in turn is shaped by the feedback and other phenomena*" [17].

C. Models@Run.Time

Models@Run.Time is a term for describing the field of utilizing software models, specified according to the model-driven engineering principles for self-adaptive software systems [19]. Opposed to traditional model-driven engineering, the novelty in *Run.Time* refers to the fact that it describes the architecture of software and systems at design time.

Giese et al. [20] distinguish between three different kinds of run time models within a self-adaptive software system:

- **System Models**: This kind of models reflects an abstract view of the system itself. It allows an adaptive system to reason about the system and to simulate different kinds of configurations. As a consequence a model of this kind always needs to be in sync with the real system.
- **Context Models**: A run time model can be used to reflect the context of a system and to specify it in a processable way . The characteristics of the context in such a model can either be derived directly from the environment by sensors or indirectly derived from other observations.
- **Requirements Models**: This kind of models captures the requirements and goals of a self-adaptive system. In a way it sets the boundaries of what a system can do. Usually this relationship is unidirectional, meaning that a system is not intended to change its requirements, but it can, prioritize one over another.

III. Industrial Setting

The industrial setting of our approach are distributed control devices that operate hydro-power plant units. Figure 5 illustrates a simplified overview of such an industrial control system.

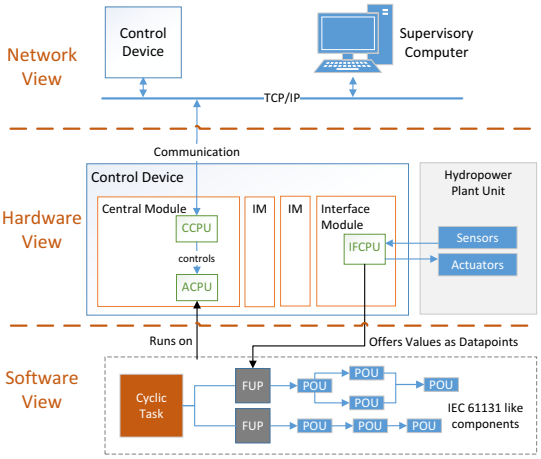


Figure 5. Overview of the target industrial control system.

On network view level, the control devices are connected via Ethernet and operated by supervisory computers. These supervisory computers are responsible for observing the state of physical processes and adjusting parameters of control devices in order to control the energy conversions. The observation and adjustment actions are done by using so-called datapoints, which are variables with specific data type such as Integer or Boolean.

The control devices are connected to hydropower plant units. Their functional responsibility is to operate these units through one of the four different functions: excitation, synchronization, protection and turbine control.

Technically, these devices have a programmable logic controller (PLC) architecture. Concerning the hardware view, a control device is built out of central modules and interface modules. A central module consists of a communication CPU (CCPU) and an application CPU (ACPU). The CCPU is responsible for network connections and controlling/monitoring the ACPU. It runs a customized Linux distribution. From the security point of view it protects the ACPU and verifies

incoming commands. The ACPU is a multi-core processor and executes the actual control logic. It runs a real-time operating system in order to ensure guaranteed cycle times. The interface modules are connecting the control device with sensors and actuators of the hydropower plant unit. Central modules and interface modules are connected via Ethernet.

The control logic software executed by the ACPU of a central module is component-based and heavily influenced by the IEC 61131 standard for programmable logic controllers [21]. It is hierarchically built out of components, compositions and tasks. Components are termed Program Organization Units (POU), and compositions are named Function Plans (FUP). POUs are coded in the programming language C and stored as binaries on the devices. Such POUs implement basic functions ranging from simple logic gates to complex algorithms. Consisting of POUs, FUPs are designed by plant engineers to implement specific control logic for a hydropower plant unit and are executed by cyclic tasks in real-time on the ACPU.

FUPs operate on datapoints that are set and read by the interface modules. The necessary datapoints are collected at the start of a cyclic task, the FUPs are then executed and the calculated datapoints are written back. The interface modules receive these datapoints and actuate accordingly. Datapoints can also be shared with other control devices or supervisory computers.

IV. Approach

Figure 6 illustrates an overview of our approach. On the left hand side we see models representing the configuration and logic of the industrial control system. Configurations of this kind are created manually by plant engineers and can be verified. On the right hand side we find a self-adaptive software system that detects anomalies and reacts to them. It verifies changes of the run time system by utilizing models originating from design time. These two parts are connected by a transformation step illustrated in the middle of Figure 6. In the following sections we discuss the three parts in more detail.

A. Design Time

At design time the goal is to have an abstract representation of the intended control system in order to ensure that the target system is correct in terms of functional properties such as datapoint compatibility and also in non-functional properties such as timing, security, safety.

Our approach is to use textual domain-specific modeling languages that enable plant engineers to specify the control logic in the form of tasks and FUPs. It is possible to specify how tasks are deployed and on what devices/central modules.

Figure 7 illustrates the four main kinds of domain-specific modeling languages that are used for describing the control logic and an industrial control system.

The *Component Metamodel* is used for specifying applications, cyclic tasks, FUPs, and the interfaces of POUs. An application is a container modeling the interaction between cyclic tasks. This is necessary because output datapoints from tasks can be used as input by tasks running on other devices. An application model contains this dependency information. The *Component Metamodel* is supposed to be manually used by plant engineers.

The *Resource Metamodel* is representing physical control systems. It includes interface modules, central modules, control devices, hydro-power plant devices and various network devices. This kind of models are created by querying the real networked system and are automatically managed by the tooling.

The *Deployment Metamodel* is used by plant engineers in order to specify where the control logic defined with the *Component Metamodel* is deployed.

The *System Configuration Megamodel* is the entry model for processing a configuration. It refers to the used applications, deployments, and physical resources. A megamodel is a model containing both the models and the relations between those models.

Supporting these modeling languages we use a contract-based design modeling language, which we present in another work [10]. Each entity such as a task or hardware part can basically refer to contracts in order to enhance them with arbitrary non-functional properties like timing, portability, security and safety.

Figure 6 illustrates on the left hand side how models created with the modeling languages from above can appear. It shows an application containing two tasks where one depends on the other. The dependency is deployed onto a specific physical connection, while the tasks reside on separate control devices. As we can see, each entity can be annotated with a contract.

Based on these models a system configuration can be verified in different ways. One way is structural. In the simplest form a configuration of this kind can be verified as to whether the connected datapoints between tasks, FUPs, and POUs are compatible. Another simple verification is to check whether connected distributed tasks deal with the same hydro-power plant unit or if they are misconfigured.

Utilizing contract-based design enhances these structural checks by adding a flexible mechanism that allows the introduction of arbitrary non-functional properties. For instance, a POU interface can be enhanced by a timing contract that guarantees a specific timing based on an assumed CPU, or it states how much memory it needs in order to guarantee its functionality. A task can be enhanced by adding contracts which state that it needs datapoints within a certain period and that they need to be measured using redundant and independent means. The structural information and the contracts can be processed by tools that are specialized for specific properties. The errors and warnings produced by such tools can be expressed using a separate modeling language that can annotate entities from the metamodels above.

We realize the metamodels technically using the Eclipse Modeling Framework [22] and use Xtext [23] for designing textual domain-specific languages. Additionally, we leverage the KLighD framework [24] for automatically visualizing the models graphically. Specialized verification tools can either

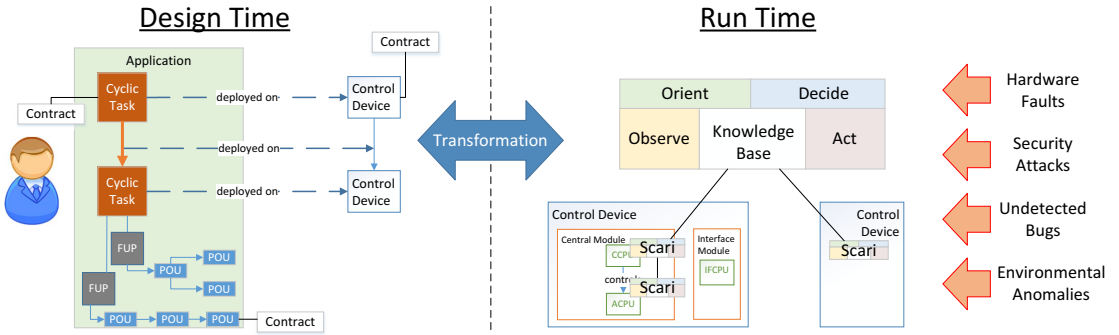


Figure 6. Overview of the presented approach.

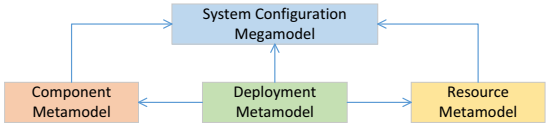


Figure 7. Overview of the design time metamodels.

utilize the models directly, or can be transformed to arbitrary formats.

B. Run Time

The goal at run time is to have a software that adapts the control system to various anomalies and faults in order to increase the reliability and security. We are not aiming to change the control logic itself. Instead, we want to ensure that the hardware and software stack below the intended control logic can perform as long as possible. In order to achieve this we are reusing the models from design time as an information source and verification input.

Our overall aim is to detect and adapt to anomalies and faults from different areas:

- **Hardware faults**: As an example, permanent memory cell faults in RAM or CPU registers, used by the ACPU or CCPU, can be detected with memory checks. After such a detection, a faulty location could be circumvented by reconfiguration of the operating system or by moving the control logic to another module or device. Further, permanent hardware faults in interface modules could be recognized and handled by using an alternative interface module. A control device should not only be able to recognize its own faulty hardware but also that of others. As datapoints are distributed to other control devices controlling other parts of the same hydropower plant unit, they should be able to observe and analyze that something might be wrong with the hardware of other control devices or networking devices. Ultimately, the control logic running on one device could be migrated to alternative ACPUs, central modules, or devices.
- **Security attacks**: Each control device in our setting knows from whom it receives or sends data to based on its configuration. This information could be used for detecting network security attacks or attackers that imitate control devices. Infected devices can be detected when the datapoints they are distributing suddenly develop odd behavior or do so over time and do not reflect the real environment. Additionally, unexpected behavior of devices can hint to security incidents, e.g. attempting access to control devices which they are not supposed to. Revealed attacks can be handled by blocking and isolating infected devices or network resources. Other kinds of attacks are for instance attempts to access restricted resources, or physical manipulation of sensors.
- **Undetected Bugs**: Software running on top of modules can be updated in order to gain new features or fix bugs. Albeit being a very useful feature to be able to update the software, these changes may introduce new bugs. Resource and performance monitors which observe the behavior of tasks could be used to detect changed and different behavior patterns.
- **Environmental anomalies**: The sensors and actuators which are the connection between the PLCs and the power plant environment (e.g. the water turbines) can break or drift over time. Detecting such environmental anomalies and reacting to them is also an important area in order to make a control system more reliable.

Detection and adaption for these areas could be done by implementing separate mechanisms specialized to one anomaly with a corresponding adaption method. However, this is a complex task involving the orchestration of these mechanisms in parallel and to ensuring that adaption mechanisms do not interfere with each other. Furthermore, some anomalies are cross-cutting. For instance, if a drift of a datapoint on a device is observed, this could indicate a hardware fault or a security attack. If a hardware fault and a security mechanism would react at the same time this could be fatal for the hydropower plant. We are thus proposing an infrastructure that allows orchestration of such diverse detection and adaption

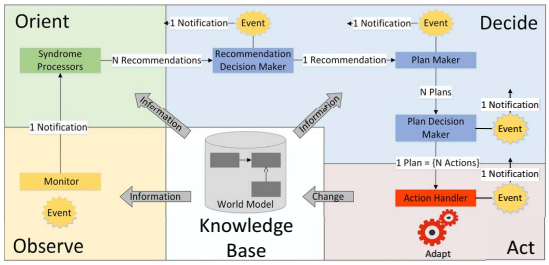


Figure 8. Scari adaptive loop.

mechanisms and ensures that only one is carried out at one point in time. We name our infrastructure **Scari** (**Sc**ure **a**nd **r**eliable **i**nfrastructure).

Figure 8 illustrates the adaptive loop of Scari. It consists of 5 parts, which are *Observe*, *Orient*, *Decide*, *Act*, and a common *Knowledge Base*.

The *Observe* part consists of *Monitors* that are specialized in discovering and measuring specific anomalies, for instance a drift in data. These monitors notify an arbitrary numbers of interested *Syndrome Processors*, residing in the *Orient* part. The *Syndrome Processors* are implementing a specific detection mechanism e.g. for hardware faults or security attacks. Technically, they can use any viable method for detection like machine-learning or simple thresholds. If one or several *Syndrome Processors* diagnose a problem, they recommend plan types for handling a situation. For instance a hardware fault *Syndrome Processor* may recommend circumventing a damaged module, while a security *Syndrome Processor* may recommend isolating a device. Next, the *Recommendation Decision Maker* selects the best recommendation on the basis of a definable prioritization for the covered events and chosen plan types. The selected recommendation is then forwarded to the *Plan Maker* that creates the actions for the plan type. Some plan types may be realized in different ways, we thus added a *Plan Decision Maker* that selects the plan with the least affected systems/resources and the lowest number of used actions. In the final part, the plan is executed by a *Action Handler*, and the system is adapted to the situation diagnosed by a *Syndrome Processor*. Each of the entities of the *Decide* and *Act* parts feed back their states as events. This enables the *Syndrome Processors* to log the state of their recommendations and to be notified in turn that the system has adapted. The *Knowledge Base* provides support for the other four parts. It contains the deployed models, including contracts, from design time and additional run time information. It serves as a source of knowledge for the *Observe*, *Orient* and *Decide* parts, while the *Act* part stores the executed changes of the system there.

The combination of MAPE-K with OODA leads in our opinion to the best of both concepts. MAPE-K introduces the *Knowledge Base* as a common information source for the different steps. OODA adds an explicit *Decide* part which is useful for the selection of recommendations. The *Plan* step of MAPE-K is distributed over several loosely coupled entities. We also take from OODA that each step gives feedback to *Monitors* and *Syndrome Processors*. This allows them to consider what happened with their notifications and recommendations.

As can be seen in Figure 6 on the right side, Scari runs on the ACPU and CCPU of each central module and on different network resources above. Each of these instances incorporates the five parts of the Scari adaptive loop, except the ACPU which lacks the *Knowledge Base* for performance reasons. This hierarchical organization of Scari has two advantages:

One is that a *Knowledge Base* only needs to *know its subgraphs*. If a *World Model* on a node changes, information is only propagated up to the parent nodes. A *Knowledge Base* may be configured to prune lower node data if it is not needed on the higher levels. For memory efficiency, information is only stored inside the nodes and up to specific layers, where it is actually needed. Distributing all information on all nodes would additionally lead to more network traffic.

The second advantage is that an adaptive loop only needs to *handle its subgraph*. A loop does not need to manage other parts of the overall control system which also eases the configuration of Scari. If it is not possible to adapt to an event occurring on a node, it can be escalated to a parent node that has more knowledge, more resources, and can therefore leverage more powerful adaption mechanisms. In our hydropower setting, it is conceivable that these adaption layers are even laid over different hydropower plants, where they are acting on a greater time scale.

Technically, we implemented Scari in C++ together with the Qt framework [25]. The *Knowledge Base* uses a C++ modeling framework developed by us and inspired by the Eclipse Modeling Framework [22]. It leverages the libgit2 library [26] for distributing models. Scari itself with its different adaptive layers is intended to be statically configured in advance. Furthermore, the architecture allows to dynamically add monitors, syndrome processors, plan creation mechanisms and actions at run time.

C. Transformation

The transformation from the design time models to the run time models resides on a supervisory computer. Next, the run time models are deployed by using the *Action Handlers* which set the corresponding *Knowledge Bases*. The other direction works by cloning a *Knowledge Base* to a supervisory computer where the run time models are transformed to a design time configuration.

V. Use Case

We demonstrate in the following use case how models at design time appear and how they can be relevant for run time adaptions.

A. Design Time

Figure 9 illustrates a visualization of the models at design time. A cyclic task with two FUPs is deployed on the central

module of control device *A*. Datapoints are sent to control device *B* over Ethernet. The CCPUs are responsible for shifting datapoints from one hardware interface to another and therefore introduce a certain delay. The execution of a POU takes a specific time depending on the speed of the processor. Timings are annotated as contracts and provided in advance. The timing for a functionality on a processor is measured extensively and packaged with the software deployed on the device. It is the responsibility of the tooling to gather the contracts of the hardware resources. The timing from the POUs is also given and packaged with the interfaces.

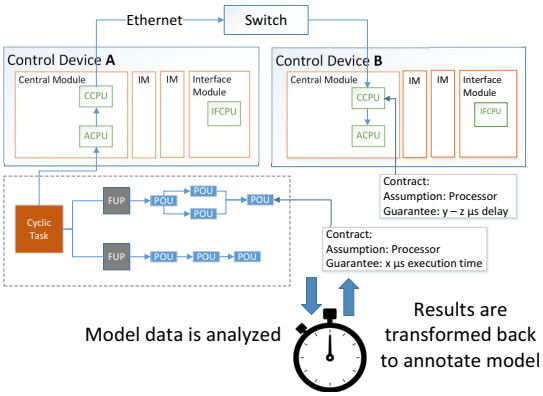


Figure 9. Design Time.

In order to verify a model of this kind it can be transformed into data that a timing analysis software can process. This software could calculate the worst case, average case and best case behavior, taking the Ethernet network into account. The results are then transformed back into a simple modeling language that annotates entities with errors and warnings.

If the various verifications are successful the design time model can be transformed and deployed by utilizing the *Action Handlers*. The available *Monitors*, *Syndrome Processors* and the configurations of the *Decision Makers* have to be configured separately. However, the *Monitors* and *Syndrome Processors* can adjust themselves depending on the deployed design time models.

B. Run Time

Figure 10 illustrates the case of a permanent hardware fault residing on an interface module. A *Monitor* running on the ACPU can observe if a datapoint is stuck and notify the higher Scari loop residing on the CCPU. The *Syndrome Processor* can reach the conclusion that there is probably a permanent hardware fault on the interface module. The Scari loop placed on the CCPU cannot change other devices and only knows about the architecture of nodes hierarchically below it. It thus escalates the situation to a higher loop that has more knowledge and resources.

Here, on this higher level, a *Syndrome Processor* could conclude that the faulty datapoints can be replaced by datapoints

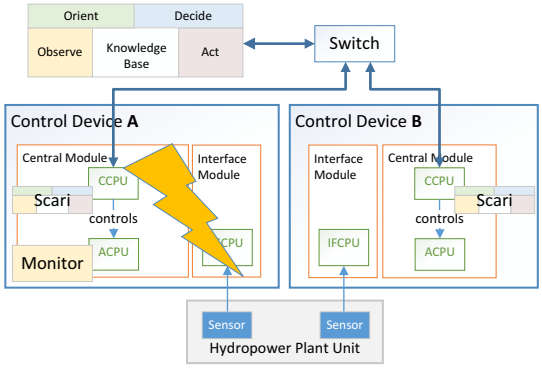


Figure 10. Run Time.

from another device that is connected with the same hydropower plant unit. This is possible because the four different functions excitation, synchronization, protection and turbine control are carried out by different devices. In such a case, the *Syndrome Processor* first adjusts a copy of the model and verifies it by utilizing similar tools as the tools used for verification of the design time models. For example, it verifies whether the timing contracts are still met. If the verification process is completed successfully, the *Syndrome Processor* recommends changing the system. The *Recommendation Decision Maker* approves the recommendation if it is the only one or if it has a higher priority than recommendations from other *Syndrome Processors*. It could reject a recommendation if the system has already changed in the meantime. After the acceptance, the *Plan Maker* creates the specific plan containing the intended actions. Next, if there are several possible plans, the *Plan Decision Maker* decides for the optimal plan depending on some criteria (e.g. least number of actions, least number of involved devices), which is then executed by an *Action Handler*. Finally, the faulty datapoints are replaced by valid ones and the device can continue to operate.

Note, that in the presented use case, we know beforehand that the cause is a hardware fault. Syndrome processors must analyze notifications by running algorithms that detect such patterns. It might also be that there are syndrome processors which are dedicated to diagnose security-related issues and interpret the situation differently.

VI. Discussion

Having a precise representation and the respective verification tools increases the confidence into a control system. However, it requires a rigorous development process where functional and non-functional properties of software components, hardware, and the dependencies between them are measured and captured in detail. This is difficult when software components are provided by different vendors. Solving this issue would require at least strict conventions and interface design as well as strong contracts and standards. Furthermore,

a thorough verified representation does not guarantee that it is working as intended by a plant engineer. The control logic itself can still be flawed which relates to the halting problem.

Applying an external self-adaptive software system for dealing with problems is a promising approach. The modular architecture allows adding arbitrary technologies for detection and adaption. It is even possible that syndrome processors recommend changing parts of the adaptive loop itself. They could recommend adding or replacing monitors, syndrome processors, plans, and actions. In this way, it becomes fully meta-adaptive. Together with models from design time, a system of this kind gains an understanding of what is being executed and about the inter-dependencies. In the context of adaption, a challenge for such systems would be to assure that they do not do more harm than good in their effort to prevent arising problems. The contracts from design time support the verification. Concerning security, it is important to secure the meta-adaptive system itself, otherwise it would represents a potential additional attack surface.

The transformation from run to design time is also an important property of our approach. This allows plant engineers to use a modified system as new base model for manual changes. Furthermore, it visualizes the current state which supports understanding and maintaining such systems.

VII. Conclusion and Future Work

In this work we outlined a novel approach of how models together with contracts can be utilized at design and run time in an industrial control system setting. At design time, textual domain-specific languages are leveraged for specifying control logic that can be verified by various tools. Through the use of contract-based design we support the enrichment with non-functional properties, such as timing, resource consumption, safety and security. At run time, we propose a self-adaptive software system that is built around models as a source of knowledge and verification. The goal of this self-adaptive software is to increase the reliability and security. It utilizes a novel adaptive loop that is a combination of the MAPE-K and OODA loop.

At the time of this writing, we implemented the textual domain-specific languages and the Eclipse-based tooling around them. We are currently exploring the space of validation and verification possibilities in order to increase the confidence that a configuration is correct. We also plan to do performance measurements and case studies in order to gather more data about the suitability of this adaptive loop in embedded systems.

We have realized the adaptive loop of Scari itself in the context of the run time part of our approach and are in the stage of implementing and trying out different detection and adaption mechanisms. In the near future we plan to investigate methods for configuring a self-adaptive system and ways of dealing with interferences caused by human operators. Furthermore, we intend to explore how models can help to assure that an adaption really performs as intended.

References

[1] NIST, "Foundations for Innovation in Cyber-Physical Systems," Tech. Rep., 2013.

[2] M. S. Alhakeem, P. Munk, R. Lisicki, H. Parzyjegla, H. Parzyjegla, and G. Muehl, "A Framework for Adaptive Software-Based Reliability in COTS Many-Core Processors," in *ARCS 2015*.

[3] A. Höller, B. Spitzer, T. Rauter, J. Iber, and C. Kreiner, "Diverse Compiling for Software-Based Recovery of Permanent Faults in COTS Processors," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*. IEEE, 2016.

[4] B. Miller and D. Rowe, "A survey SCADA of and critical infrastructure incidents," in *RIIT '12*. ACM Press, 2012.

[5] H. Muccini, M. Sharaf, and D. Weyns, "Self-adaptation for Cyber-physical Systems: A Systematic Literature Review," in *SEAMS*. ACM Press, 2016.

[6] A. Sangiovanni-Vincentelli, W. Damm, and R. Passerone, "Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems," *European Journal of Control*, vol. 18, no. 3, 2012.

[7] P. Nuzzo, Huan Xu, N. Ozay, J. B. Finn, A. L. Sangiovanni-Vincentelli, R. M. Murray, A. Donze, and S. A. Seshia, "A Contract-Based Methodology for Aircraft Electric Power System Design," *IEEE Access*, vol. 2, 2014.

[8] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis, "Multiple Viewpoint Contract-Based Specification and Design," 2008.

[9] S. Sentilles, P. Štěpán, J. Carlson, and I. Crnković, "Integration of Extra-Functional Properties in Component Models," in *Component-Based Software Engineering*. Springer Berlin Heidelberg, 2009.

[10] J. Iber, A. Höller, T. Rauter, and C. Kreiner, "Towards a generic modeling language for contract-based design," in *ModComp 2015 Workshop Proceedings*.

[11] P. Oreizy, M. Gorlick, R. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf, "An architecture-based approach to self-adaptive software," *IEEE Intelligent Systems*, vol. 14, no. 3, 1999.

[12] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 4, no. 2, 2009.

[13] D. Weyns, B. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, and K. M. Göschka, "On Patterns for Decentralized Control in Self-Adaptive Systems," in *Software Engineering for Self-Adaptive Systems II*, 2013.

[14] J. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, 2003.

[15] Y. Brun, R. Desmarais, K. Geihs, M. Litoiu, A. Lopes, M. Shaw, and M. Smit, *A Design Space for Self-Adaptive Systems*. Springer Berlin Heidelberg, 2013.

[16] Wikimedia Commons. (2014) Ooda loop. [Online]. Available: https://commons.wikimedia.org/wiki/File:OODA.Boyd.svg

[17] J. R. Boyd, "The Essence of Winning and Losing," 1996. [Online]. Available: http://dnipogo.org/john-r-boyd/

[18] A. Chandra, P. R. Lewis, K. Glette, and S. C. Stilkerich, *Reference Architecture for Self-aware and Self-expressive Computing Systems*. Springer International Publishing, 2016.

[19] G. Blair, N. Bencomo, and R. B. France, "Models@ run.time," *Computer*, vol. 42, no. 10, 2009.

[20] H. Giese, N. Bencomo, L. Pasquale, A. J. Ramirez, P. Inverardi, S. Wätzoldt, and S. Clarke, *Living with Uncertainty in the Age of Runtime Models*. Springer International Publishing, 2014.

[21] K. H. John and M. Tiegelkamp, *IEC 61131-3: Programming Industrial Automation Systems*. Springer Berlin Heidelberg, 2010.

[22] Eclipse Foundation, "Website of the EMF Project," http://www.eclipse.org/modeling/emf/, 2017.

[23] ——, "Website of the Xtext Project," http://www.eclipse.org/Xtext/, 2017.

[24] C. Schneider, M. Spönemann, and R. von Hanxleden, "Just model! putting automatic synthesis of node-link-diagrams into practice," in *IEEE Symposium on Visual Languages and Human Centric Computing*, 2013.

[25] The Qt Company, "Website of the Qt Framework," https://www.qt.io/, 2017.

[26] LibGit2, "Website of the libgit2 Library," https://libgit2.github.com/, 2017.

Patterns grasping the trade-off between distributing data and information

JOHANNES IBER, TOBIAS RAUTER, MICHAEL KRISPER and CHRISTIAN KREINER, Institute of Technical Informatics, Graz University of Technology

Today, we are at the dawn of the age of cyber-physical systems and internet of things. One of the commonalities these areas share is that such systems typically consist of networks of entities with means to gather data about the state of the surrounding environment. A fundamental design decision in such settings is whether to transfer data to more capable entities or to analyze data at the sensing entity and to share the resulting information. With this work, we discuss this trade-off by grasping it with three patterns, namely the *LOCAL DATA PROCESSING*, *CENTRAL DATA PROCESSING*, and *MIXED DATA PROCESSING* patterns.

CCS Concepts: •**Software and its engineering** → **Patterns;** *Designing software;*

Additional Key Words and Phrases: raw data, information, distributed systems

ACM Reference Format:

Johannes Iber, Tobias Rauter, Michael Krisper, and Christian Kreiner. 2017. Patterns grasping the trade-off between distributing data and information *EuroPLoP* (July 2017), 7 pages.
DOI: https://doi.org/10.1145/3147704.3147724

1. INTRODUCTION

In many kinds of distributed systems there is the fundamental decision to make whether raw data or refined information should be distributed for providing a service. In a nutshell, distributing raw data has the advantage that every interested entity can know everything. Distributing information (with this term we mean states or statements mined from the raw data) has the advantage that it is usually more compact and reliefs receiving entities from reprocessing raw data. For instance, a temperature sensor could continuously stream its sensed temperature to an interested entity. Such a case we consider as distributing raw data. Distributing information would be to average the temperature of the last ten minutes. There is a loss of detail, but the network gets less strained and the interested entity does not need to carry out analyzation mechanisms. We use the term *entity* for all participants of a distributed system, e.g. a hardware device, a software running on a smartphone, or a server.

The primary target audience of this work are engineers that face the challenge of designing a reactive distributed system. In the following, we present our attempt of grasping the named trade-off with the *LOCAL DATA PROCESSING*, *CENTRAL DATA PROCESSING*, and *MIXED DATA PROCESSING* patterns. In Section 3 we present the patterns in the context of self-driving vehicles. Finally, Section 4 concludes this work.

978-1-4503-4848-5/2017/07...$15.00
DOI: https://doi.org/10.1145/3147704.3147724

2. PATTERNS

2.1 LOCAL DATA PROCESSING

Problem. Entities, part of a distributed system, need to provide services. For doing that, they have to exchange states about the environment or themselves.

Forces

—*Throughput*: Distributing raw data strains the available network.
—*Performance*: Analyzing mechanisms can be executed on the entity.
—*Standardization*: Protocols for information sharing are available.
—*Dependency*: Communication between the entities is necessary.

Solution. Each entity is responsible for analyzing its sensed raw data to create higher level information. The entities exchange higher level information instead of raw data streams. Figure 1 illustrates the case that each participating entity processes its own data and shares the resulting information with each other.

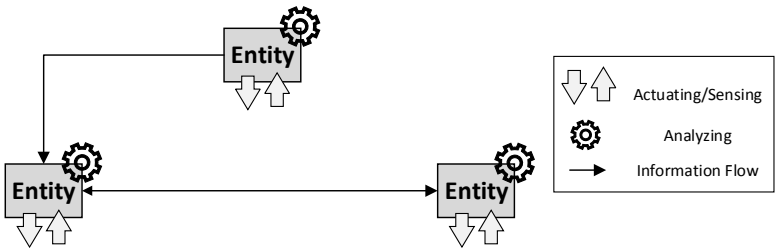


Fig. 1. LOCAL DATA PROCESSING

Consequences

+ Information is interpreted locally from data right after it is gathered by an entity .
+ The size, in terms of bytes, and frequency of information is lower than raw data, therefore this pattern produces less network traffic than distributing raw data.
+ Information sharing entity is in control of deciding what is private or confident.
+ Entities do not depend on a central entity.
- Refining information from raw data consumes computing power which may not be available on sensing entities.
- Big data effects, e.g. detecting patterns in a large amount of data, are complicated to achieve as every entity only sees its part of the environment.
- Each entity needs to be updated if one wants to change the analyzing software.
- If too much information is distributed, the network is still strained.

Known Uses

—Classic Supervisory Control And Data Acquisition (SCADA) systems consist of master terminal units (MTUs) and remote terminal units (RTUs). RTUs are devices that control objects in the physical

world, like hydro-power plant units, and which are themselves controlled by MTUs. An example of a RTU is a programmable logic controller, where a control logic processes data sensed from a physical object and reacts within a fixed period of time. Such controllers are supervised by local MTUs that can intervene by adjusting parameters or by collecting live data from the RTUs. The RTUs themselves can also be interlinked in order to provide data (for instance the current voltage) to each other. Depending on the timing constraints, a RTU may need to control within a short period of time and thus refines the information of the data itself in order to calculate an appropriate response [Boyer 2009]. This is a realization of the presented pattern if such a RTU shares the calculated information and not a stream of the sensed raw data with a MTU or other RTUs.

—Vehicle-to-vehicle and vehicle-to-infrastructure technologies are about sharing e.g. safety-relevant information between vehicles and infrastructure like traffic lights, street signs, and roadways. For instance, vehicles could share threats or hazards originating from other vehicles and take pre-emptive actions to avoid and mitigate crashes [Narla 2013]. An application of the *LOCAL DATA PROCESSING* pattern in this context is the local refining of environment information from raw data coming from various techniques like radar, lidar, GPS, and computer vision. The interpreted information is then shared with other vehicles that can take such a information into account for refining their own state of the environment.

—It is typical in wireless sensor networks that sensor nodes preprocess the raw data for lowering the cost of communication [Tubaishat and Madria 2003]. Thus, such nodes disseminate information instead of raw data.

2.2 CENTRAL DATA PROCESSING

Problem. A service provided by an entity needs raw data sensed by other entities. Collecting higher level information from others would hide patterns and dependencies which would lead to a worse service.

Forces

—*Throughput*:Distributing raw data to all entities participating in the distributed system strains the network.
—*Performance*: Most entities are too weak for leveraging costly analyzing mechanisms.
—*Timing*: A reaction does not need to follow immediately after a data has been analyzed.
—*Analyzability*: Only distributing information would hide patterns and dependencies.

Solution. Gather raw data at one central entity responsible for the intended service and utilize powerful analysis methods (e.g. machine learning) for calculating appropriate actions. Other entities in the network are only responsible for sensing the environment, transmitting raw data to the accumulating entity and executing actions send from that entity. Figure 2 shows an entity that accumulates raw data from others and shares the refined information with interested entities.

Consequences

+ All data are available to the analyzing entity.
+ The data analyzer can be optimized or replaced at one point.
+ Eases the use of online machine learning algorithms as more training data are available.
+ Easier to develop as one has only to deal with data at one point.
+ Cheap sensing and actuating entities.
- Transferring raw data produces a lot of network traffic and it may need some time.
- Entities between the data source and sink need a certain amount of memory for temporary storage.

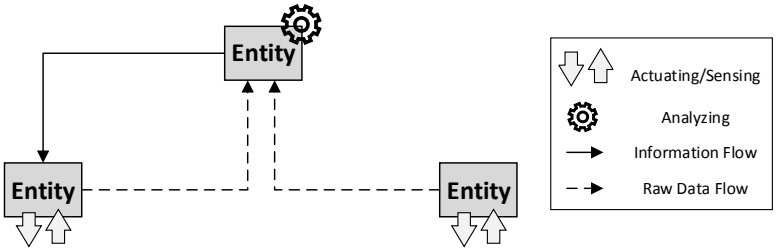


Fig. 2. CENTRAL DATA PROCESSING

- Central point needs a decent understanding of where the data comes from and about the participating entities.
- If a sensing entity looses its network connection it may become useless.
- Central entity may become a single point of failure.
- Central entity knows everything. This is relevant concerning privacy.
- Central entity may need a vast amount of computational power.

Known Uses

—The current state-of-the-art concerning the integration of the internet of things and cloud computing is an application of the *CENTRAL DATA PROCESSING* pattern. That is, lots of things feed data to clouds, where it is processed and refined to information that is the base for providing an intelligent service [Daz et al. 2016]. For instance, a microphone may record a spoken command that is send to the cloud where the command gets analyzed. A response to a command could be that the living room lighting is switched on.

—[Reinfurt et al. 2016] mention *Aggregating Device Gateway* as a variant of the *Device Gateway* pattern where a gateway for internet of thing devices accumulates messages for refining them to a higher level information. E.g. a gateway averages the temperature readings of several devices and forwards such an information to the backend once a minute.

—As mentioned in the pattern above, classic SCADA systems consist of MTUs and RTUs. Depending on the supervised physical process, it may be required for some control calculations to centralize raw input data at a MTU. Such a MTU would complete the control functions and send the resulting orders to the appropiate RTUs to effect the control [Boyer 2009].

—[Sousa et al. 2017] present the *LOG AGGREGATION* pattern where, in the domain of cloud computing, log files of different services are stored at a central entity. The main advantage is that developers can quickly query and visualize information from logs. This is useful for debugging a system consisting out of different services. For instance it is easier to track the behavior of a single user over different services in order to find out what went wrong.

2.3 MIXED DATA PROCESSING

Problem. Entity needs to provide a service even if network connections fail. The intended service can be improved if raw data and information from other entities is available.

Forces

—*Connectivity*: Network is temporary available and is unreliable.
—*Analyzability*: The intended service gets better if more data and information is available.
—*Standardization*: Protocols for sharing data and information are available.
—*Availability*: If network fails, the entity still has to provide a (degraded) service.

Solution. The entity realizes an analyzing mechanism that provides the service in a degraded form with the stored or currently sensed data and refined information. If a network is available, the entity may collect information or raw data from others. Another possibility would be that the entity connects to an other realizing the *CENTRAL DATA PROCESSING* pattern as soon as a network is available. Figure 3 shows entities that can share raw data or information with more capable entities. If a connection gets cut off the single entity is able to provide a degraded service.

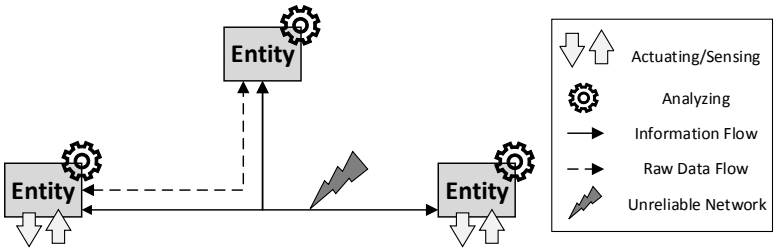


Fig. 3. MIXED DATA PROCESSING

Consequences

+ Provided service can degrade if network is not available.
+ Reduces the impact of an unreliable network.
+ Minimizes the latency between sensing data and providing a reaction.
- Not suitable for every kind of service.
- Entity needs more computational power than it would need for just transferring sensed data.
- The more powerful entity may know everything. This is relevant concerning privacy.

Known Uses

—Navigation applications can calculate routes without an internet connection. If an internet connection is available, such systems can leverage powerful datacenters that have access to current traffic information. With traffic information the calculated routes can be optimized and a more useful service is provided to a user.
—[Reinfurt et al. 2016] present the *Local Processing Gateway* as a variant of the *Device Gateway* pattern. Such a gateway mirrors or replaces functionality of the backend. This can minimize the communication with the backend and reduces latency. Another advantage is that it reduces the impact from a connection loss between gateway and backend.
—Edge-centric computing is a novel paradigm where computations and services are not gathered in central clouds, but also reside on edge entities, like smart phones, tablets, routers, or media centers

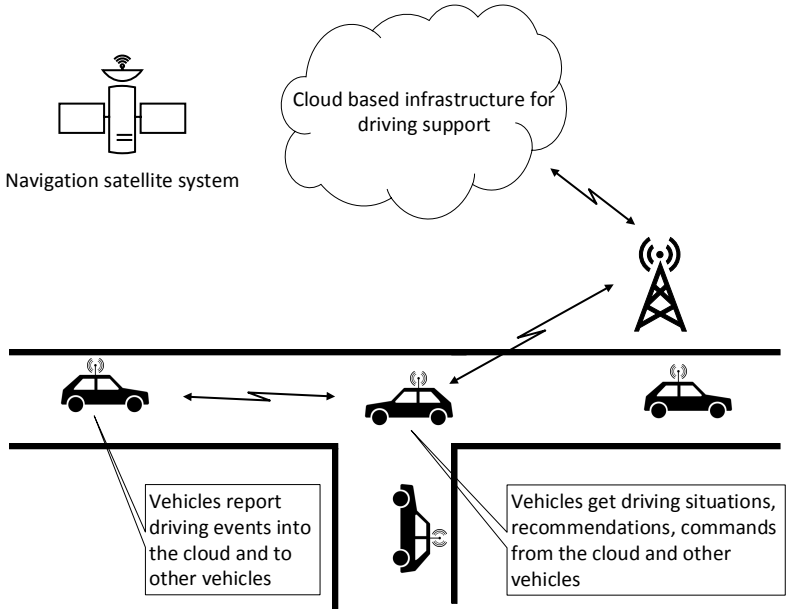


Fig. 4. Self-Driving Vehicle Scenario (Based on [Messnarz et al. 2017])

[Garcia Lopez et al. 2015]. Based on the needs of a service, such a system would dynamically reconfigure itself to process data locally together with near entities or fall back to powerful clouds where a vast amount of data and information come together.

3. SELF-DRIVING VEHICLE SCENARIO

Figure 4 illustrates an advanced driver assistance system where vehicles are connected to a cloud based infrastructure [Messnarz et al. 2017]. The connections between the vehicles and cloud are based on well established mobile internet technologies. Disconnections can take place at any time and cannot be avoided. That implies in-vehicle systems have to be capable of maintaining some mode of safe operation even when suddenly disconnected. Vehicles determine their position by fusing the results of more than one positioning system, thereby improving position accuracy. Navigation satellite systems based on GPS, Glonass, Galileo, and more, are well established and globally available. Supplementary technologies include received signal strength based positioning, e.g. by using mobile internet base stations.

When connected, vehicles can report and receive data to and from cloud services that operate on a fleet level, as well as communicate with nearby other vehicles (vehicle-to-vehicle) and infrastructure. Vehicles reporting to a fleet-level infrastructure can supply a broad range of driving, environment, and sensor events together with the vehicle identification and position to the cloud infrastructure. On this level it becomes possible to analyze data on overall fleet level, and such analysis can include the vehicle position. In turn, vehicles can receive for their current position both fleet-typical behavior under certain

environmental conditions (matching the current conditions), and any real-time exceptional conditions (e.g. accident warnings, deviations of nearby vehicles from normal behavior).

Now, in such a system we find the patterns from above multiple times. The vehicle-to-vehicle communication is a realization of the *LOCAL DATA PROCESSING* pattern. Vehicles locally refine their sensed data and share high-level information, like deviations of nearby vehicles or accidents, directly with each other. The *CENTRAL DATA PROCESSING* pattern can be found in the relationship of the vehicle with the cloud. The cloud refines the received data from the vehicles and returns information, like steering related information, to a specific vehicle. The *MIXED DATA PROCESSING* pattern can also be found in this relationship e.g. for navigation routes. The cloud may provide precise and near to optimal routing information, but if a connection gets disrupted the vehicle can navigate on its own.

4. CONCLUSION

To sum up, we presented three patterns describing different strategies of handling data. The *LOCAL DATA PROCESSING* pattern illustrates the case that entities, part of distributed system, are locally refining information. The *CENTRAL DATA PROCESSING* pattern introduces the concept of refining information from raw data at a central entity. The *MIXED DATA PROCESSING* pattern reflects a combination of these two patterns, where an entity decides dynamically whether to refine information locally or to propagate the data to a central entity. We discussed these patterns with their consequences and known uses. Note, that the three patterns can occur several times within a system depending on the specific use case.

Acknowledgments

We thank our shepherd Michael Weiss for his inspiring and constructive comments. Further, we thank our EuroPLoP 2017 focus group for giving very helpful hints and suggestions.

REFERENCES

Stuart A. Boyer. 2009. *Scada: Supervisory Control And Data Acquisition* (4th ed.). International Society of Automation, USA.

Manuel Daz, Cristian Martn, and Bartolom Rubio. 2016. State-of-the-art, challenges, and open issues in the integration of Internet of things and cloud computing. *Journal of Network and Computer Applications* 67, Supplement C (2016), 99 – 117. DOI:http://dx.doi.org/https://doi.org/10.1016/j.jnca.2016.01.010

Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, and Etienne Riviere. 2015. Edge-centric Computing: Vision and Challenges. *SIGCOMM Comput. Commun. Rev.* 45, 5 (Sept. 2015), 37–42. DOI:http://dx.doi.org/10.1145/2831347.2831354

Richard Messnarz, Alexander Much, Christian Kreiner, Miklos Biro, and Jenny Gorner. 2017. *Need for the Continuous Evolution of Systems Engineering Practices for Modern Vehicle Engineering*. Springer International Publishing, Cham, 439–452. DOI:http://dx.doi.org/10.1007/978-3-319-64218-5_36

Siva RK Narla. 2013. The evolution of connected vehicle technology: From smart drivers to smart cars to... self-driving cars. *Institute of Transportation Engineers. ITE Journal* 83, 7 (2013), 22.

Lukas Reinfurt, Uwe Breitenbücher, Michael Falkenthal, Frank Leymann, and Andreas Riegg. 2016. Internet of Things Patterns. In *Proceedings of the 21st European Conference on Pattern Languages of Programs (EuroPlop '16)*. ACM, New York, NY, USA. DOI:http://dx.doi.org/10.1145/3011784.3011789

Tiago Boldt Sousa, Hugo Sereno Ferreira, Filipe Figueiredo Correia, and Ademar Aguiar. 2017. Engineering Software for the Cloud: Messaging Systems and Logging. In *Proceedings of the 22nd European Conference on Pattern Languages of Programs (EuroPlop '17)*. ACM, New York, NY, USA.

M. Tubaishat and S. Madria. 2003. Sensor networks: an overview. *IEEE Potentials* 22, 2 (April 2003), 20–23. DOI:http://dx.doi.org/10.1109/MP.2003.1197877

The Potential of Self-Adaptive Software Systems in Industrial Control Systems

Johannes Iber(✉), Tobias Rauter, Michael Krisper, and Christian Kreiner

Institute of Technical Informatics, Graz University of Technology,
Inffeldgasse 16, Graz, Austria
{johannes.iber,tobias.rauter,michael.krisper,christian.kreiner}@tugraz.at

Abstract. New generations of industrial control systems offer higher performance, are networked and can be controlled remotely. Following this progress, the complexity of such systems increases through heterogeneous systems, hardware and more capable software. This may lead to an increase of unreliability and insecurity. Self-adaptive software systems offer a mean of dealing with complexity by monitoring a control system, detecting anomalies and adapting the control system to problems. Regarding such methods, industrial control systems have the advantage of being less dynamic. The network topology is fixed, devices rarely change, and the functionality of all the resources is known in principle. In this work, we examine this advantage and present the potential of self-adaptive software systems. The context of the presented work is control systems for hydropower units.

1 Introduction

Industrial control systems are computer systems that monitor and control physical processes. Essentially, they are used in critical infrastructures, such as electricity generation and industrial plants. New generations offer higher performance, are networked and can be controlled remotely. Such systems are considered to be part of the broad range of cyber-physical systems.

According to a National Institute of Standards and Technology workshop report [10] the key challenges of cyber-physical systems development include what is needed to cost-effectively and rapidly build in and assure the safety, reliability, availability, security and performance of next-generation systems. Industry is using more and more commercial off-the-shelf hardware platforms, which are inexpensive and offer a high performance. The downside of these platforms is that typically they only offer sparse safety and fault tolerance features [1,6]. Furthermore, industrial control systems are increasingly becoming targets of security attacks [2,8].

Industrial control systems have a big advantage compared to other kinds of systems. The devices used, including hardware and software, network topology, and communication patterns etc. are known and rarely change during operation [2,3]. In our opinion this broad knowledge enables automatic mechanisms that can react e.g. to permanent hardware faults and security attacks. The goal of

J. Stolfa et al. (Eds.): EuroSPI 2017, CCIS 748, pp. 150–161, 2017.
DOI: 10.1007/978-3-319-64218-5_12

such mechanisms would be to increase the reliability of a control system and to defend it against hackers. Self-adaptive software systems offer a means of orchestrating such automatic mechanisms. The underlying idea of self-adaptive software systems is to monitor a managed system and to adapt it in dependence on defined goals. Such a system would not change the architecture of the control systems themselves, but a self-adaptive system would run on top of it instead.

The contribution of this work is an attempt at enumerating this potential of self-adaptive software systems in industrial control systems. We outline the application possibilities in the context of hydropower plants; a domain with which we are familiar. The highlighted application areas of self-adaptive software are hardware faults, security attacks and hacks, software bugs, misconfiguration of the control logic, and faults in the physical environment. We list different anomalies originating from these areas. We present various detection and adaption mechanisms corresponding to the anomalies. It is important to note that we do not wish to change the control logic itself. What we aim to do instead is to strengthen the underlying hardware and software stacks.

The remainder of this paper is structured as follows: the next Section provides a brief introduction to self-adaptive software systems. In Sect. 3 the industrial setting of our work is presented. Subsequently, the different application areas are presented in Sect. 4. In Sect. 5 we discuss the commonalities between the different areas. Finally, concluding remarks are given in Sect. 6.

2 Self-adaptive Software Systems

> *Self-adaptive software modifies its own behavior in response to changes in its operating environment. By operating environment, we mean anything observable by the software system, such as end-user input, external hardware devices and sensors, or program instrumentation* [11].

Typically, self-adaptive software systems follow an external (architecture) approach [13]. An internal approach interweaves application and adaption logic based e.g. on programming language features such as exceptions, conditions, and parametrization. The issue with an internal approach is that sensors, actuators, parallel adaption processes and purpose of an application are complicated to engineer within one software design. This leads to notable drawbacks, e.g. with respect to scalability, testability and maintainability.

In an external approach, as illustrated in Fig. 1, the domain-specific application logic termed *Managed Subsystem* is monitored by a *Managing Subsystem*. The *Managing Subsystem* is where the actual adaption logic resides. It additionally monitors the *Environment* that may consist of other software, hardware, network, or of the physical context including humans. In the presented setting, a *Managing Subsystem* observes anomalies of a control system (*Managed Subsystem*). By applying different detection mechanisms a *Managing Subsystem* is supposed to analyze the root cause of the observed anomalies. In the final step it chooses an appropriate adaption mechanism for circumventing the problem.

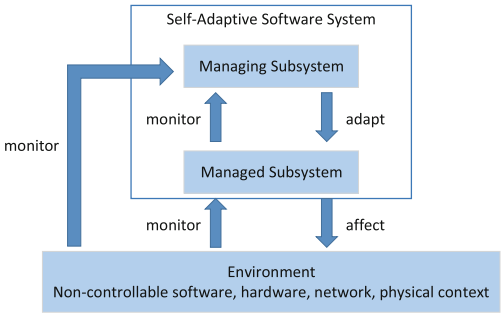


Fig. 1. Parts of a self-adaptive software system (adapted from [14])

Typically, the *Managing Subsystem* itself comprises an adaptive loop that coordinates different detection mechanisms and adapts accordingly. Muccini et al. [9] reveal in a systematic literature review that in the context of a cyber-physical system, the so-called MAPE-K loop (**M**onitor, **A**nalyze, **P**lan, **E**xecute - **K**nowledge) is by far the dominant adaptive loop in science with a share of 60%.

3 Industrial Setting

The industrial setting of the following application areas are networked control devices that operate hydropower plant units. We choose this setting because it is also our project context and we are familiar with it. Figure 2 illustrates a simplified overview of such an industrial control system.

On network level, control devices are connected via Ethernet and operated by supervisory computers. These supervisory computers are responsible for observing the state of physical processes and adjusting parameters of control devices in order to control the energy conversions. The observation and adjustment actions are done by using so-called datapoints which are variables with a specific basic data type such as integer or boolean.

The control devices are connected to hydropower plant units. Their functional responsibility is to operate these units through one of the four different functions namely excitation, synchronization, protection and turbine control.

Technically, these devices have a programmable logic controller (PLC) architecture. In the context of the hardware design, a control device is built out of central modules and interface modules. A central module consists of a communication CPU (CCPU) and an application CPU (ACPU). The CCPU is responsible for network connections and controlling/monitoring the ACPU. It runs a customized Linux distribution and can be accessed by various protocols such as SSH and Modbus. From the security perspective this protects the ACPU and verifies incoming commands. The ACPU is a multi-core processor and executes the actual control logic. It runs a real-time operating system in order to ensure guaranteed cycle times. The interface modules connecting the control device with

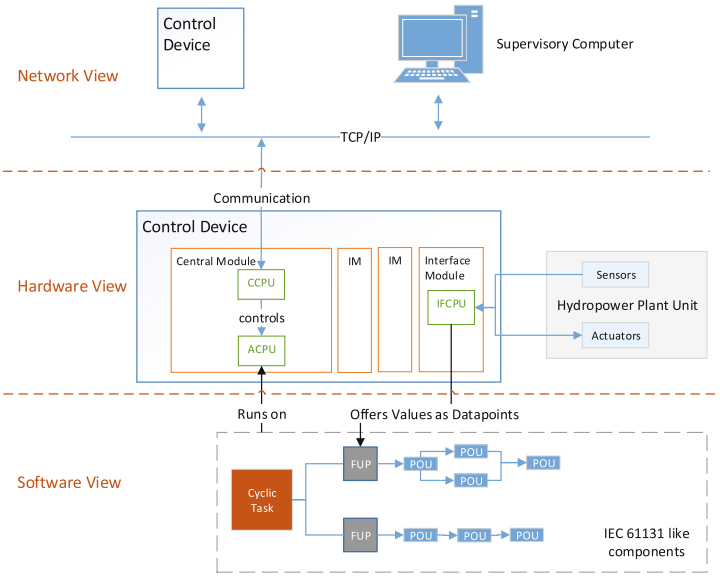


Fig. 2. Overview of the industrial control system

the sensors and actuators of the hydropower plant unit. Central modules and interface modules are connected via Ethernet.

The control logic executed by the ACPU of a central module is component-based and heavily influenced by the IEC 61131 standard for programmable logic controllers [7]. It is hierarchically built out of components, compositions and tasks. Components are termed Program Organization Units (POU) and compositions are named Function Plans (FUP). POUs are coded with the C-programming language and stored as binaries on the devices. Such POUs implement basic functions, e.g. simple logic gates, or complex algorithms. Based on these POUs, reusable FUPs are designed by plant engineers that implement the specific control logic for a hydropower plant unit. Finally, such FUPs are executed by cyclic tasks in real-time.

FUPs operate on datapoints that are set and read by the interface modules. At the start of a cyclic task the necessary datapoints are collected, then the FUPs are executed, and subsequently the calculated datapoints are written back. The interface modules receive these datapoints and actuate accordingly. Further, datapoints can be shared with other control devices or supervisory computers.

The supervisory computers are themselves part of a control hierarchy. Higher hierarchy levels control for instance different hydropower plants along a river.

4 Application Areas

Figure 3 illustrates a network of control devices, a supervisory computer and a hydropower plant unit containing sensors and actuators. The arrows from the left and right side represent areas we are proposing to tackle with detection and adaption mechanisms. In the issue of hardware faults, we are targeting those faults located inside a control device. The hydropower unit is out of scope. In the context of security attacks, our interest is to detect hacks of sensors, actuators and control devices. Unknown software bugs can be added through updating the software or occur if an untested state arises. The area misconfiguration relates to the case of a human operator making a mistake in the design of the control logic. *Faults in the physical environment* relates to the sensors and actuators that form a part of the hydropower unit. We discuss such anomalies from the point of view of a control device. In the following, we examine these five areas on the control device and network level. We present the observable anomalies together with corresponding detection and adaption mechanisms. Note that the presented anomalies, detection and adaption mechanisms are not complete and are imaginable concerning hydropower units. They may not be appropriate for all kinds of domains.

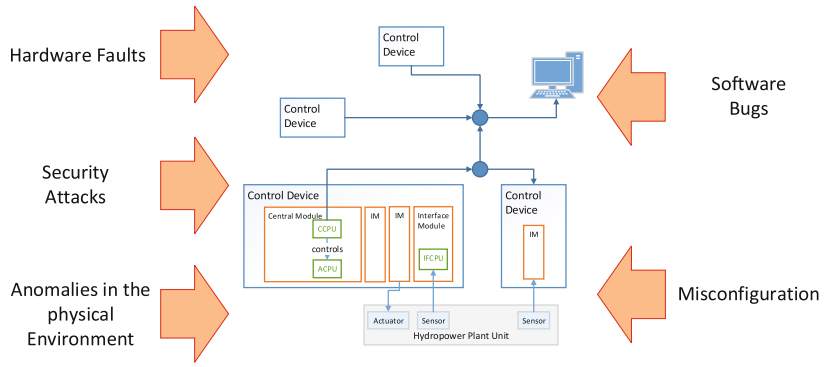
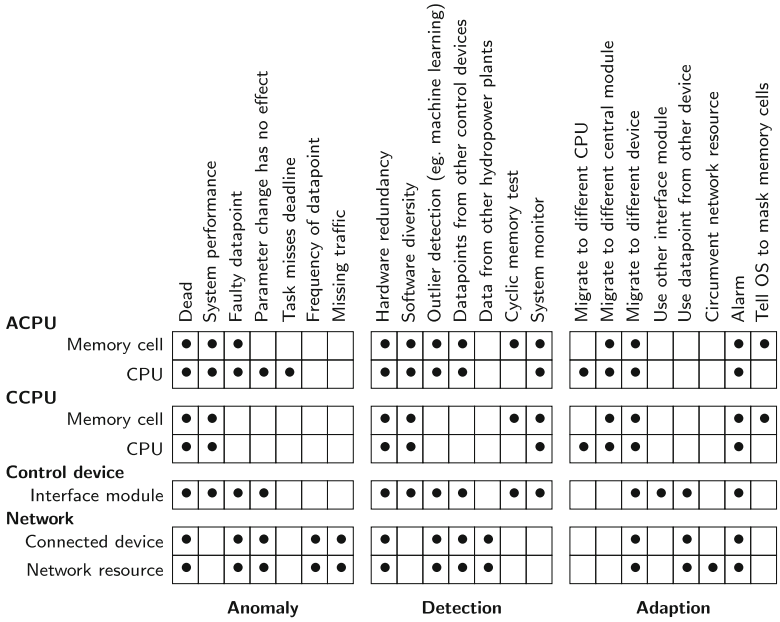


Fig. 3. Overview of the presented application areas of self-adaptive software in hydropower control systems

4.1 Hardware Faults

In this area we focus on hardware faults located in central and interface modules part of a control device. Table 1 illustrates an overview of where a fault can be located, what anomalies can occur originating from a hardware fault, how hardware faults can be detected based on the anomalies, and what can be done concerning adaption. The left hand side of Table 1 shows the location. *ACPU* is part of the central module and the place where the actual control logic resides. The *CCPU* is responsible for the communication with the network and controls

Table 1. Different locations where a permanent hardware fault can reside with corresponding anomalies, and possible detection and adaption mechanisms.



the *ACPU*. Further, the interface modules are connected with the sensors and actuators of a hydropower unit. They transmit their measured values as datapoints to the *ACPU* and receive parameters as datapoints. The network part is presented from a control device perspective. *Connected device* represents a control device that sends or receives datapoints. *Network resource* can be any other kind of devices in a network such as a switch or a supervisory computer.

The anomalies in the first set of columns may occur because of a fault residing in the locations on the left hand side. *Dead* means that it is not responding on interaction attempts. *System performance* describes the capability of a system in how it executes a task. It consists of CPU load, memory consumption and latency. The anomaly *Faulty datapoint* refers to a datapoint that is wrong. *Parameter change has no effect* describes the situation that a task or supervisory computer wants to change an actuator through a datapoint, but the sensed datapoints do not indicate that anything is behaving differently. *Task misses deadline* refers to the situation that a task needs to finish its work before the cycle starts again. *Frequency of datapoint* describes the distribution of datapoints within a network. *Missing traffic* is an anomaly concerning the absence of distributed datapoints and communication.

The second set of columns refers to detection mechanisms that use such anomalies as evidence for a hardware fault. *Hardware redundancy* is about duplication of components or devices in order to increase the reliability. A voter can

then detect discrepancies. *Software diversity* refers to the idea of realizing a software functionality in two or more distinct ways. Diversity can be achieved by developing the functionality of a software several times in different ways by using independent development teams and technologies. Another approach is to compile software with different settings [5]. Since diverse software is working in other ways, there is a chance that permanent hardware faults can be detected in a manner similar to *Hardware redundancy*. *Outlier Detection* refers to mechanisms that detect patterns that do not correspond to an expected behavior. One such a technique is machine learning. A simpler one would be utilizing certain thresholds of datapoint values. *Datapoints from other control devices* can be used for redundancy and plausibility checks. *Data from other hydropower plants* can be used for a plausibility check of the sensed data concerning the river and how the hydropower units are behaving. *Cyclic memory test* is about checking the memory regularly and detecting permanent faults manifesting in memory cells. A *System monitor* keeps watch on a system regarding CPU usage, temperature, available memory or network traffic. A change of one of the watched parameters can indicate a hardware fault.

The third set of columns shows possible adaption mechanisms. These mechanisms can be applied after a hardware fault with the corresponding location is detected. In general these mechanisms assume the existence of some redundant hardware, otherwise it would be difficult to circumvent permanent hardware faults.

Migrate to different CPU means to use an alternative ACPU or CCPU. This adaption mechanism assumes that either a CPU is available on a module, or that a standby central module is plugged into the control device. *Migrate to different central module* is similar to the adaption mechanism described above. It assumes that another central module is available for migrating the ACPU and CCPU. *Migrate to different device* is a costlier process where a redundant device is selected for taking over the control activities. *Use other interface module* assumes that an interface module offering the same datapoints is available. This is not restricted to one device. In general, it is possible to transmit datapoints, e.g. a datapoint for voltage, from one device to another. *Circumvent network resource* depends on the affected network device. A switch can be circumvented if other paths exist. A supervisory computer can be replaced with a redundant one. *Alarm* is a general mechanism that notifies a human operator who can change parts of the control system. *Tell OS to mask memory cells* leverages the capabilities of operating systems to blacklist faulty memory areas.

4.2 Security Attacks and Hacks

Industrial control systems usually behave in a foreseeable and deterministic manner. In principal, each control device knows from and to whom it receives or sends data. Furthermore, the used software running on top of a devices is known. An attacker would have to introduce a different behavior or software in order to harm a control system.

Table 2. Different locations that can be hacked and the corresponding anomalies, detection and adaption mechanisms. The hydropower unit is also considered.

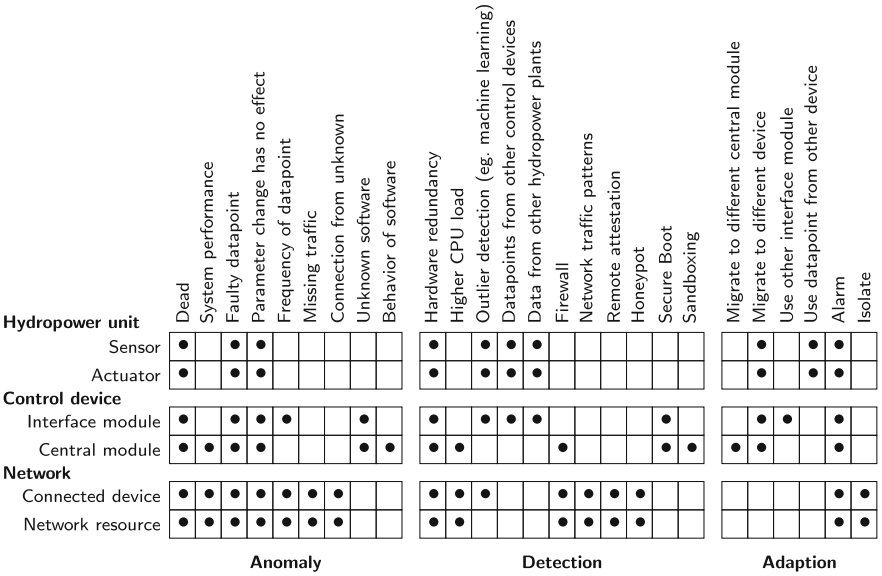
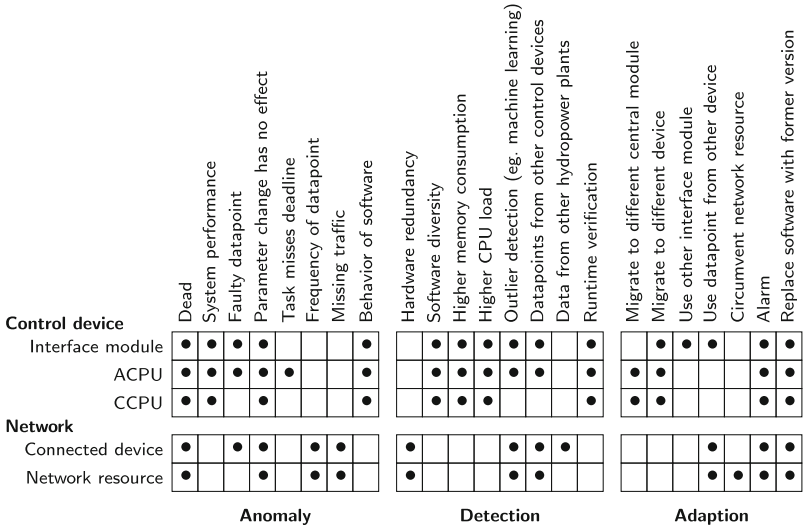


Table 2 shows different parts that could be hacked. An attacker could manipulate sensors and actuators which may lead to faulty datapoints or parameters with no effect. Such a situation can be detected with mechanisms such as those pointed out in Subsect. 4.1. At control device level, an interface or central module could be hacked. The anomalies *Unknown software* and *Behavior of software* are related to the detection mechanisms *Secure Boot* and *Sandboxing*. *Secure Boot* ensures that only trusted software is started. *Sandboxing* restricts the environment of a software and mitigates the impact of e.g. a buffer overflow. On network level, the traffic between devices plays a key role of identifying security breaches in addition to the states and effects of datapoints. The detection mechanism *Network traffic patterns* leverages the fact that the traffic in an industrial control system follows cycles and deterministic behavior [4]. *Remote Attestation* can be used to prove the integrity of one control device to another [12]. Using this method a hacked control device would fail to prove that its integrity has not been violated. A *Honeypot* is a bait for attackers that behaves like a control device in the network. Any connection attempt to a honeypot would indicate a security problem. The illustrated adaption mechanisms are known from above. The adaption mechanism *Isolate* stands for circumventing and blocking an infected device.

Table 3. Locations where anomalies of software bugs can be detected and adapted.



4.3 Software Bugs

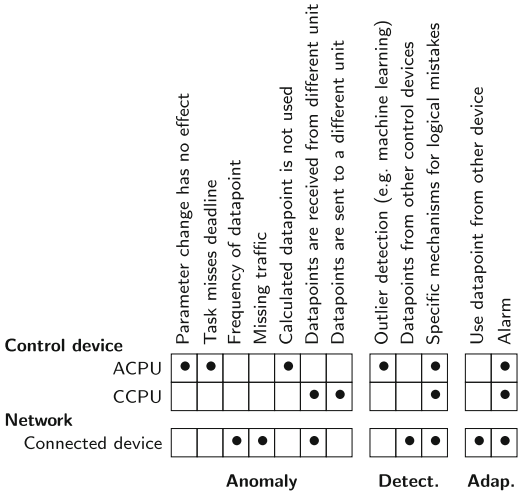
In this area, our target is to detect and adapt to software bugs (Table 3). There are two kinds of bugs that we consider. One kind are bugs that are introduced after an update. The other kind is already present in the software but lurks around until an untested case or state occurs.

The detection method *Hardware redundancy* on network level assumes that not all software is updated at once or that the hardware uses different means of obtaining data. *Runtime Verification* corresponds directly to the anomaly *Behavior of software*. It is about analyzing logs and interactions of software in order to detect new behavior. The adaption mechanism *Replace software with former version* is a rollback and assumes that a former version does not contain the detected software bug.

4.4 Misconfiguration of the Control Logic

Industrial control systems are configured by human operators. This opens the possibility of logical mistakes. Control devices and other devices are usually observed for a period of time in order to ensure that they behave as expected. Here, detection mechanism can accelerate test times. Furthermore, it may happen that single control devices are replaced and parts of a plant configuration change. Table 4 shows anomalies originating from misconfiguration. The last three anomalies are solely traceable to configuration mistakes. *Specific mechanisms for logical mistakes* stands for detection mechanisms that are domain specific for common misconfigurations. Concerning adaption mechanisms there

Table 4. Locations where anomalies of misconfiguration can be detected and adapted.

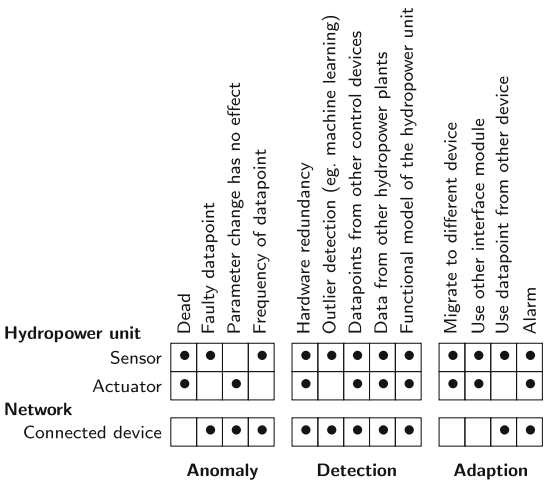


are not many strategies for resolving misconfigurations. Configuring a control device is a creative task. Automatically creating control logic in a safe manner requires a deep understanding of all kinds of devices.

4.5 Faults in the Physical Environment

Control devices in our setting are not directly connected with a water turbine. The link is mediated by sensors and actuators and these can break or drift over

Table 5. Faults in the physical environment reside in sensors and actuators between a control device and the turbine.



time. Detecting and adapting to such problems can again make a control system more reliable.

Table 5 shows anomalies that can originate from sensors and actuators. Furthermore, such anomalies can be observed through a connected device if it merely forwards them. The detection mechanisms presented are similar to the ones above. What we added is to leverage a *Functional model of the hydropower unit*. The idea behind such a model is to verify the change of parameters with the expected outcome. The proposed adaption mechanisms are similar to the ones above.

5 Discussion

As we can see from the different tables, there are several shared anomalies, detection and adaption mechanisms. Some of these presented detection and adaption mechanism rely on having redundant or stand-by hardware. One could argue that the different mechanisms can be implemented separately and focused on only one problem area. The downside of keeping them strictly separate is that they could interfere with each other or reach wrong conclusions about the real cause of a problem. A self-adaptive software system offers means to deal with cross-cutting anomalies by orchestrating different mechanisms. It would have to reason about the combined outcome of several detection mechanisms in order to find the affected area containing the real cause. Based on the cause, a suitable adaption mechanism would have to be selected. It is important to note that adaption mechanisms do need some execution time. This may introduce delays into control processes. The problem of how bad a small delay really is depends on the domain. E.g. in the hydropower plant context one could argue that the benefit of fixing a permanent hardware fault, but at the cost of introducing a small delay, is preferable to a broken or faulty control activity. Furthermore, testing an adaption before it is carried out is a complicated procedure. This means a self-adaptive system must have a precise representation of the real system in order to ensure that an adaption works as intended. All in all, it is our considered opinion that the presented detection and adaption possibilities would add a valuable improvement to industrial control systems by making them meta-adaptive. A system incorporating some or all of these features has the potential of being more reliable and secure than state of the art control systems.

6 Conclusion

In this work, we presented five areas where a self-adaptive system could improve current and future generations of industrial control systems. The highlighted application areas are hardware faults, security attacks and hacks, software bugs, misconfiguration of the control logic and faults in the physical environment. Self-adaptive software systems are part of our ongoing research towards increasing the reliability and security of control system for hydropower units. The presented anomalies, detection and adaption mechanisms are real possibilities to be faced in

our industrial setting, but may differ or appear to be different in other domains. Despite this we believe that most of our observations are readily transferable to other industrial domains.

References

1. Alhakeem, M.S., Munk, P., Lisicki, R., Parzyjegla, H., Parzyjegla, H., Muehl, G.: A framework for adaptive software-based reliability in COTS many-core processors. In: ARCS 2015 (2015)
2. Cárdenas, A.A., Amin, S., Lin, Z.S., Huang, Y.L., Huang, C.Y., Sastry, S.: Attacks against process control systems. In: ASIACCS 2011. ACM Press (2011)
3. Cheung, S., Dutertre, B., Fong, M., Lindqvist, U., Skinner, K., Valdes, A.: Using model-based intrusion detection for SCADA networks. In: Proceedings of the SCADA Security Scientific Symposium (2007)
4. Hadeli, H., Schierholz, R., Braendle, M., Tuduce, C.: Leveraging determinism in industrial control systems for advanced anomaly detection and reliable security configuration. In: Conference on Emerging Technologies & Factory Automation. IEEE (2009)
5. Höller, A., Rauter, T., Iber, J., Kreiner, C.: Patterns for automated software diversity to support security and reliability. In: EuroPLoP 2015. ACM (2015)
6. Höller, A., Spitzer, B., Rauter, T., Iber, J., Kreiner, C.: Diverse compiling for software-based recovery of permanent faults in COTS processors. In: DSN-W 2016. IEEE (2016)
7. John, K.H., Tiegelkamp, M.: IEC 61131–3: Programming Industrial Automation Systems. Springer, Heidelberg (2010)
8. Miller, B., Rowe, D.: A survey SCADA of and critical infrastructure incidents. In: RIIT 2012. ACM Press (2012)
9. Muccini, H., Sharaf, M., Weyns, D.: Self-adaptation for cyber-physical systems: a systematic literature review. In: SEAMS. ACM Press (2016)
10. NIST: Foundations for Innovation in Cyber-Physical Systems. Technical report (2013)
11. Oreizy, P., Gorlick, M., Taylor, R., Heimhigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D., Wolf, A.: An architecture-based approach to self-adaptive software. IEEE Intell. Syst. **14**(3), 54–62 (1999)
12. Rauter, T., Höller, A., Iber, J., Kreiner, C.: Thingtegrity: a scalable trusted computing architecture for the internet of things. In: EWSN 2016. Junction Publishing (2016)
13. Salehie, M., Tahvildari, L.: Self-adaptive software: landscape and research challenges. ACM Trans. Auton. Adapt. Syst. **4**(2), 14 (2009)
14. Weyns, D., Schmerl, B., Grassi, V., Malek, S., Mirandola, R., Prehofer, C., Wuttke, J., Andersson, J., Giese, H., Göschka, K.M.: On patterns for decentralized control in self-adaptive systems. In: Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II. LNCS, vol. 7475, pp. 76–107. Springer, Heidelberg (2013). doi:10.1007/978-3-642-35813-5_4

Solutions for Cyber-Physical Systems Ubiquity

Norbert Druml
Independent Researcher, Austria

Andreas Genser
Independent Researcher, Austria

Armin Krieg
Independent Researcher, Austria

Manuel Menghin
Independent Researcher, Austria

Andrea Hoeller
Independent Researcher, Austria

A volume in the Advances in Systems Analysis, Software Engineering, and High Performance Computing (ASASEHPC) Book Series

Published in the United States of America by
IGI Global
Engineering Science Reference (an imprint of IGI Global)
701 E. Chocolate Avenue
Hershey PA, USA 17033
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@igi-global.com
Web site: http://www.igi-global.com

Library of Congress Cataloging-in-Publication Data

Names: Druml, Norbert, 1980- editor.
Title: Solutions for cyber-physical systems ubiquity / Norbert Druml, Andreas Genser, Armin Krieg, Manuel Menghin, and Andrea Hoeller, editors.
Description: Hershey, PA : Engineering Science Reference, [2018] | Includes bibliographical references.
Identifiers: LCCN 2017012032| ISBN 9781522528456 (hardcover) | ISBN 9781522528463 (ebook)
Subjects: LCSH: Cooperating objects (Computer systems)--Handbooks, manuals, etc. | Internet of things--Handbooks, manuals, etc. | Automatic control--Handbooks, manuals, etc.
Classification: LCC TK5105.8857 .H367 2018 | DDC 004.67/8--dc23 LC record available at https://lccn.loc.gov/2017012032

This book is published in the IGI Global book series Advances in Systems Analysis, Software Engineering, and High Performance Computing (ASASEHPC) (ISSN: 2327-3453; eISSN: 2327-3461)

British Cataloguing in Publication Data
A Cataloguing in Publication record for this book is available from the British Library.

Chapter 9
A Self-Adaptive Software System for Increasing the Reliability and Security of Cyber-Physical Systems

Johannes Iber
Graz University of Technology, Austria

Tobias Rauter
Graz University of Technology, Austria

Christian Kreiner
Graz University of Technology, Austria

ABSTRACT

The advancement and interlinking of cyber-physical systems offer vast new opportunities for industry. The fundamental threat to this progress is the inherent increase of complexity through heterogeneous systems, software, and hardware that leads to fragility and unreliability. Systems cannot only become more unreliable, modern industrial control systems also have to face hostile security attacks that take advantage of unintended vulnerabilities overseen during development and deployment. Self-adaptive software systems offer means of dealing with complexity by observing systems externally. In this chapter the authors present their ongoing research on an approach that applies a self-adaptive software system in order to increase the reliability and security of control devices for hydro-power plant units. The applicability of the approach is demonstrated by two use cases. Further, the chapter gives an introduction to the field of self-adaptive software systems and raises research challenges in the context of cyber-physical systems.

INTRODUCTION

Cyber-physical systems (CPS) are the next-generation of systems that integrate computational and physical components. In contrary to the embedded devices of the last decades, they offer high performance, are interconnected and, with a good chance, somehow connected with the internet. Following

DOI: 10.4018/978-1-5225-2845-6.ch009

this trend, control devices typically found in industry are going to manage more and more functionality with the help of sophisticated software. According to a National Institute of Standards and Technology workshop report (NIST, 2013) the key challenges of CPS development include what is needed to cost-effectively and rapidly build in and assure the safety, reliability, availability, security and performance of next-generation CPS. Industry is using more and more commercial off-the-shelf hardware platforms, which are inexpensive and offer a high performance. The downside of these platforms is that they typically only sparsely offer safety and fault tolerance features (Alhakeem et al., 2015). Further, industrial cyber-physical systems are becoming increasingly targets of security attacks (Miller & Rowe, 2012).

The inherent problem of CPS is complexity. This issue is going to escalate as CPS become large-scale distributed systems. They have to deal with uncertainty, change during operation, be scalable and tolerant to threats (Muccini, Sharaf, & Weyns, 2016). Self-adaptive software systems are systems that target to deal with complexity. Typically, self-adaptive software systems externally observe their managed systems, detect problems and adapt the managed systems in order to repair or circumvent inconsistencies. In the case of security, a self-adaptive software system can detect security attacks and isolate the infected devices or block the attackers. In the case of hardware faults, a self-adaptive software system can detect permanent hardware faults and move the application logic running on a managed system to an alternative hardware. Such problems would be complicated for a managed system itself to circumvent, but through an external overlooking system this becomes possible and the lurking complexity of CPS may become manageable.

Because of the increased performance and connectivity of modern and future hardware, self-adaptive software systems can be deployed to former restricted devices found in industry. In this chapter, ongoing research of an approach is presented that provides a novel application of a self-adaptive software system in an industrial setting, namely control devices for hydro-power plant units which is also the context of our research project. The goal of the presented approach is to increase the reliability and security of systems through anomaly detection and adaption. Simply put, we want to extend the time of control systems as long as possible so that they can carry out undisturbed their intended purposes.

The following Sections are structured as follows: First we give a detailed overview of the underlying principles of self-adaptive software systems. After that we extensive present our approach named Scari (**Sec**ure **a**nd **r**eliable **i**nfrastructure); including an overview of our industrial setting (hydro-power plants), the vision of Scari, the detailed approach and two use cases. In the subsequent Section, we present a number of research challenges that we derive from our self-adaptive software system. Last, we conclude this Chapter.

BACKGROUND

Self-adaptive software modifies its own behavior in response to changes in its operating environment. By operating environment, we mean anything observable by the software system, such as end-user input, external hardware devices and sensors, or program instrumentation. (Oreizy et al., 1999)

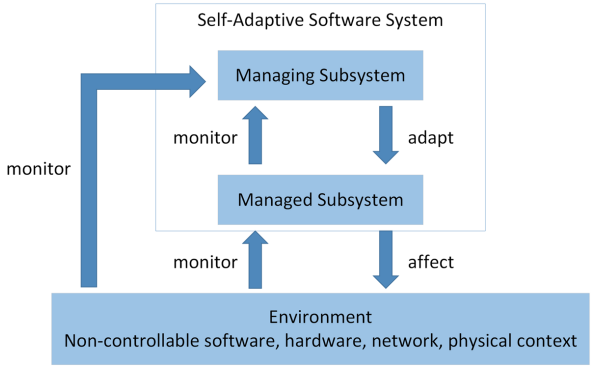
Historically, the intention of building self-adaptive software systems has been around some time. Though not being the first talking and writing about self-adaptive software systems but making significant investments, IBM introduced in 2001 the Autonomic Computing Initiative in response to their observation that *the main obstacle to further progress in the IT industry is a looming software complexity crisis* (Kephart

& Chess, 2003). They argued that systems become too interconnected, too diverse and complex for even the most skilled system integrators to install, configure, optimize, maintain, and merge. Back then, *IBM researchers predicted that by the end of the decade the IT industry would need up to 200 million workers, equivalent to the entire US labor force, to manage a billion people and millions of businesses using a trillion devices connected via the Internet* (Dobson, Sterritt, Nixon, & Hinchey, 2010). Based on this idea of the future, they envisioned the need to develop computer systems that can manage themselves given high-level objectives. Since then, the term autonomic computing has emerged into a broader context related with organic computing, bio-inspired computing, self-organizing systems, ultrastable computing, and adaptive systems, to name a few (Dobson et al., 2010). As pointed out by Salehie & Tahvildari (2009), the term self-adaptive software system is focused on the domain of software systems. In the following we only use this term instead of autonomic computing or others as it narrows the scope. The fundamental reason for applying self-adaptive software systems is the increasing cost of handling the complexity of software systems to achieve their goals (Laddaga, 2001; Salehie & Tahvildari, 2009).

Typically, self-adaptive software systems follow an external (architecture) approach (Salehie & Tahvildari, 2009). An internal approach interweaves application and adaption logic based on programming language features like exceptions, conditions, and parametrization. The issue with an internal self-adaptive software system is that sensors, actuators, parallel adaption processes and actual purpose of an application are complicated to engineer within one software design. This further leads to notable drawbacks, e.g. with respect to scalability, testability and maintainability. In an external approach, as illustrated in Figure 1, the domain-specific application logic named *Managed Subsystem* is monitored by a *Managing Subsystem*. The *Managing Subsystem* is where the actual adaption logic resides. It additionally monitors the Environment that may consist of other software, hardware, network, or of the physical context (including humans). Based on monitored data and analyzed problems the *Managing Subsystem* decides whether and what to adapt inside the *Managed Subsystem*.

In order to show what a self-adaptive system actually means and what such a system desires we discuss in the next Subsection a hierarchy of self-properties. For the process of collecting data about the environment and *Managed Subsystem*, and finally adapting the *Managed Subsystem*, usually closed-loop

Figure 1. Parts of a self-adaptive software system
Based on Weyns et al., 2013.



mechanisms are implemented inside the *Managing Subsystem*. We present three so-called adaption loops after the self-properties. These adaption loops represent the essence of architectures that aim to fulfill one or several of the presented self-properties. In the last part of this Section, we present the concept of utilizing models based on the Model-Driven Engineering principles in order to support self-adaptive software systems. This concept is named by its community Models@Run.time.

Hierarchy of Self-Properties

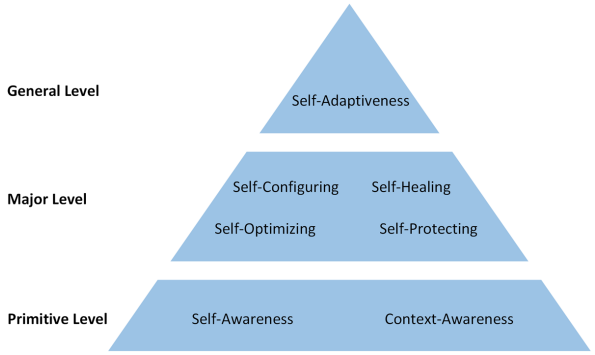
Self-Adaptiveness is a very broad term and represents the sum of several self-properties. Salehie & Tahvildari (2009) discuss these properties in detail and represent them in the hierarchy illustrated in Figure 2.

The top level named *General Level* contains global properties of self-adaptive systems. Terms, found in literature, which are basically a subset of self-adaptiveness are *self-managing, self-governing, self-maintenance, self-control, self-evaluating, and self-organizing*.

The *Major Level* terms are coined by the IBM Autonomic Computing Initiative and serve as the defacto standard in self-adaptive systems (Salehie & Tahvildari, 2009). The following four properties are generally also known as self-CHOP (M. Hinchey & Sterritt, 2006):

- **Self-Configuring:** A system reconfigures itself automatically in response to changes following high-level policies.
- **Self-Healing:** A system automatically detects, diagnoses, and reacts to software and hardware failures by healing itself.
- **Self-Optimizing:** A system continually seeks opportunities to improve its own performance and resource allocation.
- **Self-Protecting:** This property has two aspects. One is that a system automatically defends itself against malicious attacks or cascading failures. The other one is that it mitigates the effects of attacks.

Figure 2. Hierarchy of self-properties
Based on Salehie & Tahvildari, 2009.



The *Primitive Level* represents the base of the *Major Level* and consists of two properties. Without them a self-adaptive system would not be able to realize the properties from the *Major Level*:

- **Self-Awareness:** A system is aware of its own states and behaviors.
- **Context-Awareness:** A system is aware of its operational environment.

All these properties above are defining what self-adaptive systems are targeting to achive. In our approach, we are aiming for all *Primitive Level* and *Major Level* properties except self-optimization. The vision of the proposed system is to autonomously configure, heal, and protect itself based on its own state and the operational environment.

Adaption Loops

There exist different variants of how a self-adaptive software system can be organized. Muccini et al. (2016) reveal in a systematic literature review that concerning CPS, the so-called MAPE-K loop (later explained in detail) is by far the dominant adaptive mechanism with a share of 60%. It is followed by multi-agent and self-organization based technologies (both have 29% - some studies combine technologies). Multi-agent systems are large-scale open decentralized systems that consist of autonomous components or systems (Müller & Fischer, 2014) that work together for achieving a common goal. Self-organization techniques are inspired by nature where behavior emerges e.g. from cells (Jelasity et al., 2006). Multi-agent systems and self-organization techniques are out-of-scope of this work as they do not fit our industrial setting.

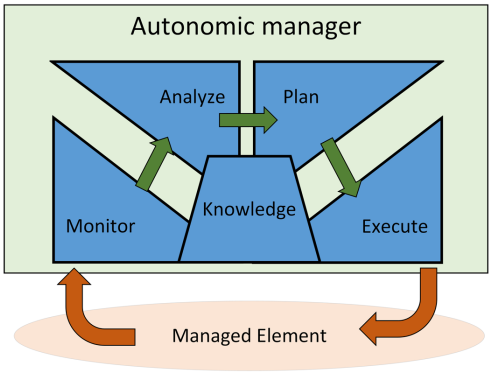
In the following we present three adaption loops that try to grasp the necessary steps and activities of a self-adaptive software system. After that, we shortly discuss the commonalities between them and highlight five patterns of how several loops can be organized.

MAPE-K

Figure 3 illustrates the MAPE-K adaptive loop, introduced by Kephart & Chess (2003) defining IBMs autonomic computing vision. It consists of the steps *Monitor, Analyze, Plan, and Execute*, and a shared part representing *Knowledge*. The target of MAPE-K is the *Managed Element* which is monitored with sensors and changed with actuators. The *Monitor* step gathers information about the *Managed Element* that is usually related to the current performance and load of the system (Brun et al., 2013). The *Analyze* step reasons about the data, identifies problems and attempts to find the source or cause of them. The *Plan* step reacts to the results of the *Analyze* step and creates a set of actions to remedy a problem. The last step, named *Execute*, implements these actions and changes the *Managed Element* through actuators. *Knowledge* is the central point where all the information within a MAPE-K loop comes together. The *Monitor* stores its observed data at this point. The *Analyze* step uses it to find anomalies. The *Plan* step leverages it to create actions and gathers its policies and goals from there. Finally, the *Execute* step stores its record of executed actions in it.

As we can see on Figure 3 there is an *Autonomic Manager* around the loop. That is basically an interface for controlling and monitoring the adaptive system. Kephart & Chess (2003) foresaw a plethora of *Autonomic Manager* each managing for instance a hardware resource (e.g. CPU, printer, storage) or software resource (e.g. database, service, legacy system).

Figure 3. MAPE-K
Based on Kephart & Chess, 2003.



OODA

Colonel John Boyd was a United States Air Force fighter pilot and Pentagon consultant who developed the first version of his OODA-loop for explaining how to achieve success in air-to-air combat in the 1950's. Later he expanded his groundbreaking work and hypothesized that it is the essence of winning and losing of organizations and people (Boyd, 1996). As pointed out by other authors it lends itself well to self-adaptive system (Chandra, Lewis, Glette, & Stilkerich, 2016; Grenander, Simpson, & Sindiy, 2009). In the OODA-loop, *Observe* means to gather, monitor, and filter data. Figure 4 illustrates the type of data that can be observed; implicit guidance and control refers to the significant influence of the *Orientation* step. In the *Orient* step a list of options is derived through analysis and synthesis, previous experience,

Figure 4. OODA-loop
Source: Wikimedia Commons, 2014.

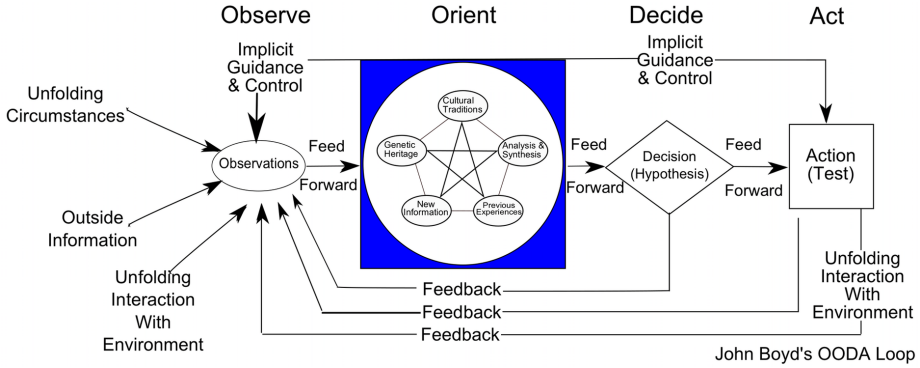
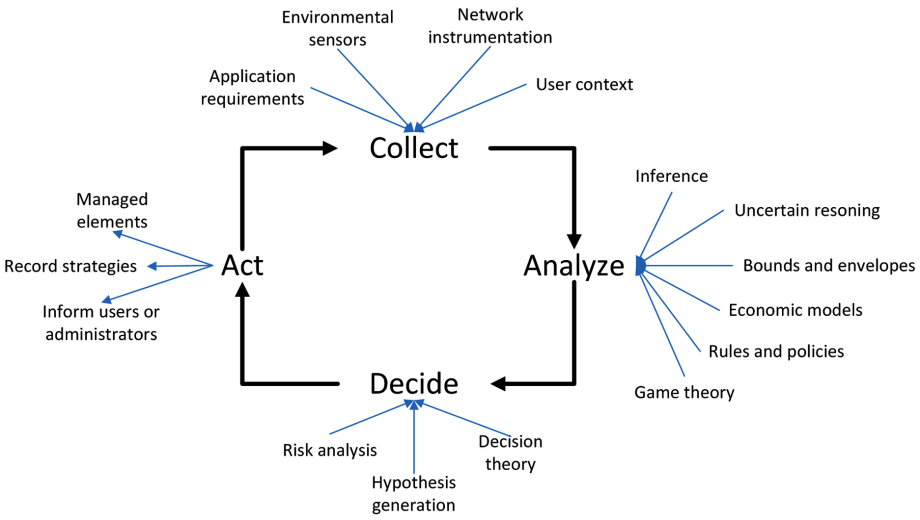


Figure 5. CADA-loop
Based on Dobson et al. (2006).



new information, and of course as the loop is intended for humans, genetic and cultural heritage. The derived list of options is then feed forwarded to the *Decide* step where the best hypothesis is selected via a ranking. In the last step, the selected option is acted out and in a way tested in the environment. As pointed out by John Boyd, *orientation shapes observation, shapes decision, shapes action, and in turn is shaped by the feedback and other phenomena* (Boyd, 1996). He demonstrated, that in the direct combat it is crucial to go through this loop faster and better than an opponent. Further, he noted that the entire loop (not just orientation) is an ongoing many-sided implicit cross-referencing process of projection, empathy, correlation, and rejection. We illustrate in our approach how we transfer the OODA-loop to our variant of an adaption loop.

CADA

Dobson et al. (2006) describe the generic *Collect - Analyze - Decide - Act* loop for autonomic communication systems. The field of autonomic communication targets to improve the ability of networks and services to cope with unpredicted changes like topology, load, task and so on. As we can see in Figure 5 it is similar to MAPE-K and the OODA-loop, but way more generic. In the *Collect* activity data is gathered from several sources, in the *Analyze* activity analyzed, then a *Decision* is made, and finally acted out in the *Act* activity. The loop is annotated with several techniques and approaches which can be applied for implementing the single activities. As mentioned by Cheng et al. (2009), reasoning in self-adaptive systems typically involve these four activities.

Discussion of the Adaption Loops

Essentially the three presented loops are variants of the same idea, which is to have a chain of activities that lead to an appropriate response to a problem of the managed system. However, the loops vary concerning the different steps and feedback. MAPE-K introduces the *Knowledge* part as common information source for each activity taking place. OODA emphasizes that the different steps give feedback to what is observed and that the adaptive loop is essentially driven by the *Orient* phase. The *Orient* phase of OODA corresponds to the *Analyze* and *Plan* steps of the MAPE-K loop. OODA introduces an explicit separate *Decide* phase that is embedded into the *Plan* step of MAPE-K. The CADA-loop is a generic version of an adaptive loop. We include it because it highlights the different technologies which can be applied in each step.

Usually, self-adaptive software does not consist of only one adaptive loop in the whole, possibly distributed, system. Such systems incorporate multiple loops connected or running in parallel. Weyns et al. (2013) gathered five patterns for decentralized control in self-adaptive systems that describe how MAPE loops can be related to each other. We describe the essence of these patterns in the following shortly:

- **Coordinated Control Pattern:** Consider a distributed system where each node owns an own MAPE loop. This pattern proposes that all the *Monitor, Analyze, Plan*, and *Execute* steps coordinate their operation with corresponding peers of other loops. For instance, *Analyze* entities interact with each other to make a decision about the need for an adaption.
- **Information Sharing Pattern:** In this pattern, all *Monitors* in a distributed system are sharing their observed states with each other, while *Analyze, Plan* and *Execute* entities are acting independently from their counterparts on other nodes.
- **Master/Slave Pattern:** There exists a central master component that is responsible for the *Analyze* and *Plan* step of adaptions. The other nodes in such a system are responsible for monitoring states and executing actions.
- **Regional Planning Pattern:** Such a distributed system is partitioned into regions where in each region a central component performs the *Plan* step. The *Monitor, Analyze,* and *Execute* steps are deployed on other nodes. The central *Plan* components can be connected with each other.
- **Hierarchical Control Pattern:** This pattern organizes MAPE loops in hierarchies. For instance, a loop is in control of a node. If it cannot adapt, a situation can be escalated to a higher loop that possesses a broader control of the target distributed system.

Another important design decision concerning self-adaptive systems is whether a control loop is realized event-driven or time-driven.

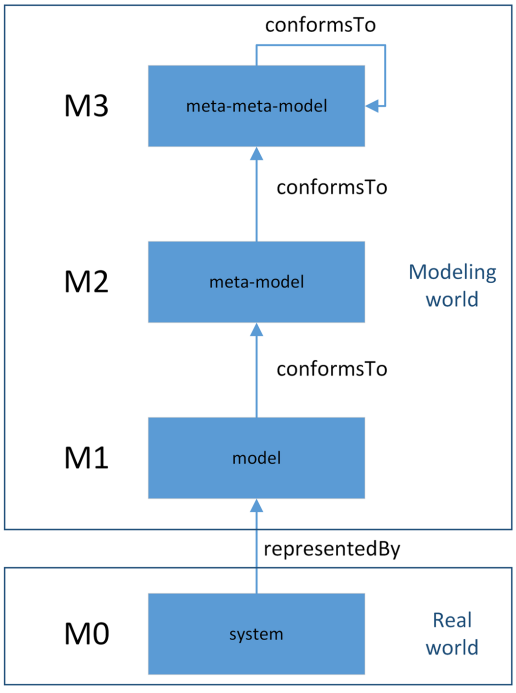
Models@Run.time

Models@Run.time is a term for describing the research field of utilizing software models, specified according to the Model-Driven Software Engineering (MDSE) principles, for self-adaptive software systems (Blair, Bencomo, & France, 2009). The runtime refers to the novelty opposed to the fact that traditional MDSE has been applied for describing the architecture of software and systems at design time. One of the most prominent examples of such a design time technology is the Unified Modeling Language (UML) standardized by the Object Management Group.

Figure 6 illustrates the core principle of MDSE (Brambilla, Cabot, & Wimmer, 2012). Note that a meta-model, also known as modeling language, is in fact a model.

M3: This layer is the basis of the MDSE architecture. Its purpose is to provide a modeling language for defining modeling languages. Usually a meta-meta-model is defined reflexively, that means it can define itself. In practice, it does not make any sense to define further meta layers, (Brambilla et al., 2012). In Figure 6 this behavior is described by a conformsTo relationship.
M2: The purpose of this layer is to describe modeling languages which are used on the next layer for specifying the actual model. It has to conform to the meta-meta-model at layer M3, like a programming language has to conform to its grammar. For instance, UML itself resides on this level.
M1: Models at this layer represent and abstract modeled systems. They have to conform to the corresponding meta-model. An example would be an UML model describing classes of a software.
M0: This layer is not part of the modeling world and part of the real world. It consists of real systems, which are abstracted and represented by M1 models.

Figure 6. Four-layer metamodeling architecture typically used in model-driven engineering
Based on Bézivin (2004).



Now, a runtime model is essentially a software model that represents at runtime parts of a real system and is causally connected to it (e.g. a system change leads to a change in the model). Such software model possesses several properties that are in our opinion beneficial for self-adaptive software systems:

- Design time models are in many domains already available and can be transformed to living specifications at runtime.
- A model can be queried in order to find resources and to learn something about a system.
- Software models are based on modeling languages (M2) and adhere to semantics. Simply put, a mechanism cannot easily construct a model randomly and arbitrary.
- Validation is an important aspect of software models and constraints can be provided for ensuring that a runtime model is correct.
- An adaption mechanism can explore if a change would be correct by forking a model and trying out different configurations.
- Transformation is an essential part of MDSE. Manipulating and transforming models to executable artifacts offers systems an opportunity to self-modify. Further a changed runtime model could be transformed to input formats for a variety of simulation and verification software.
- Runtime models which change over time can be transformed back to design models.

Giese et al. (2014) distinguish between three different kinds of runtime models within a self-adaptive software system. Note that not all kinds have to be present within an adaptive system, but all of them are useful for each activity in an adaptive loop:

- **System Models:** This kind of models reflects an abstract view of the system itself. It allows an adaptive system to reason about the system and to simulate different kinds of configurations. Consequently, such a model needs to be in sync with the real system.
- **Context Models:** A runtime model can be used to reflect the context of a system and to specify it in a processable way. The characteristics of the context cast in such a model can either be derived directly from the environment by sensors or indirectly derived from other observations.
- **Requirements Models:** This kind of models captures the requirements and goals of a self-adaptive system. In a way, it sets the boundaries of what a system can do. The collect, analyze, plan/ decide, or act parts of an adaptive loop can be partly or fully configured with these kinds of model. Usually this relationship is unidirectional, meaning that a system is not supposed to change its requirements. However, it can prioritize one over another.

SCARI: A SECURE AND RELIABLE INFRASTRUCTURE

Scari (**Sec**ure **a**nd **r**eliable **i**nfrastructure) is our approach of a self-adaptive software system that targets to increase the resilience of networked embedded systems typically found in industry. With the term infrastructure, we mean the hardware (e.g. CPU, physical network, etc.) and software (operating system, applications, etc.) stack providing the facilities for running industrial control logic. We do not target to adapt the control logic running on industrial control systems. Instead we want to ensure that devices and networks last longer, operate in the presence of hardware faults, and mitigate security attacks. Further,

we target to make control devices smarter in order to recognize anomalies in the data they receive from their environment.

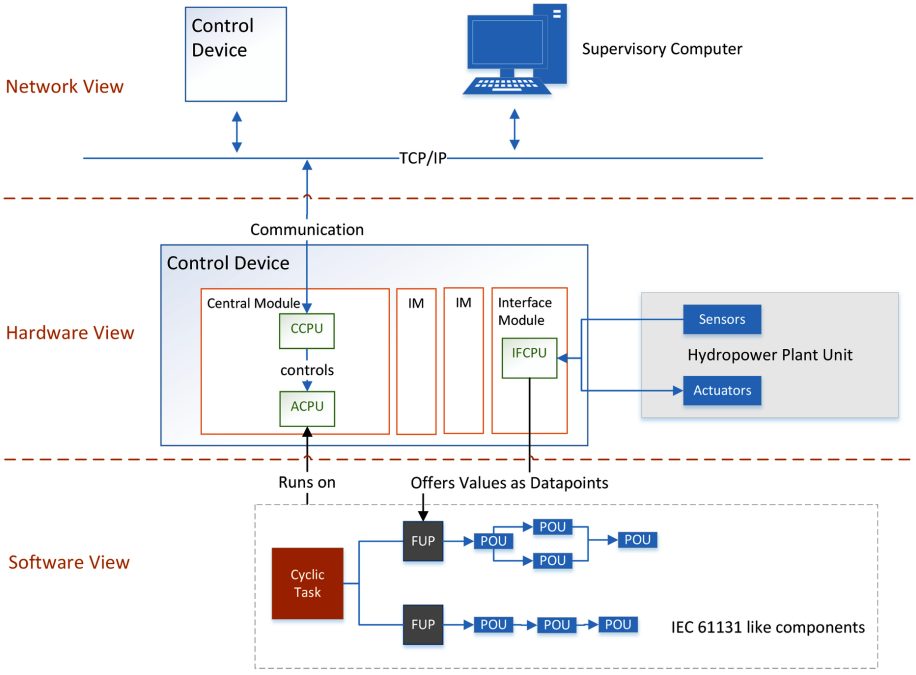
In the following, we start by explaining our specific industrial setting from the hydro-power domain. Then we move on to a detailed specification of the vision of Scari. After that we present thoroughly our approach. Last, we discuss two simplified use cases, one handling a hardware fault, the other illustrates a security attack.

Industrial Setting

The context of this work are distributed control devices that operate hydro-power plant units. The reason why we choose this context is that such systems are also the context of our research project within the presented approach is developed. Figure 7 illustrates a simplified overview of the Supervisory Control And Data Acquisition (SCADA) system we are aiming to make more reliable by applying an adaptive software system.

On network level, control devices are connected via ethernet and operated by a supervisory system. These supervisory computers are mainly responsible for two things. One responsibility is to observe the state of physical processes. The other one is to adjust parameters of control devices in order to control

Figure 7. Overview of the target SCADA system



the energy conversions. The observation and adjustment actions are done by using so-called datapoints which are variables with a certain basic data type like integer or Boolean.

The control devices are connected to hydropower plant units. Their functional responsibility is to operate these units through one of the four different phases namely excitation, synchronization, protection and turbine control.

Technically, these devices have a programmable logic controller (PLC) architecture. Concerning the hardware design, a control device is build out of central modules and interface modules. A central module consists of a communication CPU (CCPU) and an application CPU (ACPU). The CCPU is responsible for network connections and controlling/monitoring the ACPU. It runs a customized Linux distribution and can be accessed by various protocols like SSH and Modbus. From the security point of view, it protects the ACPU and verifies incoming commands. The ACPU is a multi-core processor and executes the actual control logic. It runs a real-time operating system in order to ensure guaranteed cycle times. The interface modules are connecting the control device with sensors and actuators of the hydropower plant unit. Central modules and interface modules are connected via Ethernet.

The control logic software executed by the ACPU of a central module is component-based and heavily influenced by the IEC 61131 standard for programmable logic controllers (John & Tiegelkamp, 2010). Basically, the control logic is hierarchically build out of components, compositions and tasks. Components are called Program Organization Units (POU) and compositions are named Function Plans (FUP). POUs are coded with the C-programming language and stored as binaries on the devices. Such POUs implement basic functions, e.g. simple logic gates, or complex algorithms. Based on these POUs, reusable FUPs are designed that implement the specific control logic for a hydropower plant unit. Technically, FUPs are serialized as XML files and loaded by a POU scheduler. Finally, such FUPs are called by cyclic tasks, for instance every 10 milliseconds. Again, tasks are serialized as XML files.

FUPs operate on datapoints that are set and read by the interface modules. At the start of a cyclic task the necessary datapoints are collected, then the FUPs are executed, and subsequently the calculated datapoints are written back. The interface modules receive these datapoints and actuate accordingly. Further, datapoints are shared with other control devices or supervisory computers.

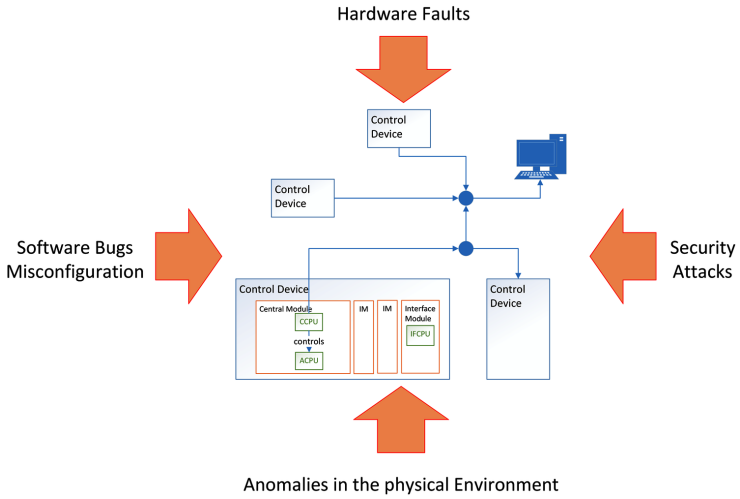
Vision

Roughly speaking, the primary goal of Scari is to provide a generic and reusable infrastructure that allows to establish and orchestrate different kinds of anomaly detection with corresponding adaption mechanisms. The aim of these adaption mechanisms is to increase the resilience of control devices and networks in order to keep control processes as long as possible alive. Depending on the impact of a situation, power plant operators should be alarmed additionally to or instead of an adaption.

Figure 8 illustrates the four different areas we are Scari developing for.

One area are hardware faults. For instance, permanent memory cell faults in RAM or CPU registers can be detected with memory checks. After such a detection, the faulty locations can be circumvented by reconfiguration of the operating system or through diverse compiled software that does not use certain CPU registers. Also, permanent hardware faults in interface modules could be recognized and handled e.g. by using an alternative interface module. A control device should not only be able to recognize its own faulty hardware. As datapoints are distributed to other control devices controlling other parts of the same hydro-power plant unit, they should be able to observe and analyze that something might be wrong

Figure 8. Problem areas we are targeting to detect and adapt to



with the hardware of other control devices or networking devices. Ultimately, the control logic running on one device could be migrated to alternative ACPUs, central modules, or devices.

The second area are security attacks. Each control device knows in principal from whom it receives or sends data to. This information could be used for detecting network attacks or attacker that behave like a control device. Further, infected devices can be detected if the datapoints they are distributing are suddenly or over time odd and do not reflect the real environment. Additionally, the behavior of real devices which are unexpectedly trying to access control devices they are not supposed to, can be a hint for a security incident. Revealed attacks can be handled by blocking and isolating infected devices or network resources. Other kinds of attacks are for instance software that tries to access resources it is not allowed to or sensors that are physically manipulated.

The third area is the environment the control devices are interacting with. The control devices are not directly connected with a water turbine. There are sensors and actuators in between that can break or drift over time. Detecting such anomalies and reacting to them would again make a system more reliable.

The last area are software bugs or misconfigurations of the control devices and networking devices. An adaptive software system can detect high CPU loads, memory consumptions or frequent real-time violations.

An adaptive software system dealing with all of these four areas has the potential of increasing the life-time, reliability, and security of industrial systems. As we can imagine of the examples, not only the analyzed datapoints can often be the same, also adaption mechanisms (e.g. migration) can be reused for handling different faults. Therefore, we believe that one generic architecture orchestrating different anomaly detection mechanisms and adaption strategies is needed. Further, a representation of the context, the requirements and the system itself through models has a significant potential of boosting detection, decision, and adaption mechanisms.

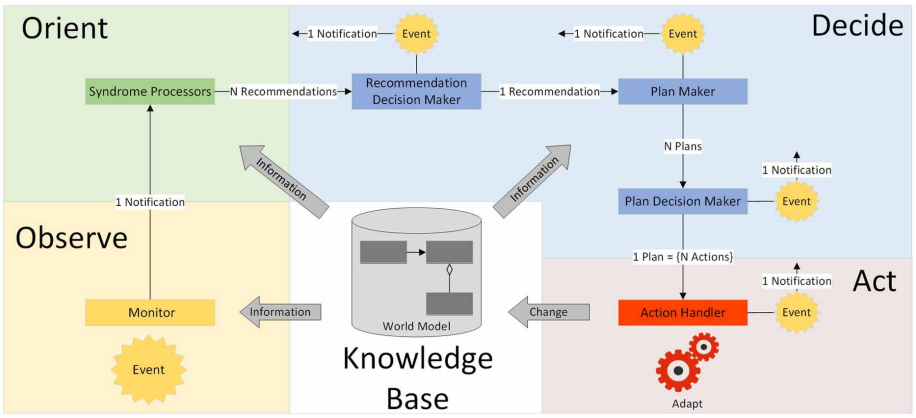
Approach

The underlying principle of Scari is a combination of MAPE-K and the OODA-loop of John Boyd. Figure 9 illustrates the different steps (*Observe, Orient, Decide, Act*), the included entities (*Monitor, Syndrome Processors, Recommendation Decision Maker, Plan Maker, Plan Decision Maker, Action Handler*), and the *Knowledge Base*. The arrows between the entities represent messages. Information between the entities only flows through these messages and they are not coupled with other means. We take the steps *Observe, Orient, Decide, Act* from the OODA-loop because they emphasize an explicit *Decide* step while the MAPE-K loop subsumes this, in our opinion, essential part within the step *Plan*. What we also borrow from the OODA-loop is the feedback from the *Decision* and *Act* steps going back to the *Observe* step. We design this feedback as *Notification* about an *Event* which triggers interested *Syndrome Processors*. As it is emphasized in the OODA-loop, the *Syndrome Processors* residing in the *Orient* step implicitly guide and control the *Observe* and *Act* steps. From MAPE-K we notably adopt the *Knowledge Base* part that is illustrated as *World Model*. It is used as shared information source for the different loosely coupled entities and reflects the state of the world. The *Plan* step of MAPE-K is actually split and embedded into the *Syndrome Processors*, the *Recommendation Decision Maker*, the *Plan Maker* and the *Plan Decision Maker*.

Combining MAPE-K and OODA takes in our opinion the best of both concepts. MAPE-K introduces the *Knowledge Base* as a common information source for the different steps. OODA adds an explicit *Decide* part which is useful for the coordination concerning which kind of mechanism is performed. As explained above, the *Plan* step of MAPE-K is distributed over several loosely coupled entities. What we also take from OODA is that each step gives feedback to *Monitors* and *Syndrome Processors*. This is allowing them to take into consideration what happened with their notifications and recommendations.

In detail the adaptive loop illustrated in Figure 9 works as follows: In the first step, named *Orient*, a *Monitor* recognizes an *Event*. This could be for instance an unexpected datapoint value monitored by a simple voter that compares two runs of the same task (we elaborate this example in one of the following

Figure 9. Adaptive loop of Scari



use cases). The voter is the *Monitor* and the unexpected datapoint represents the *Event*. The adaptive loop itself can only be triggered by an *Event*. In principle, the Scari loop is event-driven and not time-driven.

In the next step, the collected data about the Event is distributed by the *Monitor* as *Notification* message to an arbitrary number of interested *Syndrome Processors*. *Syndrome Processors* are part of the *Orient* step and specialized on recognizing specific syndromes e.g. a security attack based on the received *Notifications* (in MAPE-K *Notifications* are called symptoms). The used techniques for analyzing *Events* and to detect anomalies can technically vary. For some cases a classification approach from the machine-learning domain can be beneficial while for others one receive of a specific Event is enough for instantly diagnosing a syndrome. The *Syndrome Processors* do not have to react every time they receive a *Notification*. They can collect several *Notifications* about different *Events* over time in order to diagnose a syndrome. After a *Syndrome Processor* is sure about an anomaly or wants to find out more about a situation it fires a so-called *Recommendation.* A *Recommendation* consists of a *Plan* type and a collection of *Notifications* that are significant for the *Recommendation*. There exist two general classes of *Plan* types. One kind of types is targeting to find out more about a situation, e.g. a memory test. The other kind changes a system and adapts the software. We treat both classes in the same way because they cost time and can delay other computations. Further, it would make adaption mechanisms very fragile if costly analyzing mechanisms are carried out at the same time on the same CPU or device. Therefore, it is important that only one plan mechanism is executed on a system at one point in time.

The *Recommendation Decision Maker* plays a key role in the orchestration of different *Recommendations* that may be received from different *Syndrome Processors* within a short configurable period of time. Based on a simple definable prioritization, first of the covered *Events* and then of the chosen *Plan* types, it decides which *Recommendation* gets selected. Of course, if the world is locked because a *Plan* is currently executed, the *Recommendation Decision Maker* rejects to decide and distributes an *Event* as *Notification* that it currently cannot decide. It is up to the *Syndrome Processors* to evaluate such a situation. If a *Recommendation* is successfully selected, then the *Recommendation Decision Maker* distributes an *Event* and retransmits the final *Recommendation* marked as selected.

In the next phase, part of the same step *Decide*, the selected *Recommendation* is processed by a *Plan Maker* that can create variants of a chosen *Plan* type based on the information from the *World Model.* A *Plan* consists of a collection of *Actions*. An *Action* is an atomic activity carried out on the target system. The potentially more than one *Plans* are again distributed and a corresponding *Event* is thrown.

After that, the *Plans* are processed by a *Plan Decision Maker* which chooses the *Plan* with the least affected systems/resources and the least amount of used Actions.

In the last step, named *Act*, the selected *Plan* is taken by an *Action Handler*. This entity executes the single *Actions* contained within a *Plan* and changes the *World Model* accordingly. Note that this is the only entity in the adaptive loop that can actively change a system.

As mentioned above, the *World Model* is the central knowledge base for all parts of the adaptive loop in Figure 9. It consists of various models describing parts of the architecture of a system at runtime. Examples of such parts are control logic (Tasks, FUPs, POUs), installed software applications, hardware (RAM, CPU, etc.), network connections and so on. We engineer these architecture runtime models according to the model-driven engineering principles (Brambilla et al., 2012). So, we have metamodels defining domain-specific languages and one meta-metamodel serving as a common technical base for the metamodels. One reason for applying model-driven engineering techniques is that we can precisely describe a part of a system with a specialized language that only the interested entities need to understand. A generalized schema for representing data would make it more difficult to grasp the semantics

and less effective to support the *Observe*, *Orient*, and *Decide* steps. Another reason is that models are manageable reflections that abstract from unnecessary details of the system (Aßmann, Götz, Jézéquel, Morin, & Trapp, 2014). In addition to the *World Model*, the *Knowledge Base* incorporates a log of the distributed messages (*Notification, Recommendation, Plan, Action*) and a revision hash that changes if the *World Model* changes. The revision hash is used by the *Recommendation* and *Plan* messages in order to ensure that they refer to the current state of the *World Model*. If a message is not referring to the current state it is simply ignored by the *Decide* and *Act* entities because it comes from a former state of the world. Crucial for the adaptive loop is that the *World Model* can only be changed with an *Action* executed by *the Action Handler* while others can just access the *World Model* for deriving information.

Note that all entities within the four steps *Observe, Orient, Decide and Act* are executed in parallel in separate processes. The *Recommendation Decision Maker* may be blocked because the *Action Handler* is adapting but it is always running for denying or allowing *Recommendations*.

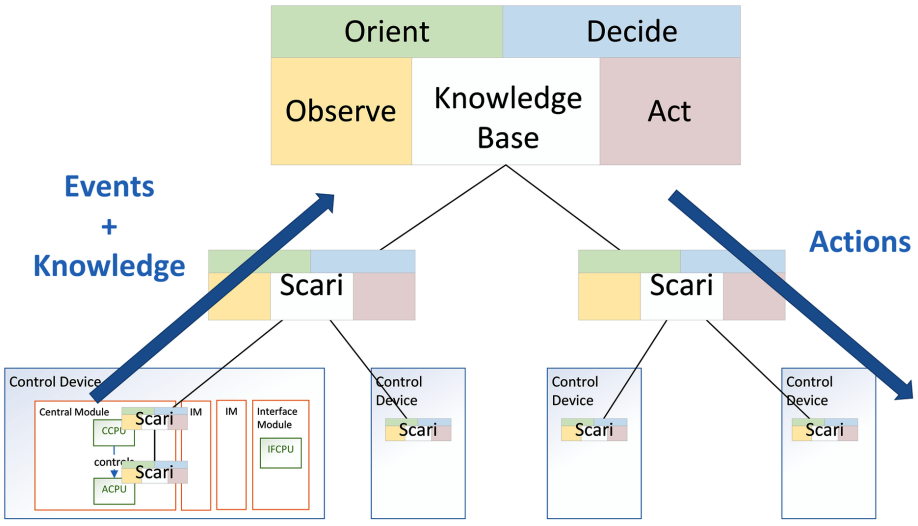
So far, we have only explained the entities and steps of the adaptive loop used by Scari but not how and where we want to deploy them. Figure 10 illustrates that we foresee multiple loops on different adaption layers. The lowest layer resides on the ACPU while the parent layer of that loop is located on the CCPU. On top of that are adaptive loops grouping control devices according to their logical structure.

We organize these loops in a directed acyclic graph. That means a loop can have one to several parent loops where it can escalate to if it cannot handle an anomaly. In literature, organizing adaptive loops in the presented way is known as *Hierarchical Control* pattern (Weyns et al., 2013).

There are two reasons for us organizing the Scari adaption loops in a hierarchical way.

The first reason is that *Knowledge Bases* only need to know their subgraphs. If a *World Model* on a node changes, information is only propagated up to the parent nodes. A *Knowledge Base* may be configured

Figure 10. Multi-layer Scari



to prune lower node data if it is not needed on the higher levels. Further it is more efficient regarding memory consumption if information is only present on nodes or until certain layers, where it is actually needed. Distributing all information on all nodes would additionally lead to more network traffic.

The second reason for a hierarchical organization is that an adaptive loop only needs to handle its subgraph. A loop does not need to manage other parts of the overall graph which also eases the configuration of Scari. If it is not possible to adapt to an *Event* happening on a node, it can be escalated to a parent node that has more knowledge, more and different resources under control and can therefore leverage mightier adaption mechanisms. In our hydro-power setting, it is imaginable that these adaption layers are even laid over different hydro-power plants (then of course acting on bigger time scales).

As shown in Figure 10 the information that flows a graph up are *Events* and *Knowledge*, while the information going down are *Actions* adjusting lower nodes. It is important to note that an adaptive loop residing on a higher layer needs to lock all *Action Handlers* of the lower loops. Otherwise, one lower adaptive loop could be faster with an own *Plan* and the result of two interfering *Actions* would be unpredictable. Even if a lower node is not in the scope of a higher layer *Plan* it could lead to uncertainty if such a node is allowed to change its behavior.

Another aspect that needs to be taken in consideration are the different time scales residing on the different layers. An adaptive loop on the ACPU can react much quicker than an adaptive loop overlooking several control devices. Also, the communication between the layers can take a notably amount of time. Further, the control logic operating a hydro-power plant unit should not be disrupted by observing *Monitors.*

Last a few words on the technical implementation of Scari. In general, we implement the different entities with the C++ programming language together with the Qt framework. The *messages Notification, Recommendation, Plan and Action* within one adaptive loop are distributed over DBus which is a software bus (the developers call it a smart socket) enabling a loose coupling between *Monitors, Syndrome Processors,* and so on. The communication between the layers is implemented with encrypted websockets. The self-adaptive infrastructure itself has to be secured, otherwise it would represent a huge attack surface. The *Plan Maker* and the *Action Handler* entities are offering a plugin architecture in order to create *Plans* for a specific plan type or to implement the actual *Action*. Concerning the *World Models* we are using an own C++ modeling framework inspired by the Eclipse Modeling Framework. Scari itself with the different adaptive layers is supposed to be statically configured beforehand.

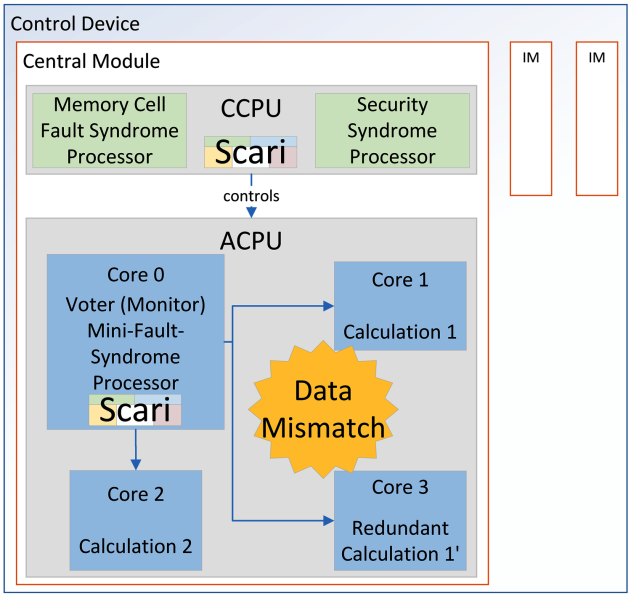
Exemplary Use Cases

In the following we present two examples that demonstrate the potential of Scari. The first one is about a permanent hardware fault located inside the RAM. The second one demonstrates a case where a control device is infected by malicious software.

Hardware Fault Example

Figure 11 shows an overview of a hardware fault case. In this scenario one calculation is running redundant on core 1 and core 3. Both calculations are observed by a voter (*Monitor*). Now, if a data mismatch is happening then the voter notifies a minimalistic syndrome processor that instantly recommends to check the used memory areas and the used CPUs. This *Recommendation* is processed by a *Recommendation Decision Maker* and forwarded to the *Plan Maker*. The *Plan Maker* selects suitable time slots

Figure 11. Overview of the hardware fault example



for the checks. This is possible because the actual task execution time is lower than the specified cycle time. Such time windows open the possibility to perform memory checks seeking permanent hardware faults. After the plan is created, we omit the *Plan Decision Maker*. We scale down the Scari loop on the ACPU for performance reasons. We want to react on the ACPU within a short time period because the control loop is directly affected. Therefore, we omit the *Plan Decision Maker* and also drop the *Knowledge Base* as it costs memory and processing power. After the memory checks are performed, we have a result, which is again processed by the minimalistic syndrome processor. If one memory area in the RAM is affected, then the syndrome processor recommends to just use the results of the working calculation and to ignore the other one. This *Recommendation* goes through the reduced adaptive loop and finally an *Event* is thrown by the *Action Handler* that the reconfiguration finished. Again, the minimalistic syndrome processor reacts on such an Event and notifies the higher-layer about the situation on the ACPU. The higher-layer is located on the CCPU where we have more time on analyzing situations. A security syndrome processor may use this as evidence that something suspicious is going on. A memory fault syndrome processor may recommend to start a redundant calculation on another module part of the device. Or it recommends that the memory area of the affected task gets remapped to circumvent the faulty memory cell and to restart the redundant calculation. Hoeller, Spitzer, Rauter, Iber, & Kreiner (2016) show that diverse compilation of software has the potential of circumventing faulty hardware locations. It is imaginable that such an approach is applicable for us to heal the hardware with recompiled software to a certain degree.

What we learn from this example is that the adaptive loops have different temporal properties. A Scari loop on the ACPU has to react faster than the higher loop on the CCPU. On the ACPU, we are also restricted on the use of parallel processes as we usually only have one core available for adaption purposes. Therefore, we omit a *Knowledge Base* and a *Plan Decision Maker*. The *Plan Maker* is configured to only create one *Plan*. If an ACPU does not execute several tasks the Scari loop may be dynamically configured to load functionality in order to diagnose more syndromes.

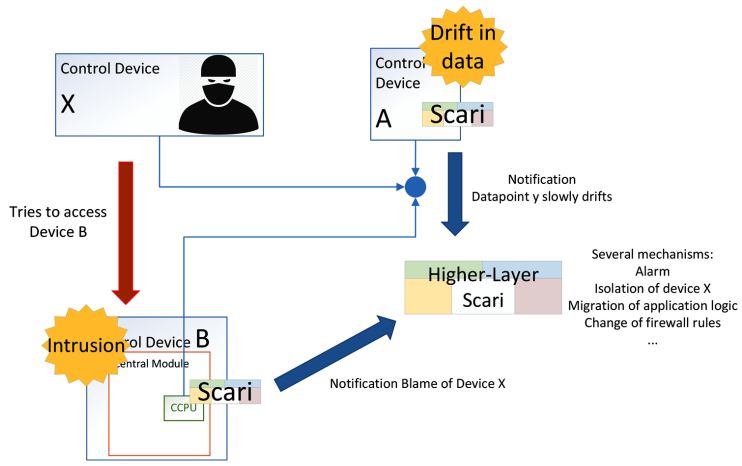
In the presented example, we are in fact dealing with cross-cutting concerns. The faulty memory location may also be caused by a security-related attack. Orchestrating different detection and adaption mechanisms is one of the main benefits of applying a self-adaptive software system like Scari. Another benefit is that recommendations and actions can be reused by different syndrome processors.

Security Example

Figure 12 illustrates an overview of a security-related example. Let's suppose that control device X has been infected with a malicious software. This hostile software may target to slowly destroy a hydro-power plant unit, similar to the intention of the famous Stuxnet computer worm (Falliere, Murchu, & Chien, 2011; Langner, 2011). Syndrome processors, located on the CCPU of control devices, connected to the same hydro-power plant unit, have the means to detect the slow but harmful drift of datapoints. They can react for instance by notifying a higher-layer adaption loop that the data they are observing, drifts. The notified layer has more insights into the situation and is able to react with mightier mechanisms for instance by isolating a device and migrating the application logic.

Further, if an infected device tries to spread the malicious software to other devices, those devices can detect such an intrusion. They obey structural information of what connections are allowed and can therefore come to the conclusion that an unexpected connection attempt from a device is in fact a hostile

Figure 12. Overview of the security example



attack. Based on this diagnosed syndrome, devices could react by blaming the infected device and the higher layer could again isolate and migrate to stand-by devices.

In the presented example, we know beforehand that it is a security-related situation. Syndrome processors have to analyze notifications by running algorithms that detect such patterns. It might also be that there are syndrome processors which are dedicated to diagnose hardware faults and interpret the situation differently. This is another benefit of Scari as it provides a common infrastructure for communication. It allows to compete syndrome processors about the best recommendation. Essentially, this is the reason why it is important to prioritize events and plan types in order to make a decision. For instance, we generally suggest that security-related events have a higher priority than hardware-related. However, it depends on the domain and the available means how events and plan types are prioritized.

RESEARCH CHALLENGES

Although, there is a high potential for the proposed self-adaptive software infrastructure, it possesses many research challenges. We derived following challenges based on our experience with Scari.

1. **Determinism:** Applications usually found in industrial settings have to fulfill real-time requirements. Monitor and especially adapt mechanisms introduce time delays that are complicated to predict. In addition, an adaption mechanism needs to be sure that timing constraints are intact after a system change.
2. **Resource Overhead:** The impact of all entities part of the self-adaptive software system on the resource requirements (e.g. performance, memory, network traffic) should be kept low simply because they are deployed on embedded devices.
3. **Tools and Frameworks:** Implementing a self-adaptive software system is not a trivial task. One has to build several tools and frameworks with well-defined interfaces. The software part of the adaption loop has to be generic and adjustable to the specific domain and adaption layer. Concerning the representation of knowledge, one needs modeling frameworks and data distribution facilities for restricted platforms. Additionally, tools are needed for configuring and deploying such a system.
4. **Runtime Models:** It is crucial for a model used at runtime to abstract only the relevant details of a system in order to support effective adaption (Bennaceur et al., 2014). Models that are too fine-grained can lead to large amounts of data, while higher-level models can ignore relevant details for diagnosing syndromes.
5. **Anomaly Detection Techniques:** Anomalies are patterns in data that do not conform to a well-defined notion of normal behavior. For detecting anomalies there exist a vast amount of different techniques (Chandola, Banerjee, & Kumar, 2009). In Scari, these techniques reside in the single syndrome processors. Finding the right detection technique for an anomaly and deciding what is abnormal is challenging.
6. **Adaption Mechanisms:** Exploring the space of possibilities concerning adaption mechanisms is a domain-specific task. Deducing adjustable generic mechanisms from domain-specific approaches can be fruitful for all different sorts of self-adaptive software systems.
7. **Development Methods for Configuring a Self-Adaptive System:** Applied adaption mechanisms directly affect a system regarding several dimensions like functionality, safety, and security. Further, the valuing of a recommendation over another in the Decide phase of an adaption loop is in fact

a design decision made by system architects. Therefore, we advocate that these aspects actively have to be taken into consideration at design time of a system, for instance during a hazard and risk analysis.

8. **Testing and Assurance:** There need to be means in order to verify that a system is still working as expected after an adaption. In Scari we currently assume that an adaption mechanism is behaving as intended. As pointed out by Salehie & Tahvildari (2009) *testing and assurance are probably the least focused phases in engineering self-adaptive software, and there are only a few research efforts addressing this topic.*
9. **Self-Optimization:** Observing a system and adapting it to become better regarding performance and memory consumption may yield significant improvements. There is of course the risk that one causes unintentionally the opposite effect solely by performing costly optimization observations and adaptions.
10. **Interference of the Managed System Through Human Operators:** Industrial control systems have to be adjustable by operators at runtime. An adaptive system has to be aware of such an intentional manipulation and should not counteract by adaption. As far as we know, little has been mentioned about this potential problem in literature.
11. **Interoperability:** As we are facing systems of systems, it can become crucial for self-adaptive software systems to work together. For this issue, there is a need for well-defined standardized protocols and formats.

FUTURE RESEARCH DIRECTIONS

At the time of this writing, we are in the stage of implementing and trying out different detection and adaption mechanisms. This relates to the two challenges *Anomaly Detection Techniques* and *Adaption Mechanisms*. In the near future, we plan to investigate methods for configuring a self-adaptive system and ways of dealing with interferences through human operators. *Testing and Assurance* is another challenge which we want to tackle by conducting research whether the potential of model-driven engineering can be leveraged for increasing the confidence in performing adaptions.

CONCLUSION

In this chapter, we outlined a novel application of an adaptive software system within a CPS setting. Increasing the resilience of current and future CPS system is one of the key challenges in order to relieve the ever increase of complexity and unpredictability. Self-adaptive software systems are a promising approach of dealing with these key challenges as they observe, diagnose problems, and apply mechanisms in order to adapt and enhance a system. With Scari we presented a concept of a self-adaptive system currently targeting hydro-power plants. We are confident that other industrial or Internet of Things domains can learn from our ongoing approach and that many aspects are reusable. We demonstrate the applicability of this system on two uses cases, one handling a hardware fault, while the other identifies a security attack. We outlined several research challenges derived from Scari and are going to investigate several of them in the future.

REFERENCES

Alhakeem, M. S., Munk, P., Lisicki, R., Parzyjegla, H., Parzyjegla, H., & Muehl, G. (2015). A Framework for Adaptive Software-Based Reliability in COTS Many-Core Processors. In *Proceedings the 28th international conference on architecture of computing systems ARCS '15* (pp. 1–4).

Aßmann, U., Götz, S., Jézéquel, J.-M., Morin, B., & Trapp, M. (2014). A Reference Architecture and Roadmap for Models@run.time Systems. In *Models@run.time: Foundations, applications, and roadmaps* (pp. 1–18). doi:10.1007/978-3-319-08915-7_1

Bennaceur, A., France, R., Tamburrelli, G., Vogel, T., Mosterman, P. J., & Cazzola, W. … Redlich, D. (2014). Mechanisms for leveraging models at runtime in self-adaptive software. In N. Bencomo, R. France, B.H.C. Cheng et al. (Eds.), *Models@run.time: Foundations, applications, and roadmaps* (pp. 19–46). Cham: Springer International Publishing. doi:10.1007/978-3-319-08915-7_2

Bézivin, J. (2004). In search of a basic principle for model driven engineering. *Novatica Journal, Special Issue*, *5*(2), 21–24.

Blair, G., Bencomo, N., & France, R. B. (2009). Models@ run.time. *Computer*, *42*(10), 22–27. doi:10.1109/MC.2009.326

Boyd, J. R. (1996). The Essence of Winning and Losing. Retrieved from http://dnipogo.org/john-r-boyd/

Brambilla, M., Cabot, J., & Wimmer, M. (2012). Model-Driven Software Engineering in Practice. *Synthesis Lectures on Software Engineering, 1*(1), 1–182. doi:10.2200/S00441ED1V01Y201208SWE001

Brun, Y., Desmarais, R., Geihs, K., Litoiu, M., Lopes, A., Shaw, M., & Smit, M. (2013). A Design Space for Self-Adaptive Systems. In R. de Lemos, H. Giese, H. A. Müller, & M. Shaw (Eds.), *Software engineering for self-adaptive systems ii* (pp. 33–50). Berlin, Heidelberg: Springer Berlin Heidelberg. doi:10.1007/978-3-642-35813-5_2

Chandola, V., Banerjee, A., & Kumar, V. (2009). Anomaly Detection: A Survey. *ACM Computing Surveys*, *41*(3), 1–58. doi:10.1145/1541880.1541882

Chandra, A., Lewis, P. R., Glette, K., & Stilkerich, S. C. (2016). Reference Architecture for Self-aware and Self-expressive Computing Systems. In P. R. Lewis, M. Platzner, B. Rinner, J. Tørresen, & X. Yao (Eds.), *Self-aware computing systems: An engineering approach* (pp. 37–49). Cham: Springer International Publishing. doi:10.1007/978-3-319-39675-0_4

Cheng, B. H. C., de Lemos, R., Giese, H., Inverardi, P., Magee, J., & Andersson, J. … Whittle, J. (2009). Software Engineering for Self-Adaptive Systems: A Research Roadmap. In B.H.C. Cheng, R. de Lemos, H. Giese et al. (Eds.), *Software engineering for self-adaptive systems* (pp. 1–26). Berlin, Heidelberg: Springer. doi:10.1007/978-3-642-02161-9_1

Dobson, S., Sterritt, R., Nixon, P., & Hinchey, M. (2010). Fulfilling the Vision of Autonomic Computing. *Computer*, *43*(1), 35–41. doi:10.1109/MC.2010.14

Dobson, S., Zambonelli, F., Denazis, S., Fernández, A., Gaïti, D., & Gelenbe, E. … Schmidt, N. (2006). A survey of autonomic communications. *ACM Transactions on Autonomous and Adaptive Systems, 1*(2), 223–259. doi:10.1145/1186778.1186782

Falliere, N., Murchu, L. O., & Chien, E. (2011). W32. stuxnet dossier. *White Paper, Symantec Corp., Security Response, 5*(6).

Giese, H., Bencomo, N., Pasquale, L., Ramirez, A. J., Inverardi, P., Wätzoldt, S., & Clarke, S. (2014). Living with Uncertainty in the Age of Runtime Models. In N. Bencomo, R. France, B. H. C. Cheng, & U. Aßmann (Eds.), *Models@run.time: Foundations, applications, and roadmaps* (pp. 47–100). Cham: Springer International Publishing. doi:10.1007/978-3-319-08915-7_3

Grenander, S., Simpson, K., & Sindiy, O. (2009). The Autonomy System Architecture. In *Proceedings of the AIAA infotech@Aerospace conference*. Reston, Virigina: American Institute of Aeronautics; Astronautics. doi:10.2514/6.2009-1884

Hinchey, M., & Sterritt, R. (2006). Self-Managing Software. *Computer*, *39*(2), 107–109. doi:10.1109/MC.2006.69

Hoeller, A., Spitzer, B., Rauter, T., Iber, J., & Kreiner, C. (2016). *Diverse Compiling for Software-Based Recovery of Permanent Faults in COTS Processors. In 2016 46th annual ieee/ifip international conference on dependable systems and networks workshop (dsn-w)* (pp. 143–148). IEEE; doi:10.1109/DSN-W.2016.34

Jelasity, M., Babaoglu, O., Laddaga, R., Nagpal, R., Zambonelli, F., & Sirer, E. … Smirnov, M. (2006). Interdisciplinary Research: Roles for Self-Organization. *IEEE Intelligent Systems, 21*(2), 50–58. doi:10.1109/MIS.2006.30

John, K. H., & Tiegelkamp, M. (2010). *IEC 61131-3: Programming Industrial Automation Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg. doi:10.1007/978-3-642-12015-2

Kephart, J., & Chess, D. (2003). The vision of autonomic computing. *Computer*, *36*(1), 41–50. doi:10.1109/MC.2003.1160055

Laddaga, R. (2001). Active Software. In P. Robertson, H. Shrobe, & R. Laddaga (Eds.), *Self-adaptive software: First international workshop, IWSAS 2000* (pp. 11–26). Berlin, Heidelberg: Springer Berlin Heidelberg. doi:10.1007/3-540-44584-6_2

Langner, R. (2011). Stuxnet: Dissecting a Cyberwarfare Weapon. *IEEE Security & Privacy Magazine, 9*(3), 49–51. doi:10.1109/MSP.2011.67

Miller, B., & Rowe, D. (2012). A survey SCADA of and critical infrastructure incidents. In *Proceedings of the 1st annual conference on research in information technology - riit '12* (p. 51). New York, New York, USA: ACM Press. doi:10.1145/2380790.2380805

Muccini, H., Sharaf, M., & Weyns, D. (2016). Self-adaptation for Cyber-physical Systems: A Systematic Literature Review. In *Proceedings of the 11th international workshop on software engineering for adaptive and self-managing systems - seams '16* (pp. 75–81). New York, New York, USA: ACM Press. doi:10.1145/2897053.2897069

Müller, J. P., & Fischer, K. (2014). Application Impact of Multi-agent Systems and Technologies: A Survey. In *Agent-oriented software engineering* (pp. 27–53). Berlin, Heidelberg: Springer Berlin Heidelberg. doi:10.1007/978-3-642-54432-3_3

NIST. (2013). *Foundations for Innovation in Cyber-Physical Systems*.

Oreizy, P., Gorlick, M., Taylor, R., Heimhigner, D., Johnson, G., & Medvidovic, N. … Wolf, A. (1999). An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems, 14*(3), 54–62. doi:10.1109/5254.769885

Salehie, M., & Tahvildari, L. (2009). Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems, 4*(2), 1–42. doi:10.1145/1516533.1516538

Weyns, D., Schmerl, B., Grassi, V., Malek, S., Mirandola, R., & Prehofer, C. … Göschka, K. M. (2013). On Patterns for Decentralized Control in Self-Adaptive Systems. In *Software engineering for self-adaptive systems ii* (pp. 76–107). doi:10.1007/978-3-642-35813-5_4

Wikimedia Commons. (2014). OODA loop. Retrieved from https://commons.wikimedia.org/wiki/File:OODA.Boyd.svg

KEY TERMS AND DEFINITIONS

Anomaly: An anomaly is a pattern in data that does not conform to a well-defined notion of normal behavior.

Model: A model is an abstraction of a system allowing predictions or inferences to be made.

Model-Driven Engineering: In this method models are the key artifacts of all development related activities and tasks.

Models@Run.time: A model@run.time is a causally connected model of the associated system that emphasizes the structure, behavior, or goals of the system.

Reliability: Reliability is the ability of a system to continue its correct service for a specified period of time.

Resilience: Resilience refers to the robustness of a system to adapt itself so as to absorb and tolerate the consequences of failures, attacks or changes.

Self-Adaptive Software System: A Self-adaptive software system is a system that modifies its own behavior in response to changes in its operating environment.

Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 130 (2018) 392–399

Procedia Computer Science

www.elsevier.com/locate/procedia

9th International Conference on Ambient Systems, Networks and Technologies, ANT-2018 and the 8th International Conference on Sustainable Energy Information Technology, SEIT 2018, 8-11 May, 2018, Porto, Portugal

Dynamic Adaption to Permanent Memory Faults in Industrial Control Systems

Johannes Iber*, Michael Krisper*, Jürgen Dobaj*, Christian Kreiner*

Institute of Technical Informatics, Graz University of Technology, Inffeldgasse 16, Graz, Austria

Abstract

Industrial control systems are making increased use of commercial off-the-shelf hardware components. One such component is memory based on DRAM technology. As pointed out by others, DRAM memory can experience permanent hardware errors, e.g. a memory cell can be permanently stuck-at zero or one. In the worst case, such a fault may have serious safety-related consequences. In this work, we present the application of a self-adaptive software system named Scari that detects erroneous datapoints, analyzes them concerning permanent stuck-at faults, and adapts to them by masking defect memory areas. Crucial for this to work is a hot-standby device that takes over the control loop during the detection and adaption phases. The goal of the mechanism presented here is automatic self-repair of a faulty control device to increase its service life and to strengthen overall resilience. The industrial setting of the presented approach is that of control devices for hydropower plant units.

Peer-review under responsibility of the Conference Program Chairs.

Keywords: self-adaptive software system; permanent memory faults; industrial control systems

1. Introduction

Industrial control systems usually consist of distributed embedded devices, such as programmable logic controllers (PLCs) that control physical processes, and supervisory computers that gather data and command PLCs (amongst other devices). In contrast to systems of last decades, modern industrial control systems offer higher performance, are distributed and often connected to the internet. Because industrial control systems are becoming large-scale distributed systems, the inherent problem of complexity is certain to escalate. Systems of this kind must deal with uncertainty, change during operation and moreover be scalable and tolerant to threats[1]. Modern systems are built out of commercial off-the-shelf hardware platforms, which are inexpensive and offer high performance. The downside of these platforms is that typically they offer only limited safety and fault tolerance features[2,3]. One such component is DRAM memory that has been shown to be vulnerable against permanent hard errors such as stuck-at zero or one[4]. Scari (**Sec**ure **a**nd **r**eliable **i**nfrastructure) is our ongoing effort of creating a self-adaptive software system that

* Corresponding author.
E-mail address: firstname.lastname@tugraz.at

1877-0509 © 2018 The Authors. Published by Elsevier B.V.
Peer-review under responsibility of the Conference Program Chairs.
10.1016/j.procs.2018.04.058

aims to increase the resilience of networked embedded devices[5,6]. Our goal is to defend the hardware/software stack underneath the executed control loops in order to execute them correctly for as long as possible. We are simultaneously targeting various different areas such as security attacks and hardware faults[7]. It thus provides an open architecture where arbitrary mechanisms can be plugged in and executed in parallel. With this work we contribute the design of a self-adaptive mechanism within Scari that detects permanent stuck-at DRAM faults and leverages mechanisms provided by Linux to circumvent them. Such a mechanism enables it to automatically repair a control device and to increase the fault-tolerance of an industrial system. We embed this mechanism in a self-adaptive framework because this enables the implementation of several competing mechanisms. Erronous data for example, could also be an indication for a security-related issue.

The remainder of this paper is structured as follows: Section 2 explains our industrial setting. Section 3 provides a brief overview of the related work including our ongoing self-adaptive software effort. Section 4 shows the contribution of this work in detail. Finally, concluding remarks and future work are given in Section 5.

2. Industrial setting

The industrial setting of this work is that of networked control devices operating hydropower plant units. We chose this setting because it is also our project context and we are familiar with it. Fig. 1 illustrates a simplified overview of such a control device. These devices can be deployed in a hot-standby setting where a redundant active device immediately takes over if a fault on the main device occurs.

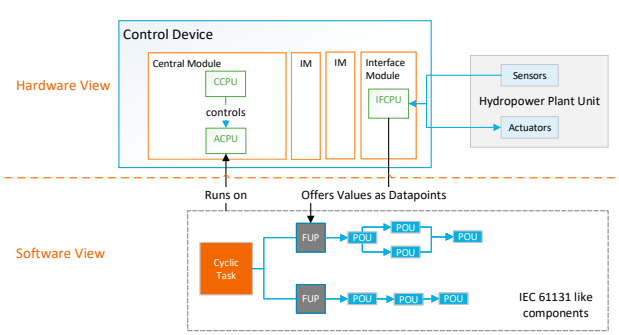


Fig. 1: Overview of the industrial control system

The control devices are connected to hydropower plant units. Their functional responsibility is to operate these units through one of the four different functions namely excitation, synchronization, protection and turbine control. Technically, these devices have a programmable logic controller (PLC) architecture. In the context of the hardware design, a control device is built out of central modules and interface modules. A central module consists of a communication CPU (CCPU) and an application CPU (ACPU). The CCPU is responsible for network connections and controlling/monitoring the ACPU. It runs a customized Linux distribution and can be accessed by various protocols such as SSH and Modbus. From the security perspective this protects the ACPU and verifies incoming commands. The ACPU is a multi-core processor and executes the actual control logic. It runs Linux together with Xenomai[8] (a framework that adds real-time capabilities to Linux) in order to ensure guaranteed cycle times. The interface modules connect the control device with the sensors and actuators of the hydropower plant unit. Central modules and interface modules are connected via Ethernet.

The control logic executed by the ACPU of a central module is component-based and much influenced by the IEC 61131 standard for programmable logic controllers[9]. It is hierarchically built out of components, compositions and tasks. Components are termed Program Organization Units (POU) and compositions are named Function Plans (FUP). POUs are coded with the C-programming language and stored as binaries on the devices. Such POUs implement basic functions, e.g. simple logic gates, or complex algorithms. Based on these POUs, reusable FUPs are designed by plant engineers that implement the specific control logic for a hydropower plant unit. Finally, such FUPs are executed by cyclic tasks in real-time. FUPs operate on datapoints that are set and read by the interface modules. At the start of a cyclic task the necessary datapoints are collected, then the FUPs are executed, and subsequently the calculated datapoints are written back. The interface modules receive these datapoints and actuate accordingly. Further, datapoints can be shared with other control devices or supervisory computers.

FUPs operate on datapoints that are set and read by the interface modules. At the start of a cyclic task the necessary datapoints are collected, then the FUPs are executed, and subsequently the calculated datapoints are written back. The

interface modules receive these datapoints and actuate accordingly. Further, datapoints can be shared with other control devices or supervisory computers.

3. Related work

3.1. DRAM errors

In general, Dynamic Random-Access Memory (DRAM) errors can be categorized into soft and hard errors. Soft errors are transient errors where a memory cell is corrupted temporary, but fully functional before and after the fault. The causes of soft errors are mostly environmental, such as alpha particles, electromagnetic interference or electrostatic discharge. Hard errors are permanent faults in which physical damage has occurred within the memory itself. For instance, a specific bit in a DRAM chip could be permanently stuck-at zero or one. As shown by [10] [11] [12], DRAM errors are dominated by hard errors rather than soft errors. Hwang et al. [12] mention that hard errors account for 95% of all observed DRAM errors in some analyzed systems. Furthermore, multiple errors are more likely to occur near the same rows and columns of the physical layout of a DRAM chip [12].

One common way of dealing with DRAM errors is to use Dual Inline Memory Modules (DIMMs) with error correcting code (ECC) techniques. Typical ECCs are able to correct single bit errors and to detect, but not correct, double bit errors in a memory word. More powerful codes, such as the Chipkill family for example, can correct up to four adjacent bits at once. A study carried out by Schroeder et al. [10] shows that 1.3% a year of all DIMMs in the Google server fleet are affected by so-called uncorrectable errors despite applying ECC techniques. Uncorrectable errors are considered by Google to be serious enough to replace the DIMM at fault. According to the study, platforms where only a simple ECC technique is applied will even have a 3-6 times higher probability of seeing an uncorrectable error than platforms where a more powerful ECC is applied.

Stefanovici et al. [4] conclude that their research shows that a DRAM chip can be perfectly fine if software retires the problematic parts of a memory. They claim that almost 90% of their observed memory-access errors could have been prevented by sacrificing less than 1 megabyte of memory per computer. In this work, we show how this proposal could appear when implemented for industrial control systems.

3.2. Scari

Scari (**Sec**ure **a**nd **r**eliable **i**nfrastructure) is our ongoing research effort of creating a self-adaptive software system that targets to increase the resilience of networked embedded system like control devices used in hydropower plants. With the term *infrastructure* we mean the hardware (e.g. CPU, physical network, ...) and software (operating system, applications, ...) stack providing the facilities for running industrial control logic. We do not target to adapt the control logic itself. Instead, we want to ensure that devices and networks last longer, operate in the presence of hardware faults, and mitigate security attacks. Furthermore we have the target of making control devices smarter in order to recognize anomalies in the data they receive from their environment. These targets are explained in detail in [7] where we show how the determinism found in industrial control systems can be used to increase resilience for the areas hardware faults, security attacks, misconfiguration, faults in the physical device environment, and software bugs. In the following, we briefly describe the relevant properties of Scari in order to explain the contribution of this work. A more detailed explanation of Scari can be found in [5] [6].

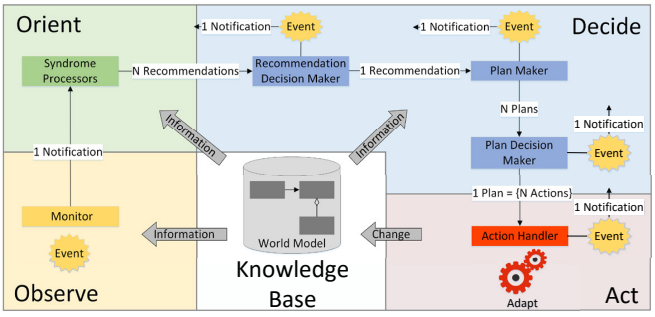


Fig. 2: Scari adaptive loop.

Fig. 2 illustrates the adaptive loop of Scari. It consists of 5 parts, which are *Observe*, *Orient*, *Decide*, *Act*, and a common *Knowledge Base*.

The *Observe* part consists of *Monitors* that are specialized in discovering and measuring specific anomalies, such as for example a drift in data. These monitors notify an arbitrary number of interested *Syndrome Processors*, residing in the *Orient* part. The *Syndrome Processors* implement a specific detection mechanism e.g. for hardware faults or security attacks. Technically, they can use any viable method for detection like machine-learning or simple thresholds. If one or several *Syndrome Processors* diagnose a problem, they recommend plan types for handling a situation. For instance a hardware fault *Syndrome Processor* may recommend circumventing a damaged module, while a security *Syndrome Processor* may recommend isolating a device. Following on from this the *Recommendation Decision Maker* selects the best recommendation on the basis of a definable prioritization for the covered events and chosen plan types. The selected recommendation is then forwarded to the *Plan Maker* that creates the actions for the plan type. Some plan types can be implemented in different ways, we thus added a *Plan Decision Maker* that selects the plan with the least affected systems/resources and the lowest number of used actions. In the final part, the plan is executed by an *Action Handler*, and the system is adapted. Each of the entities of the *Decide* and *Act* parts feed back their states as events. This enables the *Syndrome Processors* to log the state of their recommendations and to be notified in turn that the system has adapted. The *Knowledge Base* provides support for the other four parts. It contains the deployed models from design time and additional run time information. It serves as a source of knowledge for the *Observe*, *Orient* and *Decide* parts, while the *Act* part stores the executed changes of the system there.

In our approach we combine MAPE-K [13 14] with John Boyd's OODA loop [15 16]. This leads in our opinion to the best of both concepts. MAPE-K consists of the steps *Monitor*, *Analyze*, *Plan*, *Execute*, and a shared part representing *Knowledge*, while OODA stands for *Observe*, *Orient*, *Decide* and *Act*. The *Analyze* and *Plan* steps of MAPE-K are subsumed within OODA as *Orient*. Furthermore, MAPE-K introduces the *Knowledge Base* as a common information source for the different steps. OODA adds an explicit *Decide* part which is useful for the selection of recommendations. The *Plan* step of MAPE-K is distributed over several loosely coupled *Syndrome Processors*. We also take over from OODA the capability of each step for giving feedback in the form of notifications to *Monitors* and *Syndrome Processors*. This allows them to consider what happened with their notifications and recommendations.

In our industrial setting, Scari runs on all devices and is organized in a hierarchical manner. For instance CCPU loops are supervised by a Scari loop executed on network level. This hierarchical organization of Scari has two advantages: One is that a *Knowledge Base* only needs to know its subgraphs. If a *World Model* on a node changes, information is only propagated up to the parent nodes. The second advantage is that an adaptive loop only needs to handle its subgraph. A loop does not need to manage other parts of the overall control system which also eases the configuration of Scari. If it is not possible to adapt to an event occurring on a node, it can be escalated to a parent node that has more knowledge, more resources and can therefore leverage more powerful adaption mechanisms. In our hydropower setting, it is conceivable that these adaption layers are even laid over different hydropower plants, where they are acting on a greater time scale.

Technically, we implement Scari in C++ together with the Qt framework [1]. The *Knowledge Base* uses a C++ modeling framework developed by us and inspired by the Eclipse Modeling Framework [2]. It leverages the libgit2 library [3] for distributing models. Scari itself with its different adaptive layers is intended to be statically configured in advance. Furthermore, the architecture allows the dynamic addition of monitors, syndrome processors, plan creation mechanisms and actions in run time.

4. Approach

The goal of our approach is to restore the functionality of a device after memory failures by masking out the faulty memory areas and continuing to operate with the still functional memory. In order to do this, we observe datapoints, perform memory tests on the addresses of the affected datapoints and mask the faulty areas with the Linux kernel boot

[1] https://www.qt.io/
[2] http://www.eclipse.org/modeling/emf/
[3] https://libgit2.github.com/

parameter *memmap*[17]. *memmap* enables to mark specific memory addresses as reserved and thus are not assigned by Linux. The physical memory addresses can be obtained in Linux by using the *pagemap* functionality[18].

The whole process runs through several adaption cycles which react on the results of the previous adaption. Upon finding an error we immediately activate the hot standby device to buy time for the adaption and guarantee uninterrupted functionality for the overall control system. Now that we have relaxed the time constraints on the device, the monitoring level is increased to obtain a more detailed error description. Using this description a detailed memory test for the affected memory locations is carried out and if faulty memory locations are found, these are masked as reserved in the operating system. Subsequently the system can be restarted and can act as a new hot-standby device.

In the following subsections we present the self-adaptive process by using BPMN as a graphical notation. We chose BPMN because it fits the event-based and multicomponent architecture of Scari. We split the process into sections representing the Scari self-adaptive loop: Observe, Orient and Decide & Act. The Observe step is explained in Subsection 4.1 which explains the detection of permanent memory faults in detail. The Orient Step and Decide & Act steps are described in Subsections 4.2 and 4.3 which outline the different adaption mechanisms and how they work together in the self-adaptive system. Subsection 4.4 explains how we evaluate our mechanism. Finally, Subsection 4.5 discusses the pros, cons and tradeoffs of our approach.

4.1. Detection: datapoint monitor

The first step is the detection of memory faults. We detect memory faults by using a comparison-based approach. We implemented a *datapoint monitor* component for this purpose which periodically compares the data values of a running task in the system with a model which resembles the task functionality. This monitor has multiple monitoring levels to minimize the impact during normal operations. In its "low" monitoring level, the monitor is run every few seconds and simply compares the input and output values of our function plans, while in the "high" monitoring state all available datapoints are compared (even datapoints inside the FUPs). Tasks internally may consist of many function blocks with individual inputs and outputs and intermediate variables. The reading of these requires a long time period and this exhaustive comparison thus induces a much higher impact on the system performance, which is the reason why it is only applied if really needed. We introduced these monitoring levels to make sure that the detection mechanism has as little impact as possible on the production system during normal operation.

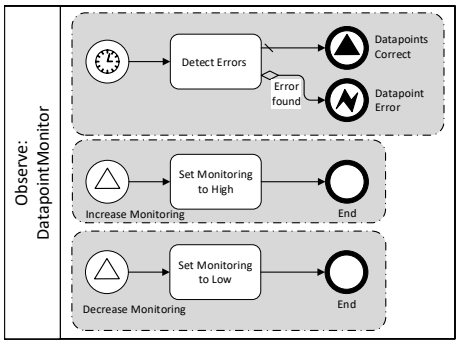


Fig. 3: Observe. The datapoint monitor periodically checks datapoints for correctness.

Fig. 3 shows the process definition for the *datapoint monitor*. It consists basically of three parts: Periodically checking for datapoint errors, increasing, and decreasing the monitoring level.

4.2. Adaption: Orient Phase

After a faulty datapoint is found by the monitor the *datapoint syndrome processor* has the task of recommending a possible solution to resolve this fault. This is implemented via several adaption loops, escalating more and more until the functionality can either be restored or the device must be shut down and repaired using other means, e.g. escalating the problem to maintenance teams. The *datapoint syndrome processor* listens for datapoint errors and recommends different actions in order to solve the errors. To run through these different escalation steps we implemented it as a state machine which advances its internal state according to defined conditions. The internal state is only changed when the *Decide & Act* steps actually decided to go for the given recommendation. This allows for other syndrome processors to intervene with higher prioritized recommendations if needed. If another recommendation is chosen which changes the world state, the escalation level has to be reset to 0 because the problem could be solved in the mean time and it would not make sense to continue based on old world information. Fig. 4 shows the process definition for the orient phase and the *datapoint syndrome processor*.

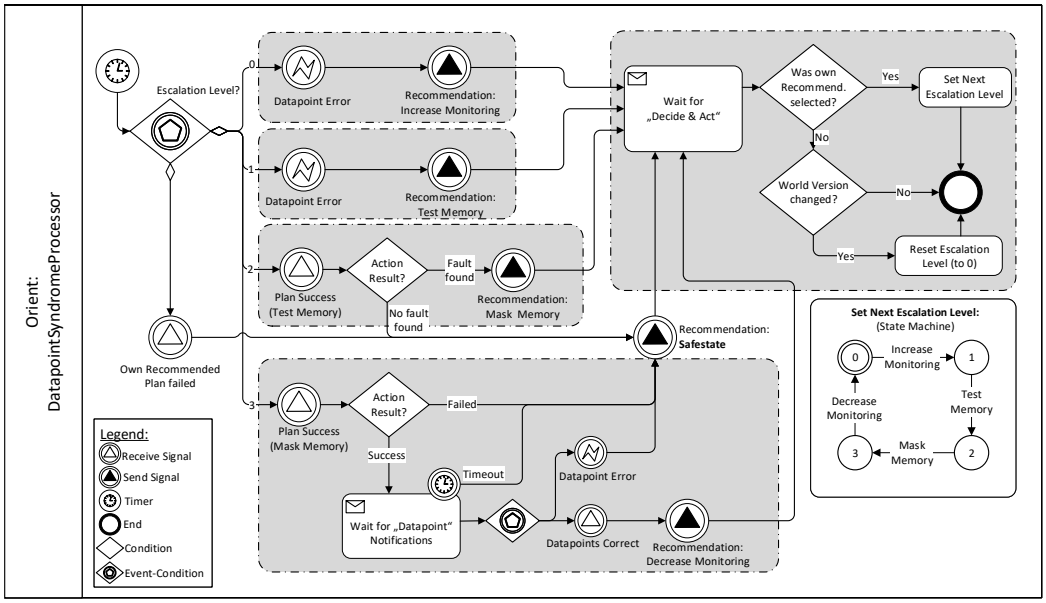


Fig. 4: Orient. Depending on the current state the datapoint syndrome processor expects specific signals and gives recommendations based on them. Only after a recommendation has been chosen and the respective plan was executed is the state advanced further.

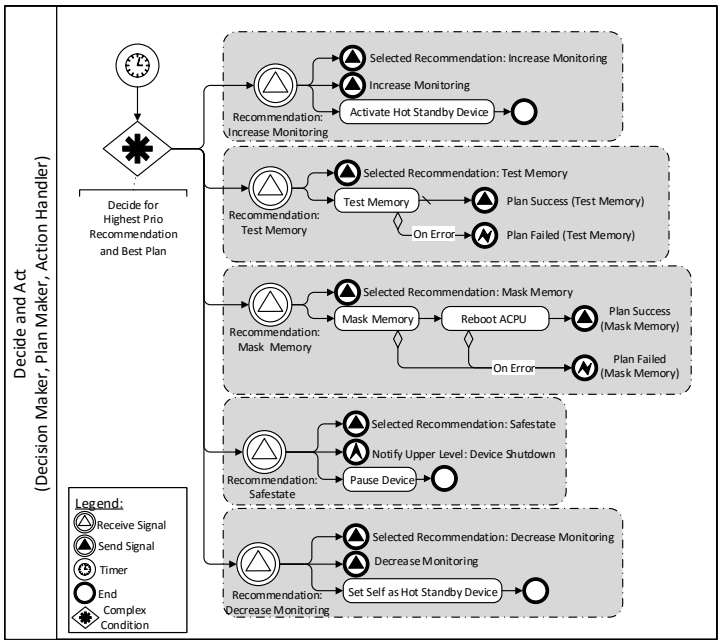


Fig. 5: Decide and Act. The Decisions Makers and Plan Makers react on the given recommendations and enact the best known plan for them.

After the recommendation is selected and the respective plan is executed, the escalation level of the syndrome processor is changed. Here the 4 escalation levels from the process definition are described:

- *Escalation Level 0*: On the first notification of a fault, the syndrome processor recommends to activate the hot-standby device to buy time for further actions and to increase the monitoring level for a more detailed analysis. The syndrome processor goes to escalation level 1.
- *Escalation Level 1*: If the error is still persistent, a memory test of the affected memory regions is recommended and the syndrome processor goes to escalation level 2.
- *Escalation Level 2*: When the memory test detected faulty memory locations, masking of those locations and going to escalation level 3 is recommended, otherwise the syndrome processor recommends going to a safe state and escalating the problem to a higher Scari hierarchy level, because it does not know of any other means to repair this fault.
- *Escalation Level 3*: After the faulty memory areas are masked the device should work again as expected, but if this is not the case, going to a safe state and escalating the problem to higher levels is recommended. This step waits for successful notification that all datapoints are correct, before returning to escalation level 0.

4.3. Adaption: Decide & Act Phase

In this phase Scari decides on the most important recommendation and creates a plan based on the component structure provided by the world model. In this phase the actual adaption takes place and all waiting syndrome processors thus also have the task of subsequently changing the internal states (depending on the outcome of the actions). The defined recommendations and the resulting actions are described in the following table:

Recommendation	**Actions**	**Description**
Increase monitoring	Activate hot standby Increase monitoring	We know an error of some kind has occurred and immediately activate the hot-standby system. For further analysis we increase the monitoring level to obtain more detailed information about the fault.
Test memory	Test memory	A test memory routine is enacted on the memory locations of the faulty datapoints.
Mask memory	Mask memory Reboot ACPU	If we find defect memory cells we mask them in the operating system, which demands a reboot to become effective.
Safestate	Notify upper level Pause device	If the error in the datapoints still occurs even after masking the memory we must go into safe state (shut down the system) and escalate to the next upper level in the Scari loop hierarchy.
Decrease monitoring	Hot standby Decrease monitoring	We set the repaired device in hot standby mode and decrease the monitoring.

4.4. Evaluation

To evaluate our approach we implemented a memory-injection mechanism in QEMU[19] to change memory content during runtime. By injecting a stuck-at error we are able to simulate our use-case and see if the adaption really works as intended. Our test setting consists of a Raspberry Pi 3 Model B running Scari and the PLC software running in QEMU on a desktop PC. In order to evaluate the correctness of our approach we first start by running the PLC software under normal conditions. Then we inject a stuck-at error at a known memory location. Finally, we observe Scari repairing the permanent error. The first recommendation *Increase monitoring* takes in average 73 ms from the decision to the successful execution. The second recommendation *Test memory* needs 630 ms. The third recommendation *Mask memory* lasts one and a half minutes. The last recommendation *Decrease monitoring* takes 72 ms.

4.5. Discussion

Many of the design decisions are guided by the aspect of performance and satisfying real time constraints. That is why we immediately activate a hot-standby device to guarantee seamless functionality. By only testing suspected faulty memory locations we also save much time and effort, compared to that which is required for a full memory test. After a successful adaption we do not deactivate the hot-standby device to avoid further interruption of operations, but instead simply switch the role of the device to be available as a new hot-standby. This results in continued functionality and fault-tolerance of the system. In such a way even a permanent memory error does not shift or disturb operations and maintenance plans and allows for the usage of commercial off- the-shelf hardware components.

5. Conclusion and future work

In this work we presented a self-adaptive mechanism in Scari for adapting to permanent memory faults in the context of control systems for hydropower plants. The novelty of this work is that we propose a mechanism for PLCs where permanent memory faults influencing datapoints are recognized and repaired during the operation of a control system. The faulty memory area is blacklisted by using *memmap* a Linux kernel boot parameter. To verify our proposed mechanism we use a customized version of QEMU [19] in order to artificially inject memory-faults.

We are investigating ways of taking the physical layout of a DRAM memory into account in our future work. Collocated and physically connected memory cells are also likely to be affected and as pointed out by [12], based on the knowledge how multi-bit errors occur, a system could take proactive measures to protect against errors. In the context of the monitoring levels we plan to implement more intelligent ways of finding the faulty datapoints by deriving a data flow graph from the function plans and applying a graph search for finding the faults. Furthermore, we are planning to expand the presented approach to network level to detect permanent hardware faults.

References

1. H. Muccini, M. Sharaf, D. Weyns, Self-adaptation for Cyber-physical Systems: A Systematic Literature Review, in: SEAMS '16, ACM Press, 2016. doi:10.1145/2897053.2897069.
2. M. S. Alhakeem, P. Munk, R. Lisicki, H. Parzyjegla, H. Parzyjegla, G. Muehl, A Framework for Adaptive Software-Based Reliability in COTS Many-Core Processors, in: ARCS 2015.
3. A. Höller, B. Spitzer, T. Rauter, J. Iber, C. Kreiner, Diverse Compiling for Software-Based Recovery of Permanent Faults in COTS Processors, in: 46th Annual DSN-W, IEEE, 2016. doi:10.1109/DSN-W.2016.34.
4. I. Stefanovici, A. Hwang, B. Schroeder, Battling borked bits, IEEE Spectrum 52 (12). doi:10.1109/MSPEC.2015.7335798.
5. J. Iber, T. Rauter, C. Kreiner, A Self-Adaptive Software System for Increasing the Reliability and Security of Cyber-Physical Systems, in: Solutions for Cyber-Physical Systems Ubiquity, IGI Global, 2017. doi:10.4018/978-1-5225-2845-6.ch009.
6. J. Iber, T. Rauter, M. Krisper, C. Kreiner, An integrated approach for resilience in industrial control systems, in: 47th Annual DSN-W, 2017. doi:10.1109/DSN-W.2017.23.
7. J. Iber, T. Rauter, M. Krisper, C. Kreiner, The Potential of Self-Adaptive Software Systems in Industrial Control Systems, Springer International Publishing, 2017. doi:10.1007/978-3-319-64218-5_12.
8. Website of the Xenomai Project, https://xenomai.org/ (2018).
9. K. H. John, M. Tiegelkamp, IEC 61131-3: Programming Industrial Automation Systems, Springer Berlin Heidelberg, 2010. doi:10.1007/978-3-642-12015-2.
10. B. Schroeder, E. Pinheiro, W.-D. Weber, DRAM errors in the wild, Communications of the ACM 54 (2). doi:10.1145/1897816.1897844.
11. V. Sridharan, D. Liberty, A study of DRAM failures in the field, in: 2012 International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2012. doi:10.1109/SC.2012.13.
12. A. A. Hwang, I. A. Stefanovici, B. Schroeder, Cosmic rays don't strike twice, in: Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '12, ASPLOS XVII, ACM Press, 2012. doi:10.1145/2150976.2150989.
13. J. Kephart, D. Chess, The vision of autonomic computing, Computer 36 (1). doi:10.1109/MC.2003.1160055.
14. Y. Brun, R. Desmarais, K. Geihs, M. Litoiu, A. Lopes, M. Shaw, M. Smit, A Design Space for Self-Adaptive Systems, Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-35813-5_2.
15. J. R. Boyd, The Essence of Winning and Losing, http://dnipogo.org/john-r-boyd/ (1996).
16. A. Chandra, P. R. Lewis, K. Glette, S. C. Stilkerich, Reference Architecture for Self-aware and Self-expressive Computing Systems, Springer International Publishing, 2016. doi:10.1007/978-3-319-39675-0_4.
17. Memmap documentation, https://www.kernel.org/doc/html/v4.14/admin-guide/kernel-parameters.html (2018).
18. Pagemap documentation, https://github.com/torvalds/linux/blob/v4.14/Documentation/vm/pagemap.txt (2018).
19. Website of Scari QEMU, https://github.com/jib218/scari-qemu (2018).

Separation of processing and coordination in computer systems

Johannes Iber, Michael Krisper, Jürgen Dobaj and Christian Kreiner
Institute of Technical Informatics
Graz University of Technology
Austria

ABSTRACT

Systems are built for a purpose. The purpose transacted is usually handled by the processing part of a system and is observed and adjusted by coordination parts. In principle, these two kinds of system parts share the same target resource; the thing that is controlled by processing and indirectly by coordination subsystems. This leads to mutual influences, which can result in timing and priorities violations as well as performance degradations. The presented pattern, SEPARATION OF PROCESSING AND COORDINATION, provides an architectural solution which shows how processing subsystems can be observed and adjusted by coordination subsystems. We show this pattern in the context of self-adaptive software systems, industrial control devices, a real-time operating system, and a hardware architecture for wireless embedded platforms.

CCS CONCEPTS

• **Software and its engineering → Patterns**; *Designing software*;

KEYWORDS

system design, processing, coordination, design pattern

ACM Reference Format:
Johannes Iber, Michael Krisper, Jürgen Dobaj and Christian Kreiner. 2018. Separation of processing and coordination in computer systems. In *23rd European Conference on Pattern Languages of Programs (EuroPLoP '18), July 4–8, 2018, Irsee, Germany*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3282308.3282322

1 INTRODUCTION

A well-known engineering practice is to divide a system into subparts in which each part is specialized for a task [6]. The presented pattern, SEPARATION OF PROCESSING AND COORDINATION, is based on this Divide & Conquer principle and extends it to integrating coordination and processing parts of systems. With the term *processing* we mean a part of a system that executes the intended purpose of a system. Usually, processing parts have exclusive access to a resource which can be data or a physical mechanism. With the term *coordination* we mean parts of a system that observe, adjust and protect processing parts. This pattern presents an architectural solution showing how coordination and processing subsystems can coexist and interact with each other. The application context of the pattern is widespread and not limited to a specific domain. We present it in the context of self-adaptive software systems, industrial control systems, a real-time operating system and a hardware architecture for wireless embedded platforms. System and software architects are the primary target audience of this work. In the following Section, we present the design pattern in detail. Finally, Section 3 concludes this work.

EuroPLoP '18, July 4–8, 2018, Irsee, Germany

ACM ISBN 978-1-4503-6387-7/18/07…$15.00
https://doi.org/10.1145/3282308.3282322

2 PATTERN: SEPARATION OF PROCESSING AND COORDINATION

Context

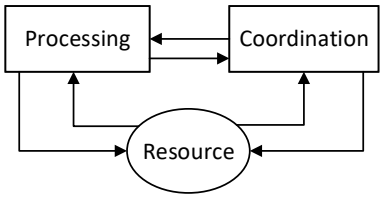


Figure 1: Illustration of the context.

The context of this pattern are systems that consist of at least one subsystem that processes a resource, while other arbitrary coordinating subsystems that need to observe, adjust and protect the processing subsystems or target resource also exist. Figure 1 illustrates this situation in an abstract manner. Note that processing and coordination may have distinct timing requirements. For instance in an industrial control system [3], a controller (processing) needs to control a physical process (resource) while also being observable and adjustable by a supervisory computer (coordination). The primary purpose of such a controller is to control the physical process within a defined time span. Additionally, the controller needs to distribute data to other devices or receive adjustments from higher ranking supervisory computers. All of these coordination tasks need to be done while supporting the physical process in real time. The supervisory computer may not be bound to specific real time requirements. This situation roughly describes the problem the presented pattern aims to solve.

Problem

In a situation such as that given in Figure 1, the essential problem is that the target resource is shared between the processing and

coordination subsystems. Processing needs to operate on the resource, while coordination needs to observe, adjust and protect the processing subsystem or resource. These distinct objectives lead to interferences between processing and coordination. In this pattern, we provide a solution for how subsystems of these kinds can be successfully put together.

Forces

- *Goals*: Coordination and processing pursue different goals. Coordination aims to observe, adjust and protect, while processing is focused on executing tasks. This leads to unavoidable interferences, because coordination needs to disrupt processing. Seen the other way around, processing may need to shift information in time to coordination.
- *Testing*: Ideally, both kinds of subsystems can be developed and tested independently from each other in order to verify their diverse goals. This is complicated if the coordination and processing parts are mixed with each other within a monolithic architecture.
- *Time scales*: The processing subsystem needs to act within a specific time frame. The coordination subsystem seeks to interfere at arbitrary points in time. This interference of coordination could lead to a serious delay within the processing subsystem.
- *Security*: Coordination and processing subsystems may face distinct attack vectors.
- *Execution environments*: Due to different time scales, goals and security requirements, it is desirable to deploy coordination and processing in distinct execution environments. For instance, processing may need to meet real time requirements, which demand a corresponding operating system. Coordination subsystems may not be bound to such requirements and can be deployed on arbitrary platforms. Deploying both on the same technical platform could restrict the capabilities of one subsystem.
- *Priorities*: Actions, performed by processing and coordination, can have different priorities. For instance, coordination could cause an emergency stop, which might have a higher priority than anything processing is doing. This could be an exception and usually processing needs to perform more important activities.

Solution

Simply put, processing needs to be isolated from coordination, communication channels enable interaction between processing and coordination, and processing needs to deal with interactions from coordination at certain points in time. The target resource is supposed to be accessed by processing. In detail the solution consists of the following three parts, namely *isolation*, *communication channels*, and *synchronization points*, that need to be applied together.

Isolation: Processing and coordination need to be executed in strict isolation to ensure that they cannot interfere each other functionally or in the context of non-functional properties such as timing, memory usage or security. Figure 2 illustrates three alternative methods of isolating processing and coordination. Figure 2a shows that coordination and processing can be realized as distinct devices

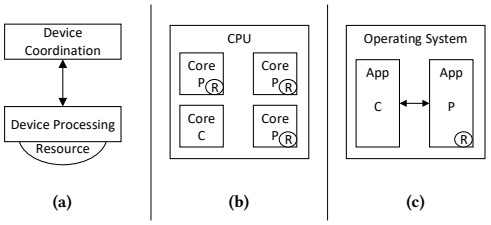


Figure 2: Three alternative ways of isolating processing and coordination.

that are connected. The target resource is managed by the processing device. Figure 2b shows how processing and coordination are executed on the same hardware but within distinct software execution spaces e.g. by leveraging a hypervisor or by executing the software on different CPU cores. Figure 2c only isolates processing and coordination on software level. This can be accomplished by leveraging the operating system where the coordination and processing software run as different processes or inside one software application by creating threads. Each of these alternative ways has different pros and cons. In general, coordination and processing are less isolated from each other by the second and especially the third method. This part of the pattern solution deals with the forces *goals*, *security*, *testing* and *execution environments*. In the context of *goals*, it enables the two kinds of subsystems to be executed without disturbing each other. This also supports *security* because it limits the attack possibilities between these subsystems. The *testing* of such a system should become easier because the subsystems can be tested independently from each other. Depending on the way processing and coordination are isolated, the subsystems can be deployed on diverse hardware and software stacks.

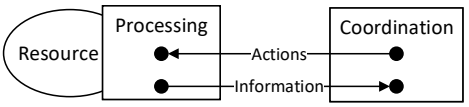


Figure 3: Communication channels are a necessary part of the pattern.

Communication channels: As mentioned above processing and coordination need to be isolated. However, coordination naturally needs to observe and adjust the processing subsystem. In order to do this in a synchronized manner, processing needs to provide communication channels for coordination. Figure 3 illustrates a simplified version of this concept. Communication channels of this kind should allow the queuing of actions for the coordination subsystems to enable the subsequent part of the solution, the *synchronization points*. Furthermore, information about the resource can be transferred from processing to coordination in a uniform manner and through explicit communication channels. Technically,

communication channels could be realized for instance as remote procedure calls, message queues, or RESTful web services. This part of the pattern solution supports the force *goals* by coordinating the way subsystems communicate. Furthermore, *testing* becomes easier because the interactions are explicit. Additionally, this part deals with the force *priorities* if a realization differentiates communication channels according to their importance.

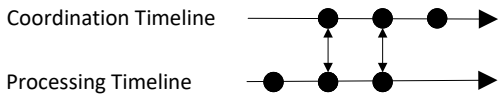


Figure 4: Synchronization points enable to set properties and exchange information between processing and coordination at specific points in time.

Synchronization points: Processing commonly executes actions that are atomic. For instance an update of a database may need to be performed with an atomic transaction, otherwise the database could be left inconsistent if the transaction is aborted. The situation is similar to this if a processing device alters a physical process where the intermediate state might be unknown to the device. It is necessary for the processing to include synchronization points during execution in which the observations and adjustments of coordination are handled. Corresponding with this, coordination may utilize synchronization points for receiving information from the processing subsystem. Figure 4 illustrates this concept. Processing can now perform atomic actions without interruption until it encounters synchronization points. Within these points it may be necessary to copy and send data or the flow of execution may be adapted. Ideally, processing can perform atomic actions where coordination has a chance to apply observations and adjustments between the actions. Depending on the purpose and domain these synchronization points need to be bound by a time limit. This part of the pattern deals with the force *time scales* because it prevents coordination to disrupt processing at arbitrary points in time. Coordination can also deal with information from processing at distinct synchronization points. Furthermore, this part of the solution supports the force *priorities* because processing can prioritize the coordination actions within the synchronization points.

Consequences

+ *Goals*: Data flows between processing and coordination become explicit. This helps to prevent unintended interferences.
+ *Goals*: The communication channels become explicit in the system architecture which also happens to separate the goals of the subsystems.
+ *Testing*: Processing and coordination become simpler to test because they offer defined communication channels.
+ *Time scales*: Processing and coordination can act in different time scales through isolated execution spaces.
+ *Priorities*: The priorities of the actions can be handled within the synchronization points of processing.
+ *Execution environments*: Processing and coordination can be executed in different environments. Both only need to deal with the communication channels.
+ *Security*: Through the separation and communication channels, processing and coordination are also isolated from a security point of view. The communication channels and the synchronization points allow the limiting of the attack surfaces.
- *Testing*: Isolating processing and coordination introduces the need for integration tests.
- *Time scales*: Binding the time of a coordination action may be complicated.
- *Time scales*: Communication channels introduce latency between coordination and processing.
- *Priorities*: Exceptional actions, such as an emergency stop of an industrial system may need to circumvent the processing synchronization points.

Related patterns

Eloranta et al. [2] describe in their work a pattern language for distributed industrial control systems consisting of 45 single patterns. The central pattern of this language is the ISOLATE FUNCTIONALITIES pattern. It proposes dividing a system into subsystems according to functionalities and to connect the subsystems with a bus. This reflects two parts of our solution namely *isolation* and *communication channels*. The MESSAGE QUEUE pattern proposes the utilizing of a queue at each node to enable the receiving node to read messages as soon as it has time to process them. This corresponds to the third part of our solution, namely, *synchronization points*. Finally, the SEPARATE REAL-TIME pattern proposes dividing a control system into different levels with different real-time requirements. This corresponds to the *isolation* part.

Known Uses

Xenomai. Xenomai [1] is a patch for Linux that introduces real-time capabilities. Figure 5 shows the basic concept of Xenomai. The patch places a so-called Ipipe between an interrupt from the hardware and the Linux interrupt handler. The Xenomai Ipipe checks whether a real time task is registered for the specific interrupt and subsequently calls the Xenomai Cobalt Core. The Xenomai Cobalt Core has access to own device drivers. Furthermore, it calls up the real time tasks and ensures their execution. After the real time tasks have been executed, the real Linux kernel with the generic tasks are carried out. Xenomai offers special socket types in order to enable the communication between the real time and non-real time tasks.

Xenomai itself incorporates the solution of this pattern. It separates the real time processing part strictly from the coordination part residing in ordinary Linux tasks. This corresponds to the third (Figure 2c) isolation method presented in the solution. It offers communication channels between these subsystems and synchronizes them through the Ipipe. The last part with synchronization points, however, needs to be implemented by the domain-specific real time processing tasks.

Self-adaptive software systems. Self-adaptive software systems are typically realized through an external (architecture) approach [7]. An internal approach interweaves application and adaptation logic

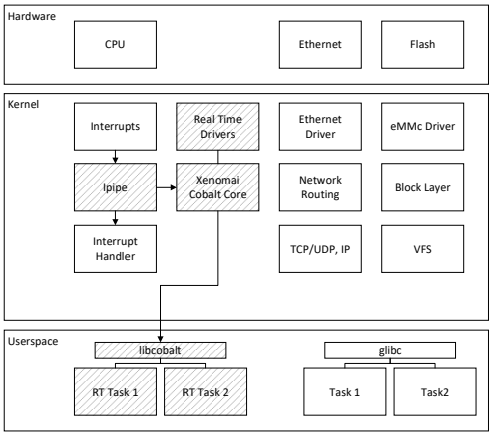


Figure 5: Illustration of Xenomai and where it intercepts interrupts.

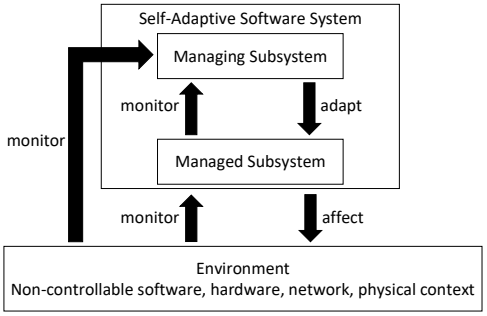


Figure 6: Parts of a self-adaptive software system (adapted from [9].

based on programming language features such as exceptions, conditions, and parametrization. The issue with an internal self-adaptive software system is that sensors, actuators, parallel adaptation processes and the actual purpose of an application are complicated to engineer within one single software design. This further leads to notable drawbacks, e.g. with respect to scalability, testability and maintainability.

In an external approach, as illustrated in Figure 6 [9], the processing part is isolated from the coordinating (adaptive) part. The domain-specific application logic, which is termed *Managed Subsystem* is monitored by a *Managing Subsystem*. The *Managing Subsystem* contains the adaptation logic. It additionally monitors the *Environment* that may consist of other software, hardware, network, or the physical context (including humans). On the basis of monitored data and analyzed problems, the *Managing Subsystem* itself decides whether and what to adapt inside the *Managed Subsystem*. Additionally, communication channels are necessary for observing and adjusting a *Managed Subsystem*. These adjustments cannot occur at arbitrary points in time because the *Managed Subsystem* needs to be in a known state. As a result synchronization points of some kind are needed. An exemplary self-adaptive system that realizes this pattern is Scari [4, 5]. It targets increasing the resilience of industrial cyber-physical systems against faults originating from hardware failures, security attacks, software bugs or misconfiguration.

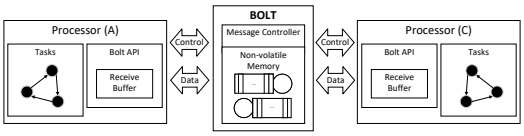


Figure 7: Overview of Bolt processor interconnect (adapted from [8]).

Bolt. Bolt is an ultra-low-power processor interconnect for the compositional construction of heterogeneous wireless embedded platforms [8]. Figure 7 illustrates this concept. Processor A is responsible for application tasks, while processor C is used for communication tasks. In the domain of wireless sensor networks, processor A would deal with sensing, e.g.
temperature, humidity, light, etc. Processor C would communicate the sensed data over a wireless communication channel with low power consumption. Bolt is a processor with a non-volatile memory that sits between these two processors and provides two message queues, one for each direction, in order to transfer data. The interface consists of a control and a data channel with a corresponding API for the application and communication tasks. The control channel coordinates the data channel and indicates the availability of data to the target processor. The Bolt benefit is that it solves the problem of processor A and C having different timing and power requirements which demand different wake up times and hardware power consumptions. For instance, processor A may continuously aggregate the environment temperature and processor C only wakes up once a day at a specific time to transmit it. Bolt eases the design and development of such an embedded device because it decouples these domains and reduces their potential interferences with each other.

The entire Bolt architecture represents an application of the presented pattern. The terms processing and coordination correspond to the application and communication tasks. The *isolation* part of the solution is realized through the separate application and communication processors. The *communication channels* part is applied by using the Bolt interface which allows the application and communication processors to be interchangeable and standardizes the communication channel. The message queues based on non-volatile memory enable the implementation of *synchronization points* where tasks can deal with data and adjustments fitting their timing and power requirements.

3 CONCLUSION

In this work we have presented a pattern that tackles the challenge of dealing with the processing and coordination parts of a system. In principle, it is applicable where a subsystem carries out the main purpose, while other arbitrary subsystems also exist that need to observe, adjust and protect the system. By applying this pattern, a system becomes simpler to develop, maintain and test. More importantly, it decreases negative impacts of subsystems on each other, e.g. concerning performance or security.

ACKNOWLEDGMENTS

We thank our shepherd Peter Scupelli for his inspiring and helpful comments. Furthermore, we thank our EuroPLoP 2018 focus group for giving very helpful hints and suggestions.

REFERENCES

[1] 2018. Website of the Xenomai Project. https://xenomai.org/.
[2] Veli-Pekka Eloranta, Johannes Koskinen, Marko Leppänen, and Ville Reijonen. 2010. A pattern language for distributed machine control systems. *Tampere University of Technology, Department of Software Systems* (2010).
[3] Brendan Galloway and Gerhard P. Hancke. 2013. Introduction to Industrial Control Networks. *IEEE Communications Surveys & Tutorials* 15, 2 (2013), 860–880. https://doi.org/10.1109/SURV.2012.071812.00124
[4] Johannes Iber, Michael Krisper, Jürgen Dobaj, and Christian Kreiner. 2018. Dynamic Adaption to Permanent Memory Faults in Industrial Control Systems. In *Proceedings of the 9th International Conference on Ambient Systems, Networks and Technologies (ANT '18)*. Elsevier. https://doi.org/10.1016/j.procs.2018.04.058
[5] Johannes Iber, Tobias Rauter, and Christian Kreiner. 2017. A Self-Adaptive Software System for Increasing the Reliability and Security of Cyber-Physical Systems. In *Solutions for Cyber-Physical Systems Ubiquity*. IGI Global. https://doi.org/10.4018/978-1-5225-2845-6.ch009
[6] M Douglas McIlroy, J Buxton, Peter Naur, and Brian Randell. 1968. Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*. 88–98.
[7] Mazeiar Salehie and Ladan Tahvildari. 2009. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems* 4, 2 (may 2009), 1–42. https://doi.org/10.1145/1516533.1516538
[8] Felix Sutton, Marco Zimmerling, Reto Da Forno, Roman Lim, Tonio Gsell, Georgia Giannopoulou, Federico Ferrari, Jan Beutel, and Lothar Thiele. 2015. Bolt: A Stateful Processor Interconnect. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems (SenSys '15)*. ACM, New York, NY, USA, 267–280. https://doi.org/10.1145/2809695.2809706
[9] Danny Weyns, Bradley Schmerl, Vincenzo Grassi, Sam Malek, Raffaela Mirandola, Christian Prehofer, Jochen Wuttke, Jesper Andersson, Holger Giese, and Karl M. Göschka. 2013. On Patterns for Decentralized Control in Self-Adaptive Systems. In *Software Engineering for Self-Adaptive Systems II*. https://doi.org/10.1007/978-3-642-35813-5_4

Poster: Towards a Secure, Resilient, and Distributed Infrastructure for Hydropower Plant Unit Control

Andrea Höller, Johannes Iber, Tobias Rauter, and Christian Kreiner
Graz University of Technology
{firstname.lastname}@tugraz.at

Abstract

Today, there are ever increasing demands on hydro-electrical power plant controllers. They have to deal with a power grid that becomes ever more unpredictable due to renewable energies. Furthermore, power plants represent a critical infrastructure that have to provide a full operation, even in the presence of cyber-attacks and internal faults.

Here, we present an approach towards tackling these challenges by integrating the knowledge of multiple research domains such as security, fault tolerance and modeling. We propose a distributed infrastructure that provides resilience via an assured dynamic self-adaption. Further, we show the integration into an existing hydropower plant unit control.

1 Introduction

For nearly a century the electrical protection, the generator voltage regulation and synchronization with the power grid of hydropower plant units was done by specialized mechanical and electromechanical devices. This is currently changing, because other renewable energy sources like wind or solar are integrated into the power grid on a grand scale [4]. Their complicated predictability concerning energy conversion has an impact on the technology of hydropower plant unit control systems because nowadays these have to react on power grid changes in time to achieve overall grid stability. At the same time, these power plants represent critical infrastructures that have to be protected against cyber-security attacks. Furthermore, a shutdown of the plant due to faults in the control device can lead to economic losses, and even large scale blackouts.

To tackle this challenge, we perform research together with our industrial partner on how to create future resilient and secure power plant controllers. As far as the authors of this work know, self-adaptability has not been applied in this domain so far. We combine research from different fields like security, fault tolerance, and models@run.time to enhance the resilience of an existing control system. These proposed principles are also applicable for cyber-physical systems in other domains.

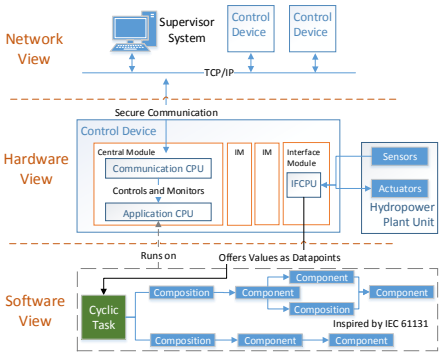


Figure 1. Overview of the existing system

2 Existing Hydropower Plant Control System

Figure 1 illustrates the existing target system of our proposed infrastructure. On network level the control devices are operated by a so-called supervisor system and connected by wire. The supervisor system is mainly responsible for deploying tasks on devices and collecting data from the single hydropower plant control devices.

The control devices are connected to hydropower plant units. Their functional responsibility is to operate hydropower plant units through the different phases excitation, synchronization, protection and turbine control.

Technically, these control devices have a programmable logic controller (PLC) architecture. Concerning the hardware, a control device is build out of central modules and interface modules. A central module consists of a communication CPU and an application CPU. The communication CPU is responsible for network connections and controlling/monitoring the application CPU, and runs a customized Linux. From the security point of view it also acts as gateway for the application CPU. The application CPU is a multi-core

International Conference on Embedded Wireless Systems and Networks (EWSN) 2016
15–17 February, Graz, Austria

ISBN: 978-0-9949886-0-7

processor and executes the actual logic. It runs a real-time operating system in order to ensure guaranteed cycle times. The interface modules are connecting the control device with sensors and actuators of the hydropower plant unit.

The software executed by the application CPU of a central module is component-based and inspired by the IEC 61131 standard for programmable controllers. Basically, the software is hierarchically build out of components, compositions and tasks. Components are coded with the C-programming language and stored as binaries on the devices. Such components implement basic function blocks, e.g. simple logic gates, or complex control algorithms. Based on these components, compositions are designed that implement the specific control logic for a hydropower plant unit. Finally, such compositions are called by cyclic tasks.

The compositions operate on so-called datapoints that are set and read by the interface modules. At the start of a cyclic task the necessary datapoints are collected, the compositions are executed, and subsequently the calculated datapoints are written back.

From the plant engineer's point of view this system is configured with model-driven techniques. Domain-specific modeling languages (DSL) are used to describe tasks, compositions, and components. As well, resources consisting of modules, control devices, networks, and connected hydropower plant units are modeled. Finally, deployment of tasks onto control devices is specified by yet another DSL. Concerning extra-functional properties (e.g. timing, security, memory consumption, ...), we rely on contract-based design [3]. All these models are traditionally used at design time. Additionally, we are going to leverage them at runtime for the infrastructure proposed.

3 Proposed Infrastructure

We enhance the control device with security and hardware fault monitoring. Observed anomalies are forwarded to the supervisor running a reasoner application that interprets these anomalies and executes mitigation strategies.

3.1 Control Device

The control device has to fulfill high performance requirements, since a lot of sensor data has to be processed in short time. However, the nature of the control application is well-suited for tuning the performance with parallelization. Considering dependability, the use of highly integrated multi-core is a double-edged sword: future semiconductor technologies are expected to be very susceptible to hardware errors due to small structures, however additional processing units also allow the integration of additional fault tolerance techniques.

More precisely, we exploit the unused computing capacity to enhance the diagnostic features regarding hardware faults with spatial redundancy. Cores that are not needed to realize the main functionality are used to perform partial redundant calculations, cross-checking the achieved results, and analyzing the trend of observed errors. This error trend is used to distinguish between transient and permanent faults.

In order to increase the diagnostic capabilities, the redundant calculations are done in a diverse way. For introducing the diversity in the redundant replicas, we use cost-efficient ways to automatically introduce diversity in execution as proposed in [2]. For example, different compiler and compiler flags can be used to generate several binaries that perform the same calculations, but show different execution characteristics (e.g. timing, hardware resource usage). We have shown that this approach is well-suited to detect common-cause faults such as memory-related software bugs, or hardware faults in shared resources [1].

Furthermore, we enhance the resilience regarding malicious security attacks. First, the diverse redundancy concept expenses the required effort to attack the functional application, since all redundant executions have to be attacked simultaneously. Anomaly-detection is performed on the communication CPU. This includes checking the plausibility of sensor data, network anomalies, and the integrity of other devices in the network [5].

3.2 Reasoner

The reasoner manages a model of the system that includes functional and extra-functional properties of the target application [3]. Furthermore, it receives information about detected anomalies from the control device. By analyzing these data, the reasoner decides, whether and how to reconfigure the system. For example, if hardware or software faults affect the control device a fault recovery strategy is executed. This fault recovery is realized by updating the component binaries with diverse binary versions [2]. The system model is used to assure that the extra-functional requirements (e.g. timing/memory constraints) are still fulfilled after the update.

If these strategies are not successful, the system is reconfigured in such a way that the faulty device is isolated or the control logic is redistributed. Again, the system model is used to assure a correct reconfiguration. Furthermore, the reasoner includes a detection mechanism that distinguishes between non-malicious faults (e.g. software- or hardware faults) and malicious faults. If it is possible to identify the attacked part of the system, this part is isolated. Every reconfiguration alarms the power plant operator.

4 Conclusions

The primary aim of our work is to enhance the security and resilience of hydropower plant unit controls. We propose a distributed infrastructure which reconfigures itself based on the observed hardware faults or security violations. This mechanism is going to leverage models at runtime which are a product of the design-time configuration.

5 References

[1] A. Höller, N. Kajtazovic, K. Römer, and C. Kreiner. Evaluation of Diverse Compiling for Software Fault Tolerance. In *DATE*, 2015.

[2] A. Höller, T. Rauter, J. Iber, and C. Kreiner. Software-Based Fault Recovery via Adaptive Diversity for COTS Multi-Core Processors. In *ADAPT Workshop*, 2016.

[3] J. Iber, A. Höller, T. Rauter, and C. Kreiner. Towards a Generic Modeling Language for Contract-Based Design. In *ModComp Workshop (MoDELS)*, 2015.

[4] M. Liserre, T. Sauter, and J. Hung. Future Energy Systems: Integrating Renewable Energy Sources into the Smart Power Grid Through Industrial Electronics. *IEEE Industrial Electronics Magazine*, 4, 2010.

[5] T. Rauter, A. Höller, J. Iber, and C. Kreiner. Thingtegrity: A Scalable Trusted Computing Architecture for Resource Constrained Devices. In *EWSN*, 2016. to appear.

A. Appendix

Listing A.1: The ScariObject of type Entity serialized to a JSON file.


```
{
    "containerName": "",
    "containerType": "",
    "objectName": "{48853f35-d575-429c-ae8a-a1881b6dba38}",
    "objectType": "Entity",
    "properties": [
        {
            "dataType": "bool",
            "nameInfo": "active",
            "typeInfo": "ATTRIBUTE_ONE",
            "value": "true"
        },
        {
            "dataType": "int",
            "nameInfo": "values",
            "typeInfo": "ATTRIBUTE_N",
            "value": [
                "42"
            ]
        },
        {
            "dataType": "Entity",
            "nameInfo": "otherEntity",
            "typeInfo": "REFERENCE_ONE",
            "value": "{4598d07e-9c5c-4e33-9893-b3683e1e3dc1}"
        },
        {
            "dataType": "Entity",
            "nameInfo": "otherEntities",
            "typeInfo": "REFERENCE_N",
            "value": [
                "{4598d07e-9c5c-4e33-9893-b3683e1e3dc1}"
            ]
        },
        {
            "dataType": "Entity",
            "nameInfo": "ownedEntity",
            "typeInfo": "COMPOSITION_ONE",
            "value": {
                "containerName": "{48853f35-d575-429c-ae8a-a1881b6dba38}",
                "containerType": "Entity",
                "objectName": "{7ab7144d-bf8a-47ce-9d50-768c1c894d6c}",
                "objectType": "Entity",
                "properties": [
                    {
                        "dataType": "bool",
```

```
                    "nameInfo": "active",
                    "typeInfo": "ATTRIBUTE_ONE",
                    "value": "false"
                },
                {
                    "dataType": "int",
                    "nameInfo": "values",
                    "typeInfo": "ATTRIBUTE_N",
                    "value": [
                    ]
                },
                {
                    "dataType": "Entity",
                    "nameInfo": "otherEntity",
                    "typeInfo": "REFERENCE_ONE",
                    "value": ""
                },
                {
                    "dataType": "Entity",
                    "nameInfo": "otherEntities",
                    "typeInfo": "REFERENCE_N",
                    "value": [
                    ]
                },
                {
                    "dataType": "Entity",
                    "nameInfo": "ownedEntity",
                    "typeInfo": "COMPOSITION_ONE",
                    "value": {
                    }
                },
                {
                    "dataType": "Entity",
                    "nameInfo": "ownedEntities",
                    "typeInfo": "COMPOSITION_N",
                    "value": [
                    ]
                }
            ]
        }
    },
    {
        "dataType": "Entity",
        "nameInfo": "ownedEntities",
        "typeInfo": "COMPOSITION_N",
        "value": [
            {
                "containerName": "{48853f35-d575-429c-ae8a-a1881b6dba38}",
                "containerType": "Entity",
                "objectName": "{7b525b5e-7e6e-41ee-abc1-b65a5fd22b12}",
                "objectType": "Entity",
                "properties": [
                    {
                        "dataType": "bool",
                        "nameInfo": "active",
                        "typeInfo": "ATTRIBUTE_ONE",
                        "value": "false"
                    },
```

```
                        {
                            "dataType": "int",
                            "nameInfo": "values",
                            "typeInfo": "ATTRIBUTE_N",
                            "value": [
                            ]
                        },
                        {
                            "dataType": "Entity",
                            "nameInfo": "otherEntity",
                            "typeInfo": "REFERENCE_ONE",
                            "value": ""
                        },
                        {
                            "dataType": "Entity",
                            "nameInfo": "otherEntities",
                            "typeInfo": "REFERENCE_N",
                            "value": [
                            ]
                        },
                        {
                            "dataType": "Entity",
                            "nameInfo": "ownedEntity",
                            "typeInfo": "COMPOSITION_ONE",
                            "value": {
                            }
                        },
                        {
                            "dataType": "Entity",
                            "nameInfo": "ownedEntities",
                            "typeInfo": "COMPOSITION_N",
                            "value": [
                            ]
                        }
                    ]
                }
            ]
        }
    ]
}
```

Bibliography

[1] I Crnkovic. *Building Reliable Component-Based Software Systems*. Ed. by M Larsson. Artech House, Inc., 2002. ISBN: 1580533272.

[2] IE Commission et al. "IEC 61131-3." In: *Programmable Controllers-Part* 3 (2013).

[3] R Mackiewicz. "Overview of IEC 61850 and Benefits." In: *2005/2006 PES TD*. IEEE. ISBN: 0-7803-9194-2. DOI: 10.1109/TDC.2006.1668522.

[4] M Liserre; T Sauter; and J Hung. "Future Energy Systems: Integrating Renewable Energy Sources into the Smart Power Grid Through Industrial Electronics." In: *IEEE Industrial Electronics Magazine* 4 (2010). ISSN: 1932-4529. DOI: 10.1109/MIE.2010.935861.

[5] S Yin and O Kaynak. "Big Data for Modern Industry: Challenges and Trends [Point of View]." In: *Proceedings of the IEEE* 103.2 (2015). ISSN: 0018-9219. DOI: 10.1109/JPROC.2015.2388958.

[6] K Stouffer; V Pillitteri; S Lightman; M Abrams; and A Hahn. *Guide to Industrial Control Systems (ICS) Security*. Tech. rep. National Institute of Standards and Technology, 2015. DOI: 10.6028/NIST.SP.800-82r2.

[7] National Cybersecurity and Communications Integration Center and Industrial Control Systems Cyber Emergency Response Team. *NCCIC / ICS-CERT Year in Review*. Tech. rep. 2010.

[8] National Cybersecurity and Communications Integration Center and Industrial Control Systems Cyber Emergency Response Team. *NCCIC / ICS-CERT Year in Review*. Tech. rep. 2016.

[9] J-C Laprie. "From dependability to resilience." In: *International Conference on Dependable Systems and Networks*. 2008. ISBN: 1471-2490. DOI: http://dx.doi.org/10.1016/j.yexcr.2010.06.008.

[10] M Brambilla; J Cabot; and M Wimmer. "Model-Driven Software Engineering in Practice." In: *Synthesis Lectures on Software Engineering* 1.1 (2012). ISSN: 2328-3319. DOI: 10.2200/S00441ED1V01Y201208SWE001.

[11] J Bézivin. "In search of a basic principle for model driven engineering." In: *Novatica Journal, Special Issue* 5.2 (2004).

[12] AG Kleppe; J Warmer; and W Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 032119442X.

[13] J Hutchinson; J Whittle; M Rouncefield; and S Kristoffersen. "Empirical assessment of MDE in industry." In: *Proceeding of the 33rd international conference on Software engineering - ICSE '11*. ICSE '11. ACM Press, 2011. ISBN: 9781450304450. DOI: 10.1145/1985793.1985858.

[14] J Whittle and J Hutchinson. "Mismatches between Industry Practice and Teaching of Model-Driven Software Development." In: *Models in Software Engineering SE - 6*. Ed. by J Kienzle. Vol. 7167. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012. ISBN: 978-3-642-29644-4. DOI: 10.1007/978-3-642-29645-1_6.

[15] J Whittle; J Hutchinson; and M Rouncefield. "The State of Practice in Model-Driven Engineering." In: *IEEE Software* 31.3 (2014). ISSN: 0740-7459. DOI: 10.1109/MS.2013.65.

[16] T Kühne. "Matters of (Meta-) Modeling." English. In: *Software & Systems Modeling* 5.4 (July 2006). ISSN: 1619-1366. DOI: 10.1007/s10270-006-0017-9.

[17] Object Management Group (OMG). *OMG UML, Infrastructure Version 2.4.1*. 2011. URL: http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF/.

[18] Object Management Group (OMG). *OMG Meta Object Facility (MOF) Core Specification Version 2.4.1*. 2013. URL: http://www.omg.org/spec/MOF/2.4.1/PDF/.

[19] A Sangiovanni-Vincentelli; W Damm; and R Passerone. "Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems." In: *European Journal of Control* 18.3 (Jan. 2012). ISSN: 09473580. DOI: 10.3166/ejc.18.217-238.

[20] P Nuzzo; Huan Xu; N Ozay; JB Finn; AL Sangiovanni-Vincentelli; RM Murray; A Donze; and SA Seshia. "A Contract-Based Methodology for Aircraft Electric Power System Design." In: *IEEE Access* 2 (2014). ISSN: 2169-3536. DOI: 10.1109/ACCESS.2013.2295764.

[21] A Benveniste; B Caillaud; A Ferrari; L Mangeruca; R Passerone; and C Sofronis. "Multiple Viewpoint Contract-Based Specification and Design." In: 2008. DOI: 10.1007/978-3-540-92188-2_9.

[22] B Meyer. "Applying 'design by contract'." In: *Computer* 25.10 (Oct. 1992). ISSN: 0018-9162. DOI: 10.1109/2.161279.

[23] A Benveniste; B Caillaud; D Nickovic; R Passerone; J-B Raclet; P Reinkemeier; AL Sangiovanni-Vincentelli; W Damm; T Henzinger; and K Larsen. *Contracts for Systems Design*. Tech. rep. INRIA, 2012.

[24] N Kajtazovic. "A Component-based Approach for Managing Changes in the Engineering of Safety-critical Embedded Systems." PhD thesis. Graz University of Technology, 2014.

[25] A Rajan and T Wahl, eds. *CESAR - Cost-efficient Methods and Processes for Safety-relevant Embedded Systems*. Springer Vienna, 2013. ISBN: 978-3-7091-1386-8. DOI: 10.1007/978-3-7091-1387-5.

[26] A Benveniste; B Caillaud; D Nickovic; R Passerone; J-B Raclet; P Reinkemeier; A Sangiovanni-Vincentelli; W Damm; T Henzinger; and KG Larsen. *Contracts for Systems Design: Theory*. Research Report RR-8759. Inria Rennes Bretagne Atlantique ; INRIA, 2015.

[27] K Vanherpen. "A contract-based approach for multi-viewpoint consistency in the concurrent design of cyber-physical systems." PhD thesis. University of Antwerp, 2018.

[28] J Kephart and D Chess. "The vision of autonomic computing." In: *Computer* 36.1 (2003). ISSN: 0018-9162. DOI: 10.1109/MC.2003.1160055.

[29] S Dobson; R Sterritt; P Nixon; and M Hinchey. "Fulfilling the Vision of Autonomic Computing." In: *Computer* 43.1 (2010). ISSN: 0018-9162. DOI: 10.1109/MC.2010.14.

[30] M Salehie and L Tahvildari. "Self-adaptive software: Landscape and research challenges." In: *ACM Transactions on Autonomous and Adaptive Systems* 4.2 (2009). ISSN: 15564665. DOI: 10.1145/1516533.1516538.

[31] P Oreizy; M Gorlick; R Taylor; D Heimhigner; G Johnson; N Medvidovic; A Quilici; D Rosenblum; and A Wolf. "An architecture-based approach to self-adaptive software." In: *IEEE Intelligent Systems* 14.3 (1999). ISSN: 1094-7167. DOI: 10.1109/5254.769885.

[32] R Laddaga. "Active Software." In: *Self-Adaptive Software: First International Workshop, IWSAS 2000 Oxford, UK, April 17–19, 2000 Revised Papers*. Ed. by P Robertson; H Shrobe; and R Laddaga. Springer Berlin Heidelberg, 2001. ISBN: 978-3-540-44584-5. DOI: 10.1007/3-540-44584-6_2.

[33] D Weyns; B Schmerl; V Grassi; S Malek; R Mirandola; C Prehofer; J Wuttke; J Andersson; H Giese; and KM Göschka. "On Patterns for Decentralized Control in Self-Adaptive Systems." In: *Software Engineering for Self-Adaptive Systems II*. 2013. DOI: 10.1007/978-3-642-35813-5_4.

[34] M Hinchey and R Sterritt. "Self-Managing Software." In: *Computer* 39.2 (2006). ISSN: 0018-9162. DOI: 10.1109/MC.2006.69.

[35] H Muccini; M Sharaf; and D Weyns. "Self-adaptation for Cyber-physical Systems: A Systematic Literature Review." In: *Proceedings of the 11th International Workshop on Software Engineering for Adaptive and Self-Managing Systems - SEAMS '16*. ACM Press, 2016. ISBN: 9781450341875. DOI: 10.1145/2897053.2897069.

[36] JP Müller and K Fischer. "Application Impact of Multi-agent Systems and Technologies: A Survey." In: *Agent-Oriented Software Engineering*. Springer Berlin Heidelberg, 2014. DOI: 10.1007/978-3-642-54432-3_3.

[37] M Jelasity; O Babaoglu; R Laddaga; R Nagpal; F Zambonelli; E Sirer; H Chaouchi; and M Smirnov. "Interdisciplinary Research: Roles for Self-Organization." In: *IEEE Intelligent Systems* 21.2 (2006). ISSN: 1541-1672. DOI: 10.1109/MIS.2006.30.

[38] Y Brun; R Desmarais; K Geihs; M Litoiu; A Lopes; M Shaw; and M Smit. "A Design Space for Self-Adaptive Systems." In: *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*. Ed. by R de Lemos; H Giese; HA Müller; and M Shaw. Springer Berlin Heidelberg, 2013. ISBN: 978-3-642-35813-5. DOI: 10.1007/978-3-642-35813-5_2.

[39] Wikimedia Commons. *OODA loop*. 2014. URL: https://commons.wikimedia.org/wiki/File:OODA.Boyd.svg.

[40] JR Boyd. *The Essence of Winning and Losing*. 1996. URL: http://dnipogo.org/john-r-boyd/.

[41] S Grenander; K Simpson; and O Sindiy. "The Autonomy System Architecture." In: *AIAA Infotech@Aerospace Conference*. Infotech@Aerospace Conferences. American Institute of Aeronautics and Astronautics, 2009. ISBN: 978-1-60086-979-2. DOI: 10.2514/6.2009-1884.

[42] A Chandra; PR Lewis; K Glette; and SC Stilkerich. "Reference Architecture for Self-aware and Self-expressive Computing Systems." In: *Self-aware Computing Systems: An Engineering Approach*. Ed. by PR Lewis; M Platzner; B Rinner; J Tørresen; and X Yao. Springer International Publishing, 2016. ISBN: 978-3-319-39675-0. DOI: 10.1007/978-3-319-39675-0_4.

[43] S Dobson; F Zambonelli; S Denazis; A Fernández; D Gaïti; E Gelenbe; F Massacci; P Nixon; F Saffre; and N Schmidt. "A survey of autonomic communications." In: *ACM Transactions on Autonomous and Adaptive Systems* 1.2 (2006). ISSN: 15564665. DOI: 10.1145/1186778.1186782.

[44] BHC Cheng; R de Lemos; H Giese; P Inverardi; J Magee; J Andersson; B Becker; N Bencomo; Y Brun; B Cukic; G Di Marzo Serugendo; S Dustdar; A Finkelstein; C Gacek; K Geihs; V Grassi; G Karsai; HM Kienle; J Kramer; M Litoiu; S Malek;

R Mirandola; HA Müller; S Park; M Shaw; M Tichy; M Tivoli; D Weyns; and J Whittle. "Software Engineering for Self-Adaptive Systems: A Research Roadmap." In: *Software Engineering for Self-Adaptive Systems*. Ed. by BHC Cheng; R de Lemos; H Giese; P Inverardi; and J Magee. Springer Berlin Heidelberg, 2009. DOI: 10.1007/978-3-642-02161-9_1.

[45] G Blair; N Bencomo; and RB France. "Models@ run.time." In: *Computer* 42.10 (2009). ISSN: 0018-9162. DOI: 10.1109/MC.2009.326.

[46] Object Management Group (OMG). *Website of the Unified Modeling Language*. 2018. URL: http://uml.org/.

[47] H Giese; N Bencomo; L Pasquale; AJ Ramirez; P Inverardi; S Wätzoldt; and S Clarke. "Living with Uncertainty in the Age of Runtime Models." In: *Models@run.time: Foundations, Applications, and Roadmaps*. Ed. by N Bencomo; R France; BHC Cheng; and U Aßmann. Springer International Publishing, 2014. ISBN: 978-3-319-08915-7. DOI: 10.1007/978-3-319-08915-7_3.

[48] M Maasoumy; P Nuzzo; and A Sangiovanni-Vincentelli. "Smart Buildings in the Smart Grid: Contract-Based Design of an Integrated Energy Management System." In: *Cyber Physical Systems Approach to Smart Electric Power Grid*. Springer Berlin Heidelberg, 2015. DOI: 10.1007/978-3-662-45928-7_5.

[49] P Nuzzo; A Sangiovanni-Vincentelli; Xuening Sun; and A Puggelli. "Methodology for the Design of Analog Integrated Interfaces Using Contracts." In: *IEEE Sensors Journal* 12.12 (Dec. 2012). ISSN: 1530-437X. DOI: 10.1109/JSEN.2012.2211098.

[50] N Kajtazovic; C Preschern; A Höller; and C Kreiner. "Constraint-Based Verification of Compositions in Safety-Critical Component-Based Systems." In: *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*. Studies in Computational Intelligence. Springer International Publishing, 2015. ISBN: 978-3-319-10388-4. DOI: 10.1007/978-3-319-10389-1_9.

[51] T Amorim; D Ratasich; G Macher; A Ruiz; D Schneider; M Driussi; and R Grosu. "Runtime Safety Assurance for Adaptive Cyber-Physical Systems: ConSerts M and Ontology-Based Runtime Reconfiguration Applied to an Automotive Case Study Runtime Safety Assurance for Adaptive Cyber-Physical Systems: ConSerts M and Ontology-Based Runtime Reconfiguration Applied to an Automotive Case Study." In: *Solutions for Cyber-Physical Systems Ubiquity*. Ed. by N Druml; A Genser; A Krieg; M Menghin; and A Höller. IGI Global, 2018. DOI: 10.4018/978-1-5225-2845-6.ch006.

[52] P Nuzzo and A Sangiovanni-Vincentelli. "Let's Get Physical: Computer Science Meets Systems." In: *From Programs to Systems. The Systems perspective in Com-*

puting. Springer Berlin Heidelberg, 2014. ISBN: 978-3-642-54847-5, 978-3-642-54848-2. DOI: 10.1007/978-3-642-54848-2_13.

[53] F Warg; B Vedder; M Skoglund; and A Soderberg. "Safety ADD: A Tool for Safety-Contract Based Design." In: *2014 IEEE International Symposium on Software Reliability Engineering Workshops*. Nov. 2014. ISBN: 978-1-4799-7377-4. DOI: 10.1109/ISSREW.2014.18.

[54] M Sievers and AM Madni. "A flexible contracts approach to system resiliency." In: *2014 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE, 2014. ISBN: 978-1-4799-3840-7. DOI: 10.1109/SMC.2014.6974044.

[55] Object Management Group (OMG). *OMG SysML Version 1.3*. 2012. URL: http://www.omg.org/spec/SysML/1.3/.

[56] M Grabowski; B Kaiser; and Y Bai. "Systematic Refinement of CPS Requirements using SysML, Template Language and Contracts." In: *Modellierung 2018*. Ed. by I Schaefer; D Karagiannis; A Vogelsang; D Méndez; and C Seidl. Gesellschaft für Informatik e.V., 2018.

[57] The CHESS Project. *Website of the CHESS Project*. 2018. URL: http://www.chess-project.org.

[58] Object Management Group (OMG). *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems Version 1.1*. 2011. URL: http://www.omg.org/spec/MARTE/.

[59] PolarSys. *CHESS Toolset User Guide*. 2018. URL: https://www.polarsys.org/chess/publis/CHESSToolset_UserGuide.pdf.

[60] T Amorim; A Ruiz; C Dropmann; and D Schneider. "Multidirectional Modular Conditional Safety Certificates." In: 2015. DOI: 10.1007/978-3-319-24249-1_31.

[61] T Amorim; D Ratasich; G Macher; A Ruiz; D Schneider; M Driussi; and R Grosu. "Runtime Safety Assurance for Adaptive Cyber-Physical Systems: ConSerts M and Ontology-Based Runtime Reconfiguration Applied to an Automotive Case Study." In: *Solutions for Cyber-Physical Systems Ubiquity*. IGI Global, 2018. DOI: 10.4018/978-1-5225-2845-6.ch006.

[62] B Zimmer; S Bürklen; M Knoop; J Höfflinger; and M Trapp. "Vertical Safety Interfaces – Improving the Efficiency of Modular Certification." In: 2011. DOI: 10.1007/978-3-642-24270-0_3.

[63] C Krupitzer; FM Roth; S VanSyckel; G Schiele; and C Becker. "A survey on engineering approaches for self-adaptive systems." In: *Pervasive and Mobile Computing* 17 (2015). ISSN: 15741192. DOI: 10.1016/j.pmcj.2014.09.009.

[64] C Krupitzer; FM Roth; M Pfannemüller; and C Becker. *Comparison of approaches for self-improvement in self-adaptive systems (extended version)*. Englisch. 2017.

[65] FD Macías-Escrivá; R Haber; R del Toro; and V Hernandez. "Self-adaptive systems: A survey of current approaches, research challenges and applications." In: *Expert Systems with Applications* 40.18 (2013). ISSN: 09574174. DOI: 10.1016/j.eswa.2013.07.033.

[66] JP Loyall; DE Bakken; RE Schantz; JA Zinky; DA Karr; R Vanegas; and KR Anderson. "QoS Aspect Languages and Their Runtime Integration." In: 1998. DOI: 10.1007/3-540-49530-4_22.

[67] K Appleby; S Fakhouri; L Fong; G Goldszmidt; M Kalantar; S Krishnakumar; D Pazel; J Pershing; and B Rochwerger. "Oceano-SLA based management of a computing utility." In: *2001 IEEE/IFIP International Symposium on Integrated Network Management Proceedings. Integrated Network Management VII. Integrated Management Strategies for the New Millennium (Cat. No.01EX470)*. IEEE. ISBN: 0-7803-6719-7. DOI: 10.1109/INM.2001.918085.

[68] S-W Cheng. "Rainbow: Cost-effective Software Architecture-based Self-adaptation." PhD thesis. 2008. ISBN: 978-0-549-52525-7.

[69] S Tuttle; V Batchellor; MB Hansen; and M Sethuraman. "Centralized risk management using tivoli risk manager 4.2." In: *IBM Redbooks* (2003).

[70] G Kaiser; J Parekh; P Gross; and G Valetto. "Kinesthetics eXtreme: an external infrastructure for monitoring distributed legacy systems." In: *2003 Autonomic Computing Workshop*. IEEE Comput. Soc. ISBN: 0-7695-1983-0. DOI: 10.1109/ACW.2003.1210200.

[71] Hua Liu; M Parashar; and S Hariri. "A Component Based Programming Framework for Autonomic Applications." In: *International Conference on Autonomic Computing, 2004. Proceedings.* IEEE. ISBN: 0-7695-2114-2. DOI: 10.1109/ICAC.2004.1301341.

[72] G Candea; E Kiciman; S Kawamoto; and A Fox. "Autonomous recovery in componentized Internet applications." In: *Cluster Computing* 9.2 (2006). ISSN: 1386-7857. DOI: 10.1007/s10586-006-7562-4.

[73] SM Sadjadi; PK McKinley; BHC Cheng; and REK Stirewalt. "TRAP/J: Transparent Generation of Adaptable Java Programs." In: 2004. DOI: 10.1007/978-3-540-30469-2_28.

[74] J Dowling. "The Decentralised Coordination of Self-Adaptive Components for Autonomic Distributed Systems." PhD thesis. Trinity College, 2004.

[75] P Robertson and R Laddaga. "Model Based Diagnosis and Contexts in Self Adaptive Software." In: 2005. DOI: 10.1007/11428589_8.

[76] A Mukhija and M Glinz. "Runtime Adaptation of Applications Through Dynamic Recomposition of Components." In: 2005. DOI: 10.1007/978-3-540-31967-2_9.

[77] J White; DC Schmidt; and A Gokhale. "Simplifying autonomic enterprise Java Bean applications via model-driven engineering and simulation." In: *Software & Systems Modeling* 7.1 (2007). ISSN: 1619-1366. DOI: 10.1007/s10270-007-0057-9.

[78] A Lapouchnian; S Liaskos; J Mylopoulos; and Y Yu. "Towards requirements-driven autonomic systems design." In: *ACM SIGSOFT Software Engineering Notes* 30.4 (2005). ISSN: 01635948. DOI: 10.1145/1082983.1083075.

[79] J Floch; S Hallsteinsen; E Stav; F Eliassen; K Lund; and E Gjorven. "Using architecture models for runtime adaptability." In: *IEEE Software* 23.2 (2006). ISSN: 0740-7459. DOI: 10.1109/MS.2006.61.

[80] V Kumar; BF Cooper; Z Cai; G Eisenhauer; and K Schwan. "Middleware for enterprise scale data stream management using utility-driven self-adaptive information flows." In: *Cluster Computing* 10.4 (2007). ISSN: 1386-7857. DOI: 10.1007/s10586-007-0040-9.

[81] Y Al-Nashif; AA Kumar; S Hariri; Y Luo; F Szidarovsky; and G Qu. "Multi-Level Intrusion Detection System (ML-IDS)." In: *2008 International Conference on Autonomic Computing*. IEEE, 2008. DOI: 10.1109/ICAC.2008.25.

[82] H Tajalli; J Garcia; G Edwards; and N Medvidovic. "PLASMA: a plan-based layered architecture for software model-driven adaptation." In: *Proceedings of the IEEE/ACM international conference on Automated software engineering - ASE '10*. ACM Press, 2010. ISBN: 9781450301169. DOI: 10.1145/1858996.1859092.

[83] P Brittenham; RR Cutlip; C Draper; BA Miller; S Choudhary; and M Perazolo. "IT service management architecture and autonomic computing." In: *IBM Systems Journal* 46.3 (2007). ISSN: 0018-8670. DOI: 10.1147/sj.463.0565.

[84] B Morin; O Barais; J-M Jezequel; F Fleurey; and A Solberg. "Models@ Run.time to Support Dynamic Adaptation." In: *Computer* 42.10 (2009). ISSN: 0018-9162. DOI: 10.1109/MC.2009.327.

[85] G Jung; MA Hiltunen; KR Joshi; RD Schlichting; and C Pu. "Mistral: Dynamically Managing Power, Performance, and Adaptation Cost in Cloud Infrastructures." In: *2010 IEEE 30th International Conference on Distributed Computing Systems*. IEEE, 2010. ISBN: 978-1-4244-7261-1. DOI: 10.1109/ICDCS.2010.88.

[86] D Cooray; E Kouroshfar; S Malek; and R Roshandel. "Proactive Self-Adaptation for Improving the Reliability of Mission-Critical, Embedded, and Mobile Software." In: *IEEE Transactions on Software Engineering* 39.12 (2013). ISSN: 0098-5589. DOI: 10.1109/TSE.2013.36.

[87] E Albassam; J Porter; H Gomaa; and DA Menasce. "DARE: A Distributed Adaptation and Failure Recovery Framework for Software Systems." In: *2017 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, 2017. ISBN: 978-1-5386-1762-5. DOI: 10.1109/ICAC.2017.12.

[88] D Weyns; MU Iftikhar; D Hughes; and N Matthys. "Applying Architecture-Based Adaptation to Automate the Management of Internet-of-Things." In: 2018. DOI: 10.1007/978-3-030-00761-4_4.

[89] Google. *Website of the Kubernetes Project*. 2018. URL: https://kubernetes.io.

[90] D Weyns and T Ahmad. "Claims and Evidence for Architecture-Based Self-adaptation: A Systematic Literature Review." In: *European Conference on Software Architecture*. Springer Berlin Heidelberg, 2013. DOI: 10.1007/978-3-642-39031-9_22.

[91] D Weyns; MU Iftikhar; S Malek; and J Andersson. "Claims and Supporting Evidence for Self-adaptive Systems: A Literature Study." In: *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '12. IEEE Press, 2012. ISBN: 978-1-4673-1787-0.

[92] KH John and M Tiegelkamp. *IEC 61131-3: Programming Industrial Automation Systems*. Springer Berlin Heidelberg, 2010. ISBN: 978-3-642-12014-5. DOI: 10.1007/978-3-642-12015-2.

[93] J Bézivin; F Jouault; and P Valduriez. "On the need for megamodels." In: *Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 2004.

[94] R Langner. "Stuxnet: Dissecting a Cyberwarfare Weapon." In: *IEEE Security & Privacy Magazine* 9.3 (2011). ISSN: 1540-7993. DOI: 10.1109/MSP.2011.67.

[95] N Falliere; LO Murchu; and E Chien. "W32. stuxnet dossier." In: *White paper, Symantec Corp., Security Response* 5.6 (2011).

[96] Eclipse Foundation. *Website of the EMF Project*. 2018. URL: http://www.eclipse.org/modeling/emf/.

[97] Eclipse Foundation. *Website of the Xtext Project*. 2018. URL: http://www.eclipse.org/Xtext/.

[98] C Schneider; M Spönemann; and R von Hanxleden. "Just model! Putting automatic synthesis of node-link-diagrams into practice." In: *IEEE Symposium on Visual Languages and Human Centric Computing*. 2013. DOI: 10.1109/VLHCC.2013.6645246.

[99] R Henia; A Hamann; M Jersak; R Racu; K Richter; and R Ernst. "System level performance analysis – the SymTA/S approach." In: *IEE Proceedings - Computers and Digital Techniques* 152.2 (2005). ISSN: 13502387. DOI: 10.1049/ip-cdt:20045088.

[100] K Marriott; PJ Stuckey; and PJ Stuckey. *Programming with constraints: an introduction*. MIT press, 1998.

[101] J Iber; A Höller; T Rauter; and C Kreiner. "Patterns for Designing Configurability into Domain-Specific Language Elements." In: *Proceedings of the 21st European Conference on Pattern Languages of Programs*. EuroPlop '16. ACM, 2016. ISBN: 978-1-4503-4074-8. DOI: 10.1145/3011784.3011785.

[102] J Iber; T Rauter; M Krisper; and C Kreiner. "The Potential of Self-Adaptive Software Systems in Industrial Control Systems." In: *Proceedings of the 24th European Conference on Software Process Improvement*. EuroAsiaSPI '17. Springer International Publishing, 2017. ISBN: 978-3-319-64218-5.

[103] A Höller; T Rauter; J Iber; and C Kreiner. "Patterns for Automated Software Diversity to Support Security and Reliability." In: *EuroPLoP '15*. ACM, 2015. ISBN: 978-1-4503-3847-9. DOI: 10.1145/2855321.2855360.

[104] H Hadeli; R Schierholz; M Braendle; and C Tuduce. "Leveraging determinism in industrial control systems for advanced anomaly detection and reliable security configuration." In: *2009 IEEE Conference on Emerging Technologies & Factory Automation*. IEEE, 2009. ISBN: 978-1-4244-2727-7. DOI: 10.1109/ETFA.2009.5347134.

[105] SD Anton; S Kanoor; D Fraunholz; and HD Schotten. "Evaluation of Machine Learning-based Anomaly Detection Algorithms on an Industrial Modbus/TCP Data Set." In: *Proceedings of the 13th International Conference on Availability, Reliability and Security - ARES 2018*. ACM Press, 2018. ISBN: 9781450364485. DOI: 10.1145/3230833.3232818.

[106] M Kylänpää and A Rantala. "Remote Attestation for Embedded Systems." In: *Security of Industrial Control Systems and Cyber Physical Systems*. Ed. by A Bécue; N Cuppens-Boulahia; F Cuppens; S Katsikas; and C Lambrinoudakis. Springer International Publishing, 2016, pp. 79–92. ISBN: 978-3-319-40385-4.

[107] B Galloway and GP Hancke. "Introduction to Industrial Control Networks." In: *IEEE Communications Surveys & Tutorials* 15.2 (2013). ISSN: 1553-877X. DOI: 10.1109/SURV.2012.071812.00124.

[108] U Aßmann; S Götz; J-M Jézéquel; B Morin; and M Trapp. "A Reference Architecture and Roadmap for Models@run.time Systems." In: *Models@run.time: Foundations, Applications, and Roadmaps*. 2014. DOI: 10.1007/978-3-319-08915-7_1.

[109] F Hayes-Roth. "Rule-based systems." In: *Communications of the ACM* 28.9 (1985). ISSN: 00010782. DOI: 10.1145/4284.4286.

[110] I Fette and A Melnikov. *The WebSocket Protocol*. RFC 6455. RFC Editor, 2011.

[111] A Höller; G Schönfelder; N Kajtazovic; T Rauter; and C Kreiner. "FIES: A Fault Injection Framework for the Evaluation of Self-Tests for COTS-Based Safety-Critical Systems." In: *2014 15th International Microprocessor Test and Verification Workshop*. IEEE, 2014. ISBN: 978-1-4673-6858-2. DOI: 10.1109/MTV.2014.27.

[112] J Iber; M Krisper; J Dobaj; and C Kreiner. "Dynamic Adaption to Permanent Memory Faults in Industrial Control Systems." In: *Proceedings of the 9th International Conference on Ambient Systems, Networks and Technologies*. ANT '18. Elsevier, 2018. DOI: 10.1016/j.procs.2018.04.058.

[113] J Iber; T Rauter; M Krisper; and C Kreiner. "Patterns Grasping the Trade-off Between Distributing Data and Information." In: *Proceedings of the 22nd European Conference on Pattern Languages of Programs*. EuroPLoP '17. ACM, 2017. ISBN: 978-1-4503-4848-5. DOI: 10.1145/3147704.3147724.

[114] J Iber; M Krisper; J Dobaj; and C Kreiner. "Separation of processing and coordination in computer systems." In: *Proceedings of the 23rd European Conference on Pattern Languages of Programs*. EuroPLoP '18. ACM, 2018. ISBN: 978-1-4503-6387-7/18/07. DOI: 10.1145/3282308.3282322.

[115] B Selić and S Gérard. *Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE*. 2014. ISBN: 9780124166196. DOI: 10.1016/B978-0-12-416619-6.00008-0.