TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Software and Multimedia Technology, Software Technology Group

Master Thesis

A CONTEXT-AWARE ROLE-PLAYING AUTOMATON FOR SELF-ADAPTIVE SYSTEMS

Lars Schütze

Born on: 8th February 1989 in Dresden
Matriculation number: 3569963
Matriculation year: 2009

to achieve the academic degree

Master of Science (M.Sc.)

Supervisor
Dr.-Ing. Sebastian Götz

Supervising professor
Prof. Dr. rer. nat. habil. Uwe Aßmann

Submitted on: 17th June 2016

TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Software- und Multimediatechnik, Professur Softwaretechnologie

Masterarbeit

EIN ROLLENSPIELAUTOMAT FÜR SELBST-ADAPTIVE SYSTEME

Rollenbasierte Modellierung und Programmierung wird in Zukunft eine entscheidende Rolle bei der Realisierung komplexer, adaptiver Softwaresysteme spielen. Dabei wird das objektorientierte Paradigma um das Konzept der Rolle erweitert. Objekte können zur Laufzeit Rollen spielen und wieder ablegen. Das Spielen einer Rolle verändert dabei den Typ des Objektes, fügt gegebenenfalls Attribute und Methoden hinzu, beziehungsweise überschreibt vorhandene Methoden. Darüber hinaus beschreiben Rollen dynamische Beziehungen eines Objektes zu anderen Objekten. Welche Rollen ein Objekt spielen kann, wird über seinen Typ (die Klasse beschrieben). Auf Klassenebene werden Klassen mit Rollentypen in Beziehung gesetzt und die Menge der zugeordneten Rollentypen bildet die Menge an Rollen, die ein Objekt dieser Klasse spielen kann. Aktuell existiert jedoch kein einheitlicher Ansatz, der beschreibt unter welchen Vorrausetzungen ein Objekt eine Rolle spielt bzw. wieder ablegt.

Thomas Kühn hat in seiner Belegarbeit „Explizite Rollenbindung mit Story Boards“ ein Verhaltensmodell auf Basis von Storyboards entworfen, das die Bindung (der Beginn des Spielens einer Rolle) und das Entfernen von Rollen beschreibbar macht. Diese Belegarbeit wurde rein auf konzeptueller Ebene erstellt und weder in einen Softwareentwurf überführt noch umgesetzt. Darüber hinaus fehlt es in diesem Konzept an externen Kontextbedingungen, die für selbst-adaptive und kontextsensitive Systeme entscheidend sind.

In der Masterarbeit soll der Rollenspielautomat von Thomas Kühn um Kontextinformationen und Compartments erweitert werden. Auf Basis dieses Konzepts soll ein Softwareentwurf erstellt werden, der auf dem plattformunabhängigen Rolltypmodell CROM basiert. Darüber hinaus soll der Rollenspielautomat unabhängig von einem konkreten Rollenlaufzeitsystem entwickelt werden. Über fest definierte Schnittstellen können dann plattformspezifische Adapter implementiert und so beliebige Laufzeitsysteme integriert werden. Des Weiteren soll der Entwurf in der Programmiersprache Java umgesetzt und anhand eines selbst gewählten Beispiels evaluiert werden.

Die folgenden Unteraufgaben müssen dabei erledigt werden:

- Literaturanalyse zu den Themen: Rollen, Rollenbindung, Selbst-Adaptive Systeme
- Einarbeitung in das Konzept zu Storyboard-basierten Rollenspielautomaten
- Erweiterung des Konzepts um Kontextinformationen und Compartments
- Entwurf einer Realisierung des erweiterten Konzepts
- Implementierung des Entwurfs
- Evaluierung der Implementierung anhand eines selbst-gewählten Beispiels

Betreuung
Christian Piechnick
christian.piechnick@tu-dresden.de
0351 / 463 38377

Statement of authorship

I hereby certify that I have authored this Master Thesis entitled *A Context-Aware Role-Playing Automaton for Self-Adaptive Systems* independently and without undue assistance from third parties. No other than the resources and references indicated in this thesis have been used. I have marked both literal and accordingly adopted quotations as such. They were no additional persons involved in the spiritual preparation of the present thesis. I am aware that violations of this declaration may lead to subsequent withdrawal of the degree.

Dresden, 17th June 2016

Lars Schütze

CONTENTS

1. INTRODUCTION

Role-based modeling and programming will become more and more important to realize big, complex, and adaptive software systems [Zhu and Alkins, 2006]. Therefore, the Object-Oriented Programming (OOP) paradigm is extended with roles, where objects can begin to play roles and drop roles dynamically at runtime. Playing a role is changing the object's type which can add or change behavior. Roles are a dynamic view of the state and behavior of objects at runtime at a point of time highlighting their relations to other objects. This has been depicted in figure 1.1. Hence, the type of an object at runtime is determined by its own type and the set of roles it plays. The roles an object can play is prescribed by its class. The class model is merged with the role model where classes become overlapped by the role types they can play. The resulting subset of role types that the class is related with is the set of roles an object can play.

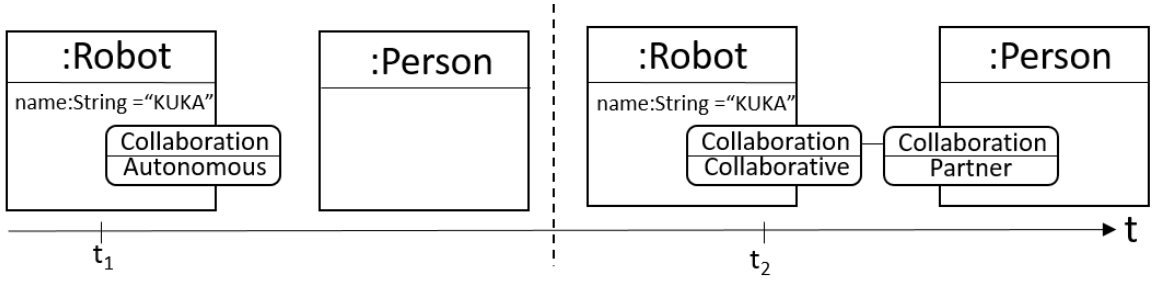

Figure 1.1.: Snapshot of two objects at two different times t_1 and t_2. At first the robot is playing the autonomous role and is not related to the person in any way. Later it has the collaborative role and is related to the partner role played by person.

Self-adaptive systems (SAS) are naturally context-aware systems. Thus, adaption is always seen in a context e.g., because a sensor value passes a specified limit, or because the reason could be derived from the knowledge about the past and presence. However, there is currently no common concept describing the situation (e.g., the context or other conditions that lead to a specific adaption) in which objects begin to play and stop playing roles. Current role programming languages therefore suffer from the problem of tangling of different aspects i.e., the context logic, the role adaption logic, and the business logic. This leads to less understandable and unmaintainable code [Antinyan et al., 2014]. Thomas Kühn has drafted in his major thesis [Kühn, 2011] a behavioral model to describe role binding with storyboards. This allows to model concisely role reconfigurations, but the concept lacks the ability to specify context-dependent behavior which is crucial for self-adaptive systems, and is built on top of an outdated understanding of the role concept which lacks compartments.

The concept of storyboards will be extended with the ability to address context-dependent conditions. Compartments will be added in order to adapt the current wider understanding of the concept of roles. This will result in a concept for *context-aware storyboards with roles* which provide a separation of concerns approach w.r.t. the above named concerns. The concept will be implemented as automaton and will be evaluated on a use case. The use case is a robotic co-working scenario based on the idea of [Haddadin et al., 2009]. Given the idea of a robot working autonomously, but able to adapt to work collaboratively when humans are entering the working area of the robot. This idea will be used as a running example throughout the thesis and the result of the thesis will be evaluated on a scenario representing this use case. The scenario is depicted in figure 1.2. Thus, the robot will register when a human is entering the working area, and will accordingly adapt its behavior to a collaborative mode. When the human is leaving the area the robot will return to autonomous mode. This is achieved using sensor data which are analysed by the self-adaptive system. This in turn triggers adaptions w.r.t. the role reconfigurations emitted by the context-aware storyboards with roles.

(a) An autonomous working robot. (b) Person near robot will adapt it to collaborate.

Figure 1.2.: The KUKA robotic arm is adaptable w.r.t. the roles it is playing. If alone it is working autonomously. If a human enters the working area the robot is adapted to collaborate.

1.1. MOTIVATION

Currently, there is no common approach for defining context and role adaption. There exist different role programming languages which allow to state role adaption directly in the code. This role adaption logic is dependent on role context (i.e., compartments) and role relations. For big role models the programming of these systems becomes cumbersome (e.g., the many relations that take part in a decision if a role will be bound). Thomas Kühn has shown that storyboards offer a good way to describe such scenarios under which role adaption will be executed [Kühn, 2011].

Self-adaptive systems, which are by nature context-dependent, adapt the system because of reasons either derived from past and present knowledge, or because of incidents (i.e., sensor value passes a specified limit). Therefore adaptions are happening in a context (e.g., place, time, state), hence they are context-dependent. Playing a role changes the type of the object, and adds or changes behavior. Thus, role binding can be seen as a form of adaption. Combining both approaches, this leads to a mix of the context of the role binding itself, and the constraints on the context under which specific adaptions should be executed. For example, querying the sensor if a person is near is part of the context logic, and the resulting

role adaption of the robot is part of the adaption logic. Thus a tangling of the concerns of context logic, adaption logic, and business logic results. This problem exists in every role programming language (e.g., SCROLL, ObjectTeams/Java, RSQL, EpsilonJ).

The main goal of the thesis is therefore to present an approach for separation of concerns for the context logic, adaption logic, and business logic. Second, the approach should be suitable regardless of the role programming language used. The thesis will therefore investigate how self-adaptive systems and role adaption can be combined with the use of storyboards.

1.2. OUTLINE

The remainder of this thesis is built as follows. In chapter 2 ("Background and Concepts") the reader will be introduced to background concepts that are used as a framework for the following concept. Beside terminology this comprises applied technology. This will outline current technologies used to tackle the problem the thesis will solve. Furthermore, since the thesis will provide an executable model executable modeling technologies are introduced. The most important foundation of the developed concept are storyboards which are introduced in detail.

In chapter 3 ("Requirements Analysis") the problem when combining the different systems and concepts is analyzed. Then goals are derived which will then lead to the requirements the concept must fulfill. Because the framework of the developed concept i.e., storyboards explained in chapter 2, consist of different aspects a technology analysis is conducted where different implementations for these aspects are compared.

Chapter 4 ("Concept for a Role-Playing Automaton for Self-Adaptive Systems") presents the concept of this thesis. This includes the presentation of all syntactical elements as well as their semantics. The derived meta-model is shown to explain how the presented goals have been mapped into the concept. The concept is then compared to solutions it has been initially derived from. This concept is then applied in chapter 5 ("Implementation") where the prototypical implementation CAESAR is introduced. This includes a self-made selection of aspects that are relevant to understand the implementation.

In chapter 6 ("Related Work") other related works are presented, evaluated and compared to CAESAR.

Chapter 7 ("Evaluation") applies CAESAR in a use case scenario. The results of the thesis are evaluated against the requirements defined in chapter 3. CAESAR is evaluated and the contribution is shown, as well as shortcomings outlined.

At last a summary is drawn in chapter 8 ("Conclusion and Future Work") where the content of the thesis is summarized, and future work is outlined.

2. BACKGROUND AND CONCEPTS

This chapter will explain preliminary concepts, techniques, and architectures which are required to be understood, and will be adapted or extended by the thesis. This includes concepts from the modeling domain, its application in behavioral diagrams, and special software architectures. At first it is explained what roles are and how they are used to model modern software. Different approaches to implement the role concept are shown. Role-based programming is introduced and current role-based programming languages are presented. Then modeling concepts which are considered related to a role-playing automaton are introduced. For behavioral modeling concepts UML activity diagrams are introduced, because story diagrams use their semantics. Petri nets are shown as an alternative to UML activity diagrams. Furthermore, storyboards and their extension with roles are introduced. Third, relevant software architectures are presented and last a summary is drawn.

2.1. ROLE-BASED DESIGN

Reenskaug wrote an article on a blog about a concept which exhibits roles called Data, Context, Integration (DCI) [Trygve Reenskaug and James O. Coplien, 2009]. Reenskaug has seen that the object-oriented paradigm does capture structures of the real world very well, but fails at describing their relations and interactions. But given the mental model that represents the real world people mind for the interaction part to be mirrored into the code. He says that "*objects capture what objects are, [and] roles capture what objects do*" [Trygve Reenskaug and James O. Coplien, 2009]. Thus, OOP is good at capturing structure well, but fail at capturing the dynamics of the system (e.g., emerging collaborations between objects in their contexts) [Riehle et al., 1998; Trygve Reenskaug and James O. Coplien, 2009]. Roles have been proposed by Bachman as a means to model complex and dynamic domains, because roles are able to capture context-dependent and collaborative behavior of objects [Charles W. Bachman, 1973]. Since then there have been almost 40 years of research which diverged the research field. Research mainly concentrated on single nature of roles e.g., its relational or context-dependent nature. At the big picture there is no common understanding of roles. However, [Kühn, Böhme, et al., 2015] tried to unite these different approaches to emphasize the relational and context-dependent part of roles resulting in a meta-model family.

2.1.1. ROLES AND ROLE MODELS

In Object Management Group (OMG) Unified Modeling Language (UML) roles are the names of association ends and in OMG Object Constraint Language (OCL) those role names are

used to navigate on the object graph [Object Management Group, 2014; Object Management Group (OMG), 2012]. However, these are just names of the role the other class is taking in the collaboration. There is no further constraining or reasoning on what methods take part in the collaboration and under which circumstances. Classes specify what instances will do with incoming messages, but says nothing about the network of communicating objects at a whole [Reenskaug, 2011]. In role-based design the dynamic view onto an object at runtime is emphasized. Roles can be played and removed by an object at runtime. They both share the same identity. Thus, roles have no own identity and need to be played by objects. Roles may also be played by other roles (deep roles). Roles are defined by a role type in a role model such as an instance is defined by its class in the class model. Roles have their own properties and methods which allows for interface-structuring, separation of concerns and dynamic collaboration description. The latter is achieved by defining relations of role types in a role model. Playing a role alters the interface of the object and adds or removes behavior and properties. Hence, the interface of an object (e.g., the roles it is playing) changes dynamically dependent on the current (role-) context and state of the object.

A role model describes role types, their properties and behavior, and their relationships. This describes object collaborations because roles have to be played by objects. In a class model classes are related without any means on which parts of the interface are involved in the relationship. A role model offers separation of concern in a way that only a single concern (i.e., a role type) is related with another. Thus, it clearly defines which part of the interface is part of the relationship. Role models allow to address single domain specific concerns and can be subject to role model composition. Thus, bigger role models out of smaller can be composed which allows for reuse and separation of concern. The role model is merged with a class model to form the role-type-class-model. Thus, all role types are bound to classes (e.g., role types overlap the classes). Role restrictions allow to define under which circumstances a role type can be bound to a class [Kühn, Böhme, et al., 2015; Riehle et al., 1998].

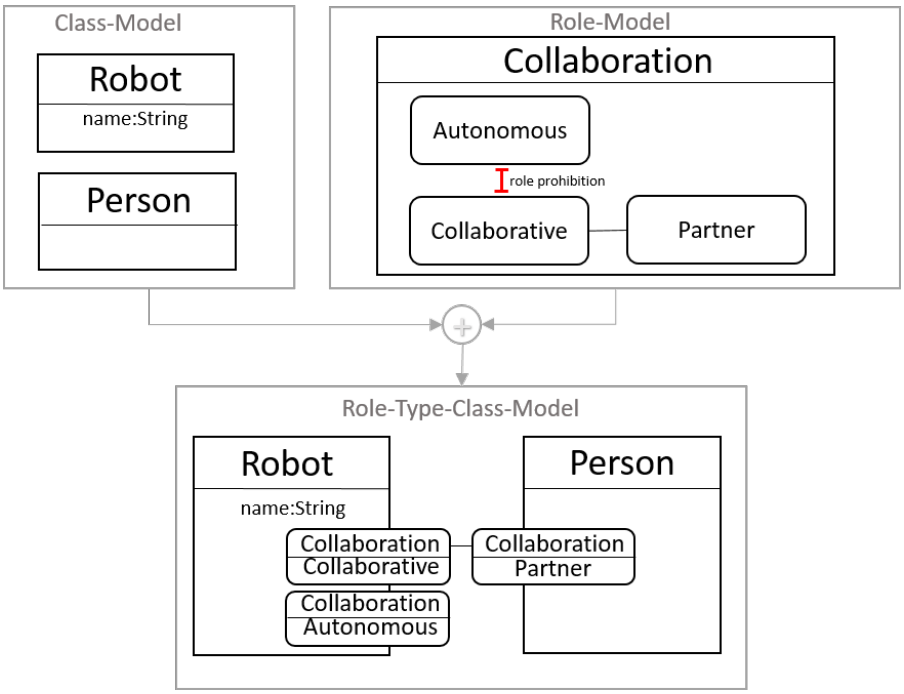

Figure 2.1.: A class model and the role model are merged to the role-type-class-model. Every class is assigned the roles it is allowed to play.

There are different types of roles depending on the role they are playing. A *rigid type* is tied to the object identity as the object cannot stop playing the role without loosing its identity, and a *non-rigid* type is dynamic as the object can start and stop playing the role. For example, a rigid type is be a book and a non-rigid type is a reader. Types can be founded if they only exist in a collaboration (e.g., a reader needs something to read). Thus, [Steimann, 2000] classifies role types as founded, non-rigid and natural types as non-founded and rigid.

Collaborations between roles can be constrained in their scope when seeing their relationships in a context. The term *context* has different meanings dependent on the area it is applied. On the one hand Anind Dey defines context as "*any information that can be used to characterize the situation of an entity*" [Dey, 2001], and on the other hand in the modeling language domain context is understood as a container or collaboration [Genovese, 2007; Trygve Reenskaug and James O. Coplien, 2009]. Because of this different meanings researchers avoid the term context and introduced other terms to denote the context of a role e.g., *team* [Herrmann, 2005], *institution* [Baldoni et al., 2006], *environment* [Zhu and Zhou, 2006], and *compartment* [Kühn, Leuthäuser, et al., 2014]. Throughout this work the term *compartment* will be used to denote the context of a role. Relations can just happen in an compartment and cannot be across two compartments. Thus, an object joins a compartment by assuming one of its roles. The role model defines which roles belong to a compartment.

Role-based modeling can be implemented using classical object-oriented paradigms such as multiple inheritance, interfaces, or mixin inheritance. Because there is no silver-bullet approach on implementing role-based modeling there is a deep gap between design and implementation.

Using Aspect-Oriented Programming (AOP) [Kendall, 1999] exhibit static roles which are mixed into classes during compile time. Mixing layers is an approach proposed by [Smaragdakis et al., 2002] which is not limited to C++ templates, but best implemented with them. Every layer defines its own roles which either are inherited or refined in the next layer. Thus, every instance per layer describes a role. The problem is that all these approaches are compile time implementations and adding or removing roles is not possible at runtime.

Another possible solution that allows for adding and removing roles at runtime is the Role Object Pattern (ROP) [Bäumer et al., 1998]. It separates objects into a core class and its role classes. Communication is done via delegation and the core has to manage all classes an object plays. Role models can be directly expressed with ROP and implemented with object-oriented programming languages. Beside ROP fits good for role models, but leads to *object schizophrenia* at implementation level [Kendall, 1999]. Object schizophrenia is a phenomenon that appears when the defined *self* semantics of an object is ambiguous. This happens in object-oriented programming by inheritance. The base class is used as template while a derived class can overwrite behavior. Calling *self* on the object becomes ambiguous as it could either relate to the base class or overwritten behavior of the derived class. Object schizophrenia also arises when *delegation* is used which means that a method can be executed in behalf of another object. Then, *self* is not statically bound to the receiving object, but the original object. In Java delegation is not supported as a language feature, but can be emulated by passing the original object as parameter and replace the call to `this` with the originals object reference. Object schizophrenia could also appear when a set of objects form a single *logical entity* (e.g., the identity of the logical object is split into multiple objects). In role modeling real-world entities are represented by multiple objects. Thus, there is a *broken identity*. The role objects are *entity-equivalent* when they belong to the same real world entity. But object schizophrenia is not a problem itself [Herrmann, 2010]. The object schizophrenia problem arises when the role model is not entity-equivalent [Sekharaiah et al., 2002].

[Kühn, Böhme, et al., 2015] developed a meta-model family for roles called Compartment

Role Object Model (CROM) and Compartment Role Object Instances (CROI) which is to my knowledge the only available complete meta-model implementation for roles today. They will be further explained in section 3.3.

2.1.2. ROLE BINDING

Role binding is a process where roles are bound to an object. The process of binding roles consists of the binding technique, and the binding operation.

The binding technique is how the relations are implemented. Therefore, different role-programming implementations use different design patterns. These design patterns are themselves not based on roles, but object-orientation which exhibits a problem with references not being bidirectional. Thus, the roles and objects do not just have to save its context, but the context has also to save all collaborations that take place in itself.

The binding operations are put together to form the role binding process. The binding process binds the object with the role together in the context.

2.1.3. ROLE RUNTIME SYSTEMS

In the previous sections the concepts of roles and the techniques involved to implement them has been layed out. This section will introduce some common role-programming languages like ObjectTeams/Java (OT/J) [Herrmann, 2007], EpsilonJ [Tamai et al., 2005], and Smart Application Grids (SMAGs) [Piechnick, Richly, et al., 2012]. These implementations differ in their role-binding mechanisms. OT/J offers implicit role-binding, whereas EpsilonJ offers explicit role-binding. To name more existing implementations that will not be presented in this section are powerJava [Baldoni et al., 2006], an execution model for DCI [Reenskaug, 2011], and view-based programming using roles and dynamic dispatch with SCala ROles Language (SCROLL) [Leuthäuser, 2015]. Beside different names for the same concept (e.g., team, environment, context, compartment) they all offer implementations for the overall concept of roles on the language level but differ in their features.

OBJECTTEAMS/JAVA

Stephan Herrmann developed a programming language for role-based programming [Herrmann, 2005, 2007] that could be seen as an ontology which defines its own concepts of role, *team* and their relationships. The goal was to offer the possibility to correspond a program written in OT/J with the domain model (or mental model of a domain expert). A *team* is the notion of a context for roles in ObjectTeams. A team encapsulates roles which are defined in teams. Teams allow to define methods and visibility levels (e.g., private, public) on methods and roles. Role binding is defined on the class level where role classes and classes are related. Therefore, ObjectTeams provides the `playedBy` relation which can be annotated to a role class definition (see listing 2.1 lines 9-11). When a base class instance enters a team's method (see listing 2.1 line 13) it will be automatically lifted to the given role. Internally, the team constructs a map from base to role instances. Thus, role types are bound statically to their base class. The novel feature of ObjectTeams is the introduction of implicit lifting and lowering. When a base instance enters a team it is implicitly lifted to the appropriate role type or when data is passed outside of a team it is implicitly lowered to its base type at the data stream sink which they call *translation polymorphism* [Herrmann, Hundt, and Mehner, 2004]. Roles are called bound when the role is attached with a `playedBy` relation. Otherwise it is an unbound role and will never be attached to a base class automatically. However, a role instance cannot change its base instance at any time during its lifetime, but it can be unattached from its base class which will result in its destruction. This means, that ObjectTeams does not support role migration

on the language level. Beside implicit lifting and lowering ObjectTeams supports explicit role binding where it is directly stated which object will be playing the role. [Herrmann, Hundt, and Mosconi, 2008, § 2.4.] defines *explicit role creation* as a mechanism to explicit state which object plays which role. Listing 2.1 on lines 17 and 20 shows an example on how to create roles explicitly for a given object. OT/J allows for role inheritance.

```
1  public class Robot {}
2  public class Person {}
3  public team class Collaboration {
4    private Robot r;
5    private Autonomous a;
6    private Collaborative c;

8    public class CoWorker playedBy Robot {}
9    public class Autonomous playedBy Robot {}
10   public class Collaborative extends CoWorker {}
11   public class Worker playedBy Person {}

13   public void robotIsReady(Robot r) {
14     this.r = r;
15     robotIsAutonomous(r);
16   }
17   public void robotIsCollaborative(Collaborative c) {
18     this.c = c;
19   }
20   public void robotIsAutonomous(Autonomous a) { this.a = a; }
21   public void personEntersArea(Worker w) {
22     unregisterRole(a);
23     robotIsCollaborative(r)
24   }
25   public void personLeavsArea(Worker w) {
26     unregisterRole(w)
27     unregisterRole(c);
28     robotIsAutonomous(r);
29   }
30 }
31 Collaboration c = new Collaboration();
32 Robot r = new Robot();
33 Person p = new Person();
34 c.robotIsReady(r);
35 //Person enters working area
36 c.personEntersArea(p);
37 //Person leaves working area
38 c.personLeavsArea(p);
```

Listing 2.1: An implementation of the use case role model depicted in figure 1.2. It uses implicit lifting and explicit role creation.

EPSILON AND EPSILONJ

Tamai present a role-based model Epsilon which features *environments* and roles as first class constructs in the runtime as well as model description time [Tamai et al., 2005].

The model is implemented as an extension for the Java programming language called EpsilonJ by emphasizing the Java 5 features of annotations[1]. An *environment* is regarded as a collaboration field between objects and roles. Objects can freely join and leave environments which is done by assuming a role in the environment and discarding a role respectively. Epsilon supports description of collaborations not only at the model level but also at programming level. The programming language defines environments as `context`. The context encapsulates roles in environments. Thus, roles are always defined in a context but can be referenced from outside using the qualified name. The qualifier `static` allows to define singleton [Gamma et al., 1995] roles per context instance. The language allows to drop roles by calling `unbind()` on the role instance. The playing object will discard the role, but the role object will be preserved allowing for role migration. EpsilonJ allows to *import* a method on an object that is bound to a role using the `require` statement. Then the role does not have to implement the method and the object that is bound into the role is providing the implementation. In the other direction an export can be stated with `replacing roleMethod() with playerMethod()`. EpsilonJ does not allow for role inheritance. Thus, the `CoWorker` role is not implemented in the code.

```
1  class Person {}
2  class Robot {}
3  context Collaboration {
4    static role Collaborative{}
5    static role Autonomous {}
6    role Worker {}
7  }
8  Collaboration c = new Collaboration();
9  Robot r = new Robot();
10 c.Autonomous.bind(r);
11 //Person p enters working area of Robot r
12 Worker w = c.Worker.newBind(p);
13 c.Autonomous.unbind();
14 c.Collaborative.bind(r);
15 //Person p is leaving the working area
16 w.unbind();
17 c.Collaborative.unbind();
18 c.Autonomous.bind(r);
```

Listing 2.2: Simple implementation of a robotic co-worker role model with runtime as defined in the role model figure 1.2. Normally, there have to be checks if the robot is already playing the collaborative role, which are omitted for simplicity reasons.

SMART APPLICATION GRIDS

The main goal of Smart Application Grids (SMAGs) is to allow unanticipated adaptions at runtime, because after Lehman's law one cannot foresee every scenario at design time. This is achieved by using a Models@Run.time approach and SMAGs being a *composition system*. According to Aßmann, a composition system consists of a *component model*, a *composition technique*, and a *composition language*.

The component model consists of *ComponentType*s which are stateful, self-contained software modules. They define a functional interface described by grouping several *Port-*

[1] Annotations are meta-data which can be processed at compile and runtime. See JCP 175 for more details: https://jcp.org/en/jsr/detail?id=175

*Type*s. A PortType represents a language-independent interface description. PortTypes are offered or required by a ComponentType. It has a unique name, and provides a functional view by naming all interface methods, and a state view by stating a list of external modifiable parameters. A PortType can be implemented by at least one to several *Port*s. A PortType can require other PortTypes in order to be able to be used. There are two PortTypes, namely *BehavioralPortType*s which allow access to application functionality, and *EventPortType*s which provide events other components can register on. A Port is a platform-specific implementation of a PortType. It specifies meta-data to state its suitability for a specific domain. ComponentTypes use an IDL compiler to be transferred to a language specific interface description. The components that implement the ComponentTypes are language specific. They implement standard implementations for all functions offered by their provided PortTypes. To allocate and release all required resources components offer an install and uninstall script which is executed. In other words, the SMAGs component model is equivalent to role modeling. Figure 2.2 shows the connections. The PortType and Port represent the role model while the ComponentType with all required and offered PortTypes represents a role-type-class-model.

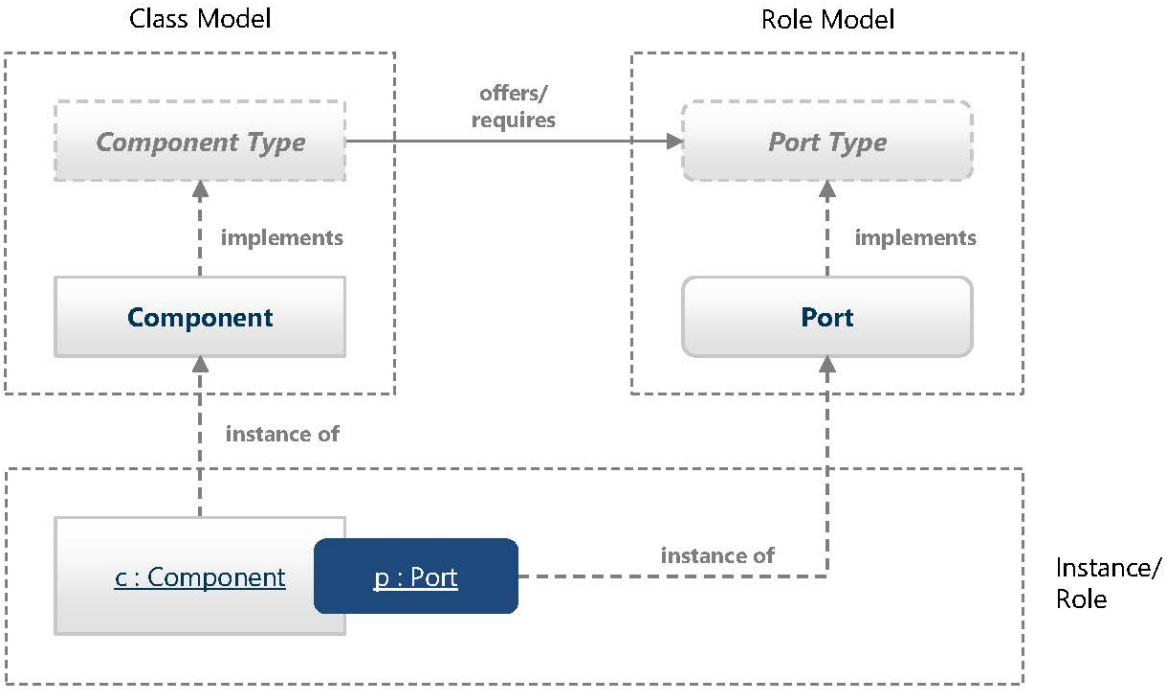

Figure 2.2.: A meta-architecture of SMAGs applications. A component type is implemented by a component. The port type implemented by a port. At runtime, a port instance is combined with a component instance to build a functional unit. Taken from [Piechnick, 2016]

The composition technique consists of passive connectors which connect required PortTypes of a component at runtime with offered PortTypes of other components. The components can be extended at design-time using inheritance or by an *extend operator* at runtime. Furthermore, it offers a *bind*, an *adapt*, and a *filter composition operator*. All these composition operators are implemented using the *composition filter* approach [Bergmans et al., 1992].

The composition language allows to describe the architecture in two different layers. The meta-architecture (which consists of ComponentTypes and PortTypes) can be compared to the creation of a role model, and the generation of a role-type-class-model. The architectural implementation consists of the implementation of platform-specific components for each

ComponentType.

Figure 2.3 shows the *SMAGs Repositories* which allows to reuse and deploy SMAG model elements at runtime. It is a static repository for ComponentTypes, PortTypes, meta-architectures, architectures, Components, and Ports. SMAG applications can publish, search, and retrieve elements. An admin interface allows to publish and unpublish or modify elements. It uses URI to reference elements, and Web Ontology Language (OWL) concepts to enable computable meta-data.

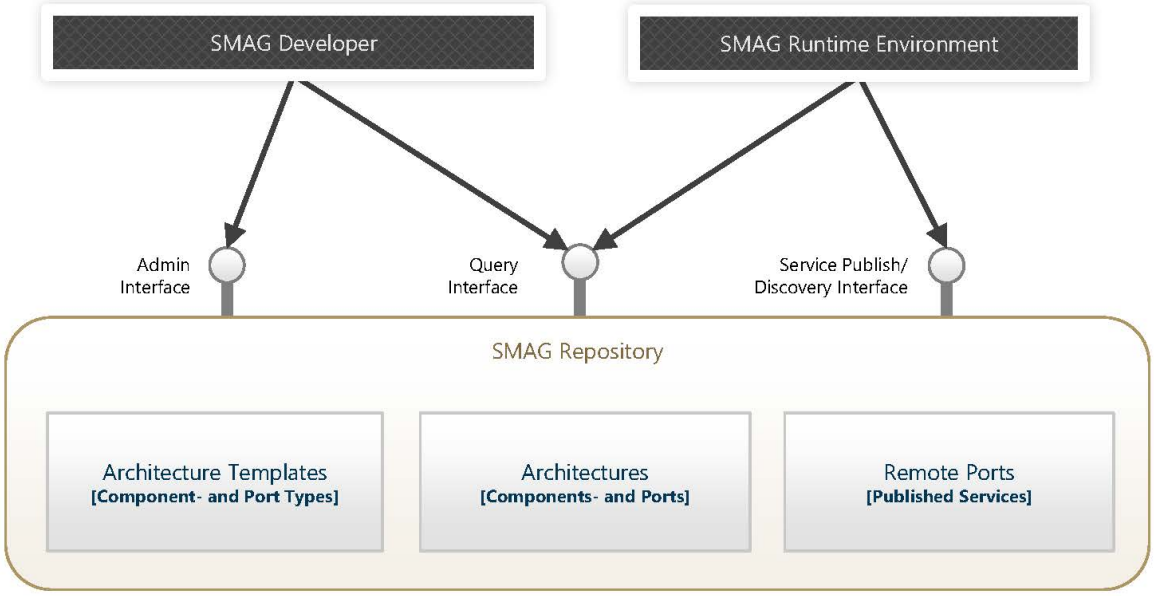

Figure 2.3.: The repository based architecture of SMAGs. Taken from [Piechnick, 2016]

The overall SMAGs architecture is shown in figure 2.4. It allows the adaption of the application at runtime using a variant of the MAPE-K loop. Playing a role is changing or adapting the behavior of the object at runtime. Therefor, SMAGs uses roles for adaption [Piechnick, Richly, et al., 2012].

2.2. MODELING CONCEPTS FOR A ROLE-PLAYING AUTOMATON

In [Davis et al., 1999] role-playing is defined as a bridging mechanism to close the gap between architecture analysis and application task analysis. In [Chrszon et al., 2016] role-playing is understood as a role which is bound to an object that therefore will begin to play the role. Thus, role binding is the requirement for role-playing. In this thesis the understanding of role-playing of Chrszon et al. is shared. Furthermore, for the implementation of binding roles and playing roles external systems should be used. Since role-binding and role-playing is externalized the automaton is just modeling the reasons which lead to play a specific role. Thus, a role-playing automaton is a Finite State Machine (FSM) that models the circumstances that lead to playing specific roles. Edges define reasons and requirements to further adapt the players and objects while the nodes define what roles are bound and unbound. This conforms to the view mentioned by Chrszon et al. because in order to play a role it first has to be bound.

There exist some fundamental concepts in the modeling domain that allow to define FSM, which can be used to implement the said behavior, and thus are used and exploited in the thesis. This section will introduce these concepts and models. At first a glance about models and meta-models is given, then an overview about selected behavioral diagrams is

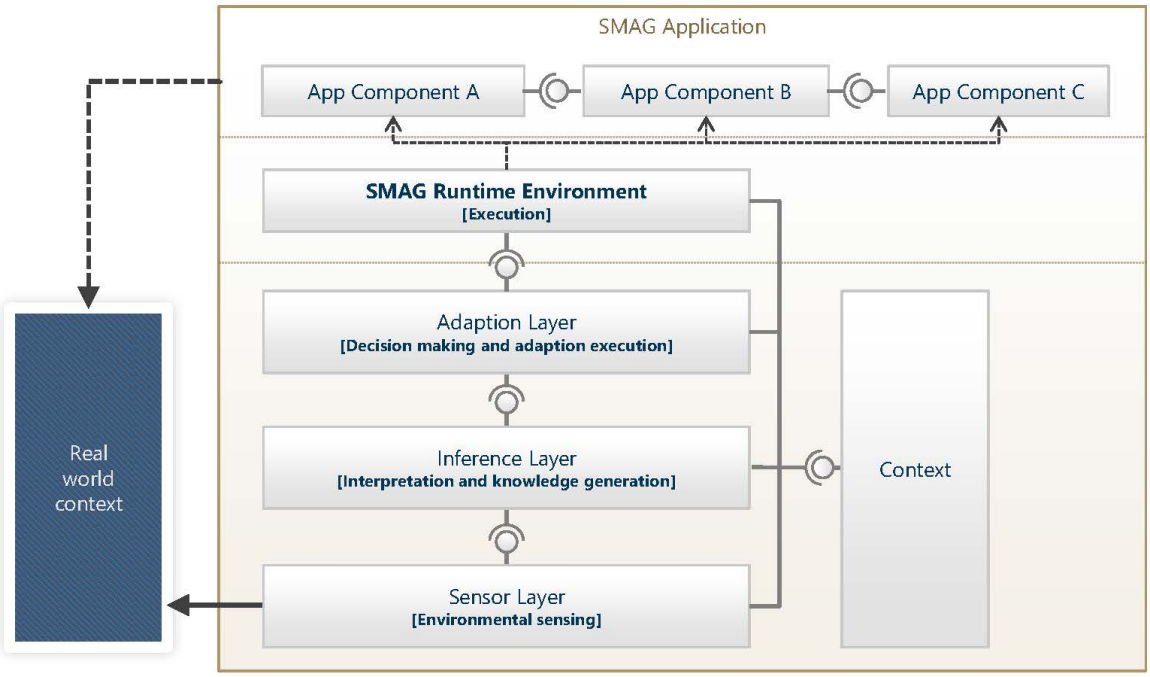

Figure 2.4.: The architecture of SMAGs. Implementing a variant of the MAPE-K loop. Taken from [Piechnick, 2016]

presented and last relevant software architectures are described.

2.2.1. MODELS AND META-MODELS

The notion of a *model* is very broad and its understood differently in different domains. Even in the domain of software engineering there is no common ontology which means that there are different understandings in the notion of a model. In software engineering a model is an artifact formulated in a modeling language, such as UML, normally describing (part of) a system with different diagrams [Kühne, 2006]. In general, models are used to abstract the reality (e.g., a language-based program or the real world) for better understanding. Formalization allows for code generation and computational processing. However, in the context of model driven engineering this view is too narrow as "everything is a model" which applies the model term to e.g., Java programs, too.

Meta- is a prefix designating another subject which describes or analyzes the original subject at a more abstract, higher level way [Oxford English Dictionary, 2016]. If you have a discussion about how to discuss it is called meta-discussion. In the software engineering domain a meta-model is a model which describes the concept of a model, hence it is a model of a model. This introduces a hierarchy where the lower tiers model concept is described by a more abstract model (the meta-model) resulting in a meta-model hierarchy. Thus, all elements of a level are instances of the concepts defined on the level above. Currently, these meta-model hierarchies are primarily emerging as the four layer approach introduced by [CDIF Technical Committee, 1994]. One of its adoptions is the meta-model hierarchy of OMG's Metaobject Facility (MOF) [Object Management Group (OMG), 2011].

There exist four levels in the hierarchy of MOF which begins at the bottom with M0, the *instance level*. There exist three meta levels M1, M2, and M3. M1 is the *model*, M2 describes the concept of models (*meta-model*) and M3 describes the structure of which concepts of concepts of models (*meta-meta-model*) are consisting of. Further levels are not needed anymore as M3 is able to explain its own concepts with itself. Figure 2.5 explains

the hierarchy on a simple example.

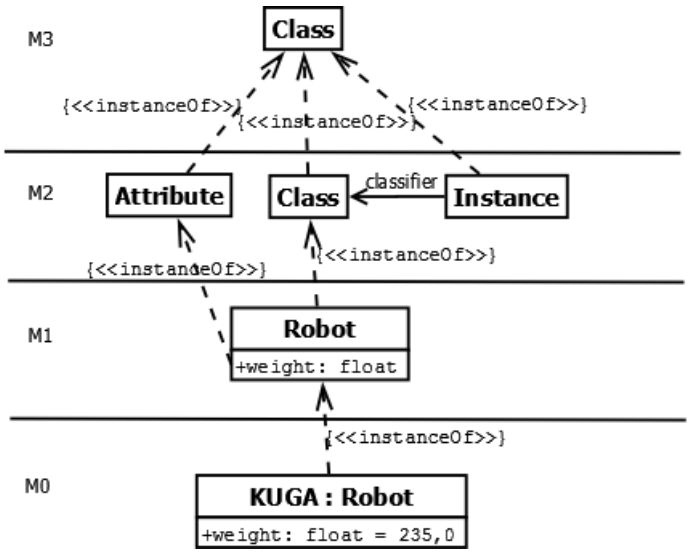

Figure 2.5.: The four layer meta-model hierarchy of MOF depicted on a simple example inspired by the use case explained in chapter 1.

For the OMGs MOF you have objects of the real world in M0. M1 is the model about those objects. M2 comprises UML and models the concept of classes, relations etc. M3 is the highest hierarchy and describes the structure of how classes, attributes and relationships relate together.

2.2.2. BEHAVIORAL DIAGRAMS AND AUTOMATA

The role-play automaton is a behavioral diagram because it models behavior of objects and their relations. To understand design decisions and to understand how it fits into existing behavioral diagrams and automata some UML and non-UML diagrams are introduced. Beside UML diagrams story diagrams will be introduced because they deliver the key concepts for the role-play automaton.

UML ACTIVITY DIAGRAMS

UML activity diagrams give the ability to model activities connected by transitions. It is inherent parallel which means that you can model parallel activities just by forking the control flow. UML activity diagrams allow to add events to transitions as requirement to be activated as well as guards to further constrain the transition. However, the UML standard does not define a clear semantic for this meta-model. Therefore, the Semantics of a Foundational Subset for Executable UML Models (fUML)[2] has been developed to standardize the semantics of UML activity diagrams. Given a fUML instance and a VM the model can be executed. Those instances are interchanged in XMI files. To offer human readable activity modeling the OMG standardized Action Language for Foundational UML (ALF)[3] which is a textual Domain Specific Language (DSL) which in itself is mapped to fUML.

In UML activity diagram's activities can be nested. There is a control flow where set of tokens define the current active state. An activity is activated when it is visited by a token. A transition is fired when a token reaches it. Besides there exist the object flow relates

[2]Semantics of a Foundational Subset for Executable UML Models standard specification website: http://www.omg.org/spec/FUML/1.2.1/

[3]Action Language for Foundational UML standard specification website: http://www.omg.org/spec/ALF/

output pins to input pins. Activities can both return values (results) and write values into given objects which will be carried further e.g., into the next activity.

The UML activity diagram has two types of tokens. One representing the current control flow and another transporting the variables from one activity into another named object flow. Activities are parametrized having input and output parameters and results. Pins at the activity indicate a parameter that has to be bound to an object flow (cf. figure 2.6).

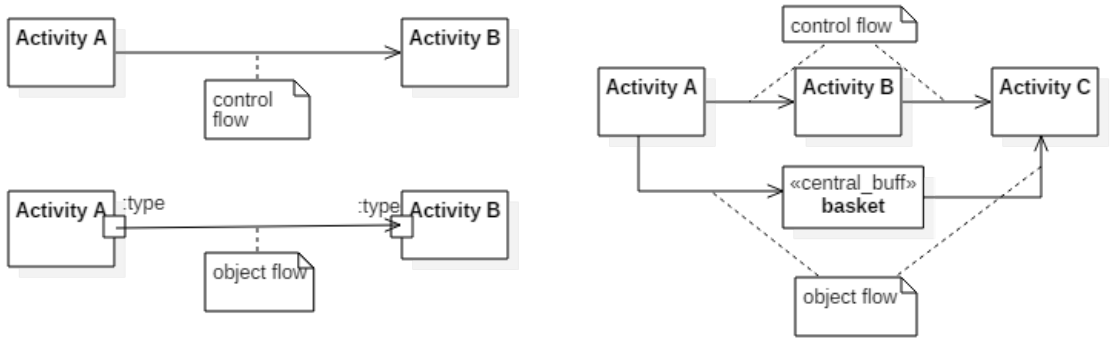

Figure 2.6.: Overview of the syntax of control flow and object flow in UML activity diagrams.

PETRI NETS

Just like activity diagrams petri nets are inherent parallel. petri nets (PNs) [Petri, 1962] make it possible to model and visualize behaviors comprising concurrency, synchronization and resource sharing. Furthermore, they are theoretically founded and thus enables for much more than modeling. It enables for qualitative analyze of the net and there are plenty of further additions by other researchers [David et al., 1994]. Petri nets are similar to UML activity diagrams. They consists of two types of nodes: places, and transitions which are connected by arcs. The arc is directed and connects either a transition to a place or vice versa. There exist non-autonomous petri nets which are either timed or synchronized, and thus cannot be executed alone. Otherwise, they are called autonomous petri nets. A PN is marked which means that every place contains a non-negative integer stating how many tokens are contained in a place. The marking of a point in time defines the state of the PN in that point of time. In the simple case a transition is enabled if every place connected by an arc as input to the transition has at least one token. If there are restrictions on the amount of tokens needed to activate the transition the incoming arc will hold a non-negative integer stating the amount of tokens needed. An enabled transition can be fired, which means it will consume the stated amount of tokens at the incoming places and adds the stated amount of tokens at the outgoing places.

Figure 2.7 shows an petri net. It models two computers sharing the same memory. Thus, only one computer has access to the memory at a time. This is modeled such that if a computer requests access (e.g., computer CP1 at P_1) and the memory is free (a token is at P_7), then T_1 is enabled and can fire. The tokens in P_1 and P_7 will be removed and one token is added to P_2, which means that CP1 is using the shared memory. Hence, computer CP2 could not access the memory at the moment even if it requests access. When CP1 does not need the access to the memory anymore T_2 will fire and the token in P_2 will be removed and a token is added to each outgoing place, P_3 and P_7. If the petri net is an autonomous net T_2 is activated just in the moment P_2 holds a token, which means CP1 is using the computer. The petri net has no means in order to not fire T_2. This would require an extension of the petri net (e.g., using a non-autonomous petri net). Therefore, the PN has many extensions

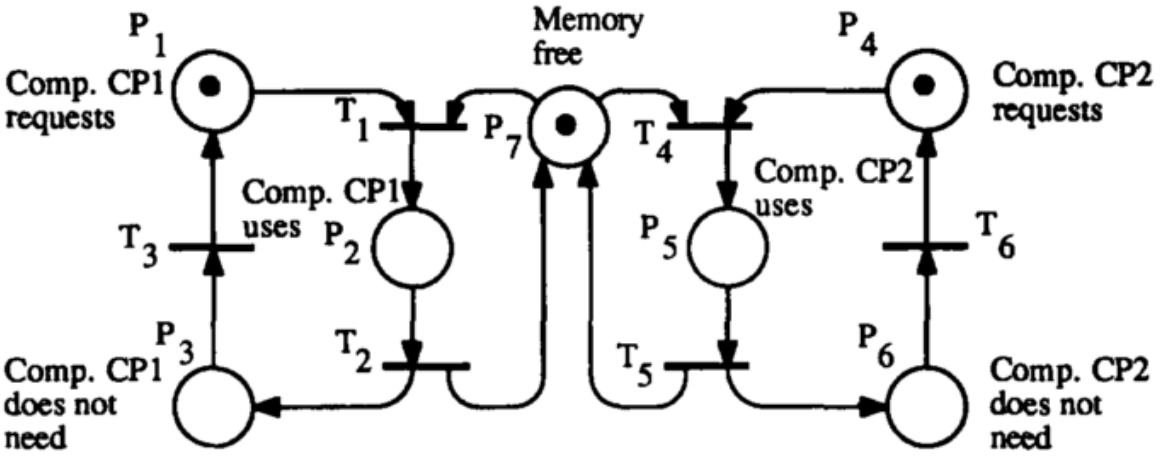

Figure 2.7.: A petri net modeling two computers and a shared storage [David et al., 1994].

like the Colored Petri Net (CPN) which adds colors to tokens so that they can hold different data and yield semantic [Das et al., 1991] and many more variants (e.g., [Genrich et al., 1981] and a survey on petri nets by David and Alla [David et al., 1994]).

As you can see there are many relations between UML activity diagrams and PN. While PN have the mathematical foundation activity diagrams have the simplicity. For complex systems there is a need to analyze the models. Thus, Yang et al. developed a mapping from UML activity diagrams to petri net [Yang et al., 2010] to analyze the modeled activity diagrams.

2.2.3. STORYBOARDS

Storyboards have a long history as they were originally introduces [T. Fischer et al., 2000; Zündorf, 2001]. Since then, there were a lot of researchers talking about storyboards, but used different names like *story diagram* [Detten et al., 2012; T. Fischer et al., 2000; Heinzemann et al., 2011; Zündorf, 2001], or *story board* [Kühn, 2011], while [Luyten et al., 2010] is talking about story boards for user interface design, and *story charts* by [I Diethelm et al., 2002].

Storyboards are theoretically founded on *graph transformation*. It is applied in the development environment FUJABA, which is also called a "*successor of PROGRESS* [Schürr, 1994]" [Fujaba, 2005]. Fujaba's storyboards allow for defining complex manipulations on the object graph with a concise graphical notation. This graph manipulation is achieved by a sequential application of graph transformation rules. Graph transformation systems allow to find specific patterns in the original graph and to modify (e.g., add new nodes or edges, or delete existing nodes or edges) them according to the graph transformation rules. In the last years storyboards have been founded on a meta-model [Heinzemann et al., 2011].

This section is as follows. At first an overview about graph transformation is given, then storyboards and story patterns are introduced. At last, storyboards extended with roles is presented.

FOUNDATIONS OF GRAPH TRANSFORMATIONS

A graph consists of nodes and edges where edges connect two nodes. The graph in this case is an object graph where nodes represent object instances and edges represent relations between objects. Furthermore, edges are directed, which means there is a source and a target node.

A *graph rewrite rule* specifies a left hand side (LHS) and a right hand side (RHS) and a rule isomorphism. The rule isomorphism is needed to apply the graph transformation in order to specify the isomorphism between the LHS and RHS. The application of a graph rewrite rule is called *graph transformation*. The graph the transformation is applied on is called the *host graph*. A graph transformation is applied in three steps: (1) The LHS is searched in the host graph (*match*); (2) Every node which is defined in the LHS but not RHS is deleted from the host graph; (3) every node that is defined in the RHS but not LHS are added to the host graph. Formally, the matching represents the identification of a sub graph on the host graph, whose isomorphic to the LHS. This is also known as the *sub graph isomorphism*, which is a well known problem as it is NP-complete [Eppstein, 1995; Garey et al., 1979]. A more formal definition can be found in [Kühn, 2011] while [Detten et al., 2012] applies the above definitions on graphs of object oriented program instances called *Typed Attributed Graph Transformations*.

STORYBOARDS AND STORY PATTERNS

In model-driven software development the software model is the centered artifact which describes not only the structure of a software but also its behavior. These software models are either executed or used to generate the software described. Therefore, UML delivers structural meta-models and behavioral meta-models (e.g., class diagrams and activity diagrams). The problem with activity diagrams is that the activities are mostly declared in natural language which prevents execution. Thus, story diagrams where proposed by [T. Fischer et al., 2000; Zündorf, 2001] to overcome the problem. The story diagrams replace natural language with a formal language. A story diagram models activities with model transformations, which are classified as endogenous, in-place transformation after the classifications of [Czarnecki et al., 2006]. The control flow consists like the UML activity diagram out of activity nodes and activity edges. The activity node is described as graph transformation language, named *story pattern*. It uses a graph notation to define modifications at object-oriented structures of object-oriented software. Modification means the addition and deletion of objects and relations (called *link*). To explain storyboards first story patterns have to be explained.

STORY PATTERN

The story patterns have a similarity to UML object diagrams. Story patterns represent an object structure which is subject to change. Therefore, the objects and links at the graph, which are going to be added or removed are annotated. This deploys a declarative approach as it says what is subject to change, and not how. These story patterns are based of well known *graph transformation systems* [Rozenberg, 1997]. Hence, story patterns can be analyzed. Story patterns allow to define graph transformation rules where graph transformations according to the defined rules are adding or removing nodes or edges on the host graph. If the graph matching is successful the transformation is applied to the host graph. In the case of storyboards the host graph is an object graph. That means, there exist a type model of the host graph where nodes are object variables, and edges are link variables. This is called a *Typed Attributed Graph Transformations* [Detten et al., 2012]. The graph matching suffers from the *sub graph isomorphism problem*, which is NP-complete [Eppstein, 1995]. Hence, at least one node at the LHS has to be bound. The story patterns are graph transformations which are embedded into activity nodes. Type models are needed to define useful graph transformations on the object-oriented graph. Thus, a typed attributed graph transformation is needed. Then there is a type graph and the nodes have attributes according to the type graph.

To allow concise modeling there is a short hand notation, where LHS and RHS are directly annotated onto the graph elements. To LHS belong all objects and links which are without a

stereotype or stereotyped with `<<destroy>>`. To RHS belong all objects and links which are annotated with `<<create>>` or without annotation.

Story patterns can declare variables with object variables. Those variables represent an object on the host graph. Uniquely identified by their name they form instances of types in the type model which are classes of the host graph. Link variables are relations between objects. They represent references between objects in the host graph. These variables can be *unbound*, *bound*, or *maybe bound*. *Unbound* means that the variable has to be found in the matching when executing the story pattern. A *bound* variable has been bound before the execution or is supplied externally. A *maybe bound* variable will be bound if has not already been bound before. An *unbound* variable has to be stated as `variableName : Type` and for a *bound* variable the type is omitted. Beside the variable binding there is also a binding semantics which is either *mandatory*, *negative*, or *optional*. If a variable is declared as *mandatory* the story pattern will fail if the variable is not matched. Where a *negative* variable will result in a fail if it is matched. An *optional* variable may be found but is not necessary to be bound. The interested reader will find more about variable binding, its semantics and legal combinations under [Detten et al., 2012, section 3.2.4.2.f]. Story patterns allow to define object attributes which can be part of LHS or RHS. Such attributes are formulated with OCL like `<<attributeAssignment>> ::= #Attribute.name ':=' Expression`.

STORYBOARD

Storyboards do not just model the structure of software, but also its behavior. Thus, the model is executable. Storyboards are a combination of graph transformation and UML activity diagrams. The control flow are activity nodes and activity edges. An activity node is either a story pattern or an activity call. Edges may are constrained by guards which are boolean expressions over bound variables or keywords like `[success]` or `[failure]`. Storyboards do not allow to model concurrency as *fork* and *join* nodes of UML activity diagram are not supported.

Beside the theoretical foundation of storyboards on PROGRES, Fischer et al. translated storyboards to Java using Fujaba development environment [T. Fischer et al., 2000]. This will make them loose the backtracking semantics, but gives the ability to work seamless with parts of existing systems. Storyboards have just borrowed the semantics of UML activity diagrams, but did not implement the storyboard using them. For the execution of storyboards UML 1.5 is used with the semantics of UML 2.0 [Detten et al., 2012]. However, it is still criticized that UML activity diagrams have no formal semantics and thus are not an executable model. This has been the initial reason to develop storyboards. To give an alternative approach on executable models this section also introduced petri nets which are formally founded and thus executable. Since early 2011[4] fUML has been released which proposes a standardized semantics for an executable subset of UML including UML activity diagrams, and a reference implementation to validate the applicability of the standard.

STORYBOARDS WITH ROLES

Storyboards with roles are an extension over storyboards proposed by Thomas Kühn [Kühn, 2011]. The thesis lists some problems of storyboards such as there is no possibility to physically delete objects. There is always the possibility that there exist a reference onto the object, thus the object will not become garbage collected. Furthermore, the construction of objects is not guaranteed as there is no possibility to select a specific constructor. [Kühn, 2011] proposes roles as solution to all these problems. Roles are easier to construct and to delete. For construction the standard constructor is enough. Deletion is done by removing

[4] fUML website: http://www.omg.org/spec/FUML/ visited at 2016/06/03

the role from the player and the context. Thus, the role is not referenced anymore and will become eventually garbage collected.

The host graph is not the object graph anymore, but the role-play graph. That is, a graph which consists of three types of nodes. First the player node, which represents an object taking part in a collaboration. Second the role node, which represents a role that is played. And third the compartment node, which represents the context in which the player plays the role. The difference to storyboards lies in the story patterns.

The story patterns are extended to state bound and unbound objects and required or permitted roles. Links may be assigned between roles which means there is a relationship between these roles. The story pattern has to check if these links are allowed w.r.t. the role model. Thus, the type model for storyboards with roles comprises the class model and the role model. An example is shown in figure 2.8 (a).

To make the model more concise the `<<create>>` and `<<destroy>>` stereotypes are replaced by annotations directly to the role nodes (see figure 2.8 (b)). A star means the role is added to the object, where a stroke through role means it is removed. The crossed role means the role is forbidden and the untouched role means a match. The links are not annotated anymore, too. Their semantics have to be deduced from the context. If a link is between any required or permitted role and a remove role the link will be removed, too. If the link is between a required or permitted role and a new role then the link will be created, too.

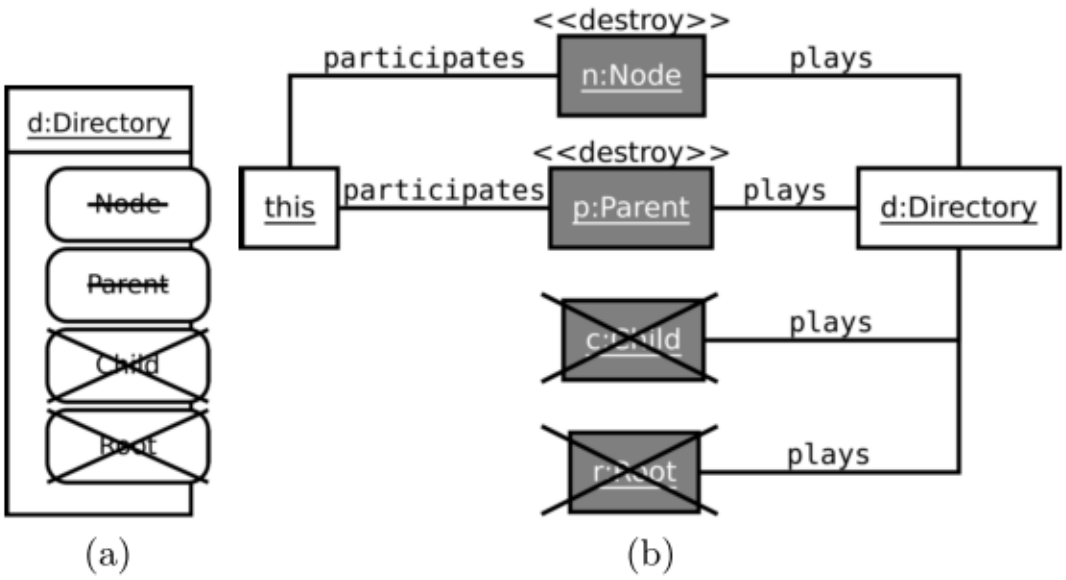

Figure 2.8.: Storyboard with roles of Thomas Kühn [Kühn, 2011]. (a) shows the new concise syntax which is equivalent to the model on the right side. (b) shows the concept of storyboards applied to roles. A player plays roles.

Kühn specifies conditions to further constrain the match. Therefore, internal conditions are defined which allow to define constraints on the attributes of the bound or unbound variables and roles. External conditions allow to further restrict the match with arbitrary boolean constraints. Both, internal and external conditions are part of the LHS.

Like the original storyboards do storyboards with roles require at least one variable to be bound to begin execution.

2.3. RELEVANT SOFTWARE ARCHITECTURES

This section will introduce relevant software architectures the thesis is concerned about. It will explain Context-Aware Computing, Self-Adaptive Systems, and Event-Based Systems and will relate them together. At the end I will show that all three architectures need to be integrated together to fulfill the need in ubiquitous environments such as handhelds.

2.3.1. CONTEXT-AWARE COMPUTING

Since its first discussion in 1994 [Schilit et al., 1994] context-aware computing is getting more and more important. In today's world of ubiquitous and pervasive computing many activities take part in computational environments which can be tracked and analyzed by software [G. Fischer, 2012]. Context outside of these computational environments must be either sensed or provided by a dedicated person. Context-aware computing includes the context into its computation. There have been many definitions of context e.g., as the situation (people who are involved and the objective of the interaction), the location, environment and time of an entity [Nanda et al., 2016]) which are almost too specific [Dey and Abowd, 1999]. Therefore, Anind Dey defined context as:

> "[...] any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves." [Dey, 2001]

Therefore, he combines self-awareness and the awareness of the surroundings with the awareness about what is *relevant* to the interaction. This may be used to either reduce the amount of information, increase the quality of information or to offer useful interactions (user interface). These are all criteria of adaptiveness. This context-awareness often comes together with self-adaptiveness which will be described in the next section. Sama et al. name these systems Context-Aware Adaptive Applications (CAAAs), which are supported by a context-aware middleware, which offers an event-driven context manager for collection and querying the context, and an adaption manager to apply application adaption [Sama et al., 2008].

There are many different software architectures to realize context-aware systems, but still most frameworks and middleware rely on event-based approach [Barrenechea et al., 2011].

2.3.2. SELF-ADAPTIVE SYSTEMS

A Self-Adaptive System (SAS) can modify itself in the presence of external or internal changes it is observing. This implies, that the system is context-aware, or at least self-aware. [Krupitzer et al., 2015] found three different types of SAS (i) the traditional *reactive* approach, where the application reacts on the occurrence of events, (ii) the *predictive* approach, where the application decides the need of adaptations before and event happen, and (iii) the *proactive* approach, where the application decides to adapt without any indication that an event will happen. Sama et al. name these systems Context-Aware Adaptive Applications, which are supported by a context-aware middleware, which offers an event-driven context manager for collection and querying the context, and an adaption manager to apply application adaption [Sama et al., 2008].

De Lemos et al. state that "*[i]n addition to the ever increasing complexity, software systems must become more versatile, flexible, resilient, dependable, energy-efficient, recoverable, customizable, configurable, and self-optimizing by adapting to changes that may occur in their operational contexts, environments and system requirements*" [De

Lemos et al., 2013]. To achieve this adaptability decisions need to be made in the design space of the software system. Thus, most state-of-the-art adaptive systems are built around a control loop [Andersson et al., 2009]. Since this loops measure the system and the success of undertaken adaptations they are also called *feedback loops*. Brun et al. analyzed different types of self-adaptive systems regarding feedback loops [Brun et al., 2009]. The remainder of this section will talk about the *MAPE-K* loop, which is one of the well recognized engineering methods to realize self-adaptation [Arcaini et al., 2015]. MAPE-K (monitor-analyze-plan-execute over a knowledge base) has been found by IBM [IBM, 2005].

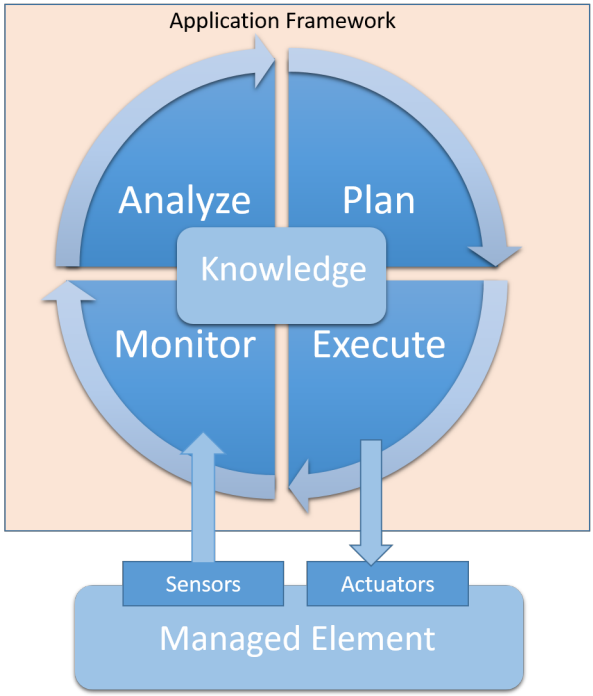

Figure 2.9.: MAPE-K feedback loop [IBM, 2005] with its four phases (monitor, analyze, plan, execute), the knowledge base, and an element that is monitored and adapted.

Figure 2.9 depicts the MAPE-K loop together with a monitored and adapted element. In the *monitor* phase environmental data is collected, for example from sensors, and is analyzed in the *analyze* phase. In this phase events can be fired e.g., if the sensor data shows that a person entered the working area of a robot. Using Event-Condition-Action (ECA) rules an adaption goal can be calculated. The *plan* phase decides if there is need for reconfiguration, and creates reconfiguration plans if necessary according to the adaption goal. These plans are executed in the *execute* phase. They all share the same *knowledge* base, which consists of all necessary information about the environment, i.e. the *context model* [Barrenechea et al., 2011; Piechnick, Püschel, et al., 2014].

As stated in section 2.1.1 a role is a dynamic service of an object in a specified context. Roles are played by an object, dynamically altering the behavior and structure of an object. Roles are always played in a context. This allows to adapt the object by means of the roles it is playing. Section 2.1.3 introduced SMAGs, which is a system for dynamic and unanticipated adaptation using a variant of the MAPE-K feedback loop and role-based progamming [Piechnick, Richly, et al., 2012]. Thus, SMAGs adapts objects at runtime by the use of roles.

2.3.3. EVENT-BASED SYSTEMS

The central element of Event-Based Systems (EBSs) are asynchronous events. Distinct components of the system react independently of the current state of the system to events. This makes EBSs a convenient approach for distributed systems. The interaction between the event producer and consumer is loose and the systems are decoupled [Dustdar et al., 2013]. The event producer sends the event to the system which will distribute the event to all interested sub-systems. Publishing an event to the system can be seen as broadcast to the interested sub-system without any prior knowledge. There are different dimensions events comprise. Events could have temporal properties and be just valid during a constrained timespan. There could also be complex events which are generated after a sequence of events occurs. Therefore, there have to be a relation or a fuzzy relation between the events in a given timespan. For the topic of the thesis Complex Event Processing (CEP) will be highlighted in detail. CEP has become the paradigm of choice when dealing with monitoring or reactive applications [Liu et al., 2010]. The advantage is that the CEP system decouples the continuous stream of e.g., sensor information of RFID chips and the processing system. More exact the producer does not know the receivers and the receivers do not have to know about the data nor the event sources. The CEP decouples producer and receiver. Furthermore, it can detect coherences between events such as temporal relations and create complex events out of this. These can be specified with event patterns.

2.4. SUMMARY

Class-based programming is good at capturing the structure of the real world, but suffers to capture collaboration of objects exhibiting at runtime. Role-based programming uses the class-based approach and extends it with roles. Roles are played by an object, dynamically altering behavior and structure of the object. Since roles are always played in a context, they are good at capturing collaborations of objects at runtime. Beside theory, there are different approaches how to implement roles which have their pros and cons. Using Aspect-Oriented Programming allows for separation of roles and classes, but is a static approach. An approach that can be directly embedded into object-oriented programming is the Role Object Pattern. The problem with ROP is that is exhibits object-schizophrenia. Because the concept of roles is very promising, there have been many attempts to implement runtimes which allow to program with roles and execute the code. The most important are ObjectTeams/Java, which uses implicit role binding, and EpsilonJ, which uses explicit role binding. The first allows to bind roles implicit to objects, regarding a type system and type inference. The latter allows to directly state role bindings in the code, thus gives more flexibility. The problem with EpsionJ is, that the programmer needs to be aware of the context itself. Another solution is Smart Application Grids. Models and their meta-models form a hierarchy which is called the "meta-model hierarchy". It has been implemented in the four layer hierarchy by MOF. UML activity diagrams allow to model software behavior. Activities are interconnected by a control flow. However, they do not allow to specify the software model formally in order to execute the software model. Because there is no execution semantics for UML activity diagrams fUML has been developed to deliver an execution semantics including a subset of UML activity diagrams.

Petri nets are a formally founded alternative which allows to execute the model. The many additions researchers have introduced to petri nets (e.g., timed petri nets, synchronized petri nets, colored petri nets) make it a viable tool for many domains. Beside execution, the formal foundation allows to analyze the model. The problem is that there is no common execution software for petri nets. Even if there would be a framework to execute petri nets the many specializations would hinder execution. Every change to the underlying meta-model would

require an adaption of the framework or a new framework itself.

Storyboards want to close the gap between software models and their behavioral specifications in UML activity diagrams as they propose formally founded story patterns which are embedded into activity nodes. Thus, allowing to execute the software model. However, they have inherent problems, because the deletion of objects is not guaranteed, as well as there is no possibility to choose a specific constructor for object creation [Kühn, 2011]. Hence, Thomas Kühn proposed storyboards for roles. Roles can be deleted by dropping the player and removing them from the context. Furthermore, roles do not need specific constructors. Thus, the standard constructor may be used.

Self-Adaptive Systems today almost always comprise context-awareness. Sama et al. [Sama et al., 2008] supports this statement as they call them CAAAs. Because of the ubiquitous environment the software has to be context-aware, and continually adaptive to context changes. One of the well recognized engineering approaches to realize self-adaptive systems is the MAPE-K loop. It allows to define phases for monitoring, analyzing the gathered data, prepare an adaption plan is necessary, and last execute the plan. Piechnick et al. [Piechnick, Richly, et al., 2012] have shown that using roles allows for dynamic and unanticipated adaption. Sama et al. have shown that self-adaptiveness almost always need context-awareness. While most context-aware systems today use the event-based approach [Barrenechea et al., 2011] it follows that context-awareness, self-adaptiveness, and the event-based approach together form one whole system.

3. REQUIREMENTS ANALYSIS

This chapter will analyze the requirements of the system designed and implemented in this thesis. Beforehand, there was an exhaustive analysis. The analysis encompasses talks with faculty members, the analysis of storyboards and storyboards with roles (see section 2.2.3), and the analysis of context-aware and self-adaptive systems (see section 2.3). The developed system should be used in the domain of context-aware, self-adaptive systems. Storyboards (section 2.2.3) have stood out to be able to formulate complex scenarios with a concise syntax. The extension of storyboards with roles have shown their theoretically applicability for role-reconfiguration in the major thesis of Thomas Kühn [Kühn, 2011]. The thesis will therefore show that the concept of storyboards with roles can be extended to be used in conjunction with context-aware, self-adaptive systems. Hence, the concept of storyboards with roles has to be extended to accomplish the new tasks required in the new domain. The result of this thesis will be *context-aware storyboards with roles* which fulfill the thesis' topic as *context-aware role-playing automaton for self-adaptive systems*. Therefore, the problem with current approaches are stated and a goal is formulated from which the requirements of the developed system are derived. Thereafter, a technology selection is conducted which will introduce and select current implementations of different frameworks. These frameworks are candidates for specific tasks handled by the system. The selection is conducted out of frameworks from the same technical space as the system. At last, a summary is drawn.

3.1. PROBLEM ANALYSIS

Developing context-aware, self-adaptive systems with role programming languages (cf. section 2.3) exhibits the problem of mixing different aspects of a software system. The problem arises when programming role reconfiguration rules depending on external context (beside the context the roles are played in). This adds at least one more dimension (whereby external context can be multi-dimensional, too e.g., time and place [Piechnick, Püschel, et al., 2014]) to the role binding constraints. Thus, different concerns (i.e., business logic, context logic, role adaption logic) are tangled in the code base.

Graphical role-oriented modeling can help to formulate scenarios with a concise meaningful syntax. Therefore, Thomas Kühn has shown that Storyboards with roles [Kühn, 2011] can formulate role-reconfigurations with a concise syntax. However, they lack the ability to address context-relevant conditions. Moreover, they are used to specify the adaption process within a *method* of a system. That is too narrow, as in the context of SAS we want dynamic, unanticipated adaption. The need of specific methods restricts the system too much.

The topic of the thesis is to design and implement a *context-aware role-playing automaton for self-adaptive systems*. SAS, which are almost always context-aware systems [Sama et al., 2008], as well as these frameworks or middleware relies on an event-based approach [Barrenechea et al., 2011] (cf. section 2.3.1). In the presence of SAS the most applied architecture is a variant of the MAPE-K loop (see figure 2.9). Normally, the plan phase consists of selecting actions that need to be applied to the system. These actions are supplied by a high level component that describes an adaption plan. These plans are written in ECA rules.

Storyboards as introduced in section 2.2.3 are translated into Java code. Thus, the outcome is one method for a whole storyboard consisting of many `try-catch` blocks [T. Fischer et al., 2000]. The *success* and *failure* transitions are programmed as variables. For the system developed within this thesis, a more sophisticated implementation should be used leveraged from the Model-Driven Software Design (MDSD) domain. The context-aware storyboards with roles will be translated into UML activity diagrams. These diagrams will be subject to execution. This concept is not new, as Diethelm et al. already used UML activity diagrams [I Diethelm et al., 2002]. However, they did not execute them, but generated standard Java code.

3.2. GOALS AND REQUIREMENTS

Since storyboards with roles excel in stating role-reconfiguration, and in order to address the problems stated in the prior section we have to extend storyboards with roles to become *context-aware storyboards with roles*. Furthermore, the SAS has to outsource role-adaption planning to the storyboard, which will perform the role-reconfiguration planning. Thus, the context-aware role-playing automaton has to be integrated into the phases of the MAPE-K loop. Figure 3.1 shows the adapted MAPE-K loop, where the *plan* phase is exchanged by a context-aware role-playing automaton. The difference is now that the automata is supplied with information about the context and reconfigures roles based on this information.

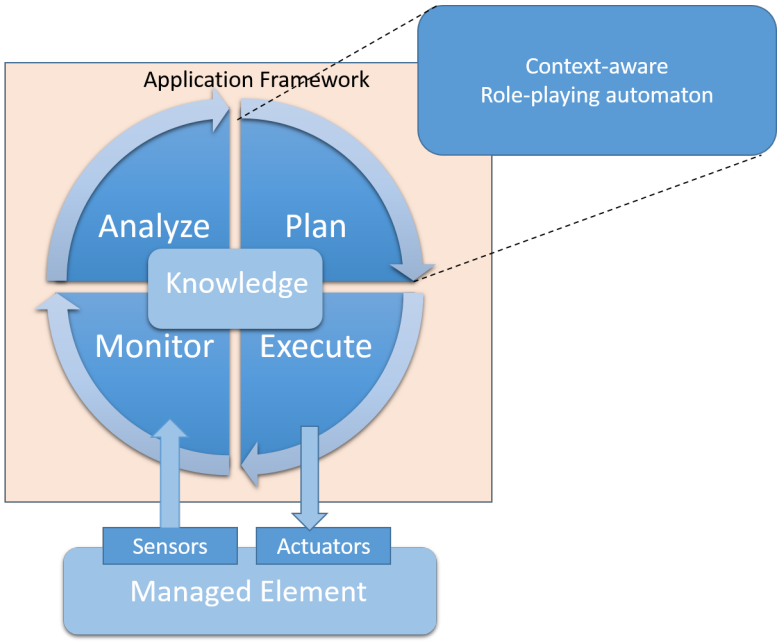

Figure 3.1.: MAPE-K feedback loop [IBM, 2005] with its four phases (monitor, analyze, plan, execute), the knowledge base, and an element that is monitored and adapted. The original plan phase is replaced by a context-aware role-playing automaton.

Since many context-aware systems use the event-based approach it is possible to state context-changes of any kind as an *event*. Such an event represents a happening in the real world. The events can be received from sensors, or generated out of gathered data (see complex-event-processing, section 2.3.3).

This list encompasses the goals. The *system* means the system developed in this thesis i.e., context-aware storyboards with roles.

G-1 **Extend storyboards with roles with compartments.** The role community developed further since the definition of storyboards with roles in 2011. Therefore, the new notion of context, also known as compartment, has to be integrated into the storyboards with roles.

G-2 **Extend storyboards with roles with context.** The storyboards with roles are not context-aware. In order to be used for planning of role-reconfiguration for self-adaptive systems, they have to be able to access the context in some way or another.

G-3 **The system need to be integrated into the MAPE-K loop, especially into the plan phase.** The planning of role-reconfigurations should be adopted by the context-aware storyboards with roles. Therefore, they need to be adaptable into the MAPE-K loop, especially the plan phase.

G-4 **The system should not hinder unanticipated adaption.** Thus, they should be modeled as automata and not in the meta-level of methods.

G-5 **The system should be independent on specific role runtime.** The adaption logic of the SAS should fulfill adaption. This is beyond scope of the system.

G-6 **Graph Transformation.** A story pattern depicts a graph transformation rule. Therefore, it must be possible to find the specified patterns.

The goals are a high-level overview about what need to be done to remedy the problems shown above. The following list formulates requirements out of the goals shown above. These requirements will be used in chapter 4 to formulate the concept, and in chapter 5 to drive the implementation.

R-1 **Compartments.** Roles are played in a compartment. Therefore, the syntax and semantics of role-playing should be extended to consider compartments. This would satisfy G-1.

R-2 **Context.** SAS almost always use a variant of the MAPE-K loop. Therefor, context should be an element of the knowledge base i.e., context model. Furthermore, most context-aware systems use the event-based approach. Thus, the system could be forwarded by events. This would comply with the use of UML activity diagrams which use events on transitions. Storyboards are already defined with UML activity diagrams in mind. The events can be extended to *ContextEvents* which allows access to the context objects.

R-3 **Storyboard as automata.** Storyboards have been defined as models which combine UML activity diagrams and graph transformation to form an executable model. However, the system should not hinder unanticipated adaption (G-4). Thus, the system will be run on its own and will be adapted to the SAS (G-3). This allows to satisfy those two goals.

R-4 **Runtime independence.** The system should not be dependent to a specific role runtime. Section 2.1.1 introduced some role runtimes. The user should be able to adapt the system to use any role runtime. This would satisfy G-5.

R-5 **Graph transformation. Negative queries.** Story patterns depict graph transformation rules. Therefore, it must be possible to query the role-play graph to find the specified patterns. Moreover, it must be possible to state negative queries. This satisfies G-6.

3.3. TECHNOLOGY ANALYSIS AND SELECTION

Chapter 2 has introduced the concepts of storyboards, and storyboards with roles (cf. section 2.2.3). These concepts employed some special concepts themselves, namely the matching of patterns on a graph, and the transformation of a model to executable code (e.g., model execution or code generation). Hence, this section will evaluate existing implementation of these concepts and will choose one of each of them. There exist many different implementations that can be used to achieve the same. They all have their pros and cons. This section will evaluate implementations from the same technical space (e.g., Java programming language[1]) against a relevant subset of the criteria. Thus, tools outside of Java will not be evaluated.

At first technology to implement graph search is introduced. It is shown with a simple running example how the technology can be used. Furthermore, a conceptual adaption of these technologies is presented and evaluated w.r.t. its complexity, possibility to implement, and its maintainability. Thereafter, it is conducted how the storyboard instances can be represented in order to be executable models. Therefore, petri nets and UML activity diagrams are discussed. Then, UML activity diagram execution engines are presented. At last the technology selection is presented.

3.3.1. PATTERN MATCHING

Pattern matching has been introduced as part of the foundation of graph transformation (cf. section 2.2.3). The task of this section is to find implementations for graph search and to choose one that will be used in the implementation (cf. chapter 5).

The application needs to find the relevant objects (if any) that fulfill the story pattern. Therefore, the framework needs to be able to search a probably big graph and in the worst case with multiple fix points (e.g., starting from instances of a given class, or every object that plays some role). It has to be able to be adapted to the queried graph (e.g., the role-play graph) and the query language must be able to express negative queries (e.g., for forbidden roles). In regular relational SQL databases this will result in an outer join which will draw significant performance [Kühn, 2011]. Therefore, just graph databases or graph query frameworks are regarded. Furthermore, the structure of the story pattern (LHS, RHS) has to be translated into the frameworks DSL to query the role play graph.

GUERY FRAMEWORK

Guery Framework[2] is a pattern matching framework [Dietrich et al., 2012]. It offers the ability to define *motifs* and search motif instances on arbitrary graphs given the user implements an adapter specifying the implementation of nodes and relations. A motif is a term used in bio-informatics which relates to the motif term of biochemistry. A *structural motif* is a pattern of in a protein structure, and a *sequential motif* is a sequence pattern of nucleotides in a DNA sequence. However, both are patterns in a structure that could be represented as a graph. For a more complete and formal definition of a motif, and graph of motifs see [Dietrich

[1] Java programming language by Oracle: https://docs.oracle.com/javase/8/docs/technotes/guides/language/ website visted at 2016/06/06

[2] Guery Framework (new adapted version): https://github.com/lschuetze/queryframework

et al., 2012]. Guery uses the JUNG graph library [O'Madadhain et al., 2003]. Other graph search frameworks need to load the whole graph into memory in order to perform searches. Guery uses the adapters to conduct a graph search. For very big graphs it offers a streaming API allowing to search on graphs that would otherwise exceed memory capabilities of the executing computer. Furthermore, Guery supports parallelisation in the evaluation of queries. Negative queries are directly integrated into the framework's query DSL. The DSL allows for complex queries and even allows to formulate transitive closures. Figure 3.2 shows a conceptual framework and needed adaptions in order to connect these frameworks. Such a scenario would require substantial effort to implement the adapters. The JUNG graph framework requires an adapter to the graph to be searched. Furthermore, the context-aware storyboards with roles have their own view on roles, compartments and their relationships. This has to be adapted to the role-runtime used by the self-adaptive system. Because Guery does not provide a manipulation language the adapter need to forward changes on both sides to the other. To keep it simple the negative query is omitted. However, guery uses a simple `not connected by`. A problem is that the graph uses a single type for vertices and edges, thus force to double specify each vertice by its type and its class. Similar for edges that have to state its edge type.

```
1  motif RobotPlaysAutonomously
2  select natural,role,compartment
3  where "natural.type=='player'" and "natural.class=='Robot'"
4    and "role.type=='role'" and "role.class=='Autonomous'"
5    and "compartment.type=='compartment'"
6    and "compartment.class=='Collaboration'"
7  connected by plays(natural>role)[1,1]
8    and in(role>compartment)[1,1]
9  where "plays.type=='plays'" and "in.type=='compartment'"
10 group by "natural"
```

Listing 3.1: A Guery motif to find a robot that is in autonomous mode in the collaboration compartment.

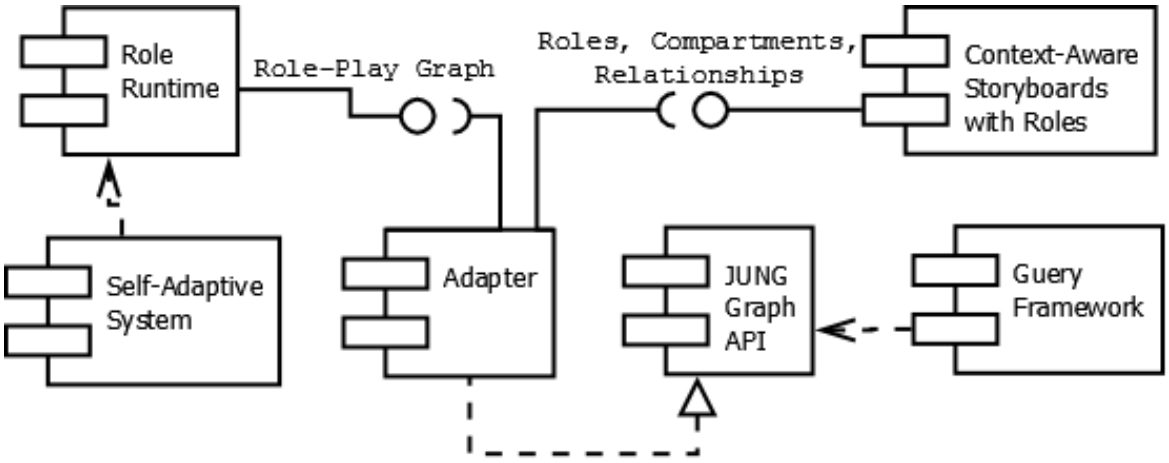

Figure 3.2.: To connect the role-runtime graph and the view on roles, compartments and their relationships an adapter [Gamma et al., 1995] has to be programmed.

NEO4J

Neo4j[3] is a graph database which can be used directly from Java. A graph database allows to store graphs as it emphasizes the relation between elements as data model [Miller, 2013]. Hence, the relations between the data is as important as the data itself. Neo4j's query language is called *Cypher*[4], and is specific in the regard that it is a declarative language. The user can specify what he is searching, and does not have to worry how to search it (e.g., database specific things like indices do not have to be worried about). For example, listing 3.2 shows how to query Neo4j to find a player of the class robot which is playing the autonomous role in the collaboration compartment. As you can see you can formulate typed queries in a syntax relative similar to the query definition in natural language. Being a database, Neo4j offers the advantage that negative queries can be formulated more efficiently. Taking the example from above you can state negative queries as `OPTIONAL MATCH` which allows to check if a relation is `null` and thus not existent. This example is shown in listing 3.3.

The problem with Neo4J is that the role play graph has to be loaded into the database. This graph is outside of the scope of the system developed in this thesis and therefore manipulation is not detected, thus forcing to re-importing the graph into Neo4J on every single access. Otherwise, an adapter could be implemented from the role-runtime to the graph database which updates the database and the graph respectively. This is an easier scenario as described by using Guery.

```
1 MATCH (robot : Player) -[:plays]-> (Autonomous) -[:in]-> (
    Collaboration)
2 WHERE robot.class = {"Robot"}
3 RETURN robot
```

Listing 3.2: A Cypher query to find a robot that is in autonomous mode in the collaboration compartment.

Figure 3.3 shows how the adaption could be achieved. A database is always managed by an instance of `GraphDatabaseService` at the user code level. This would be part of the adapter. The adapter could register an `TransactionEventHandler` who listens to the database service which will be notified for every transaction. As soon as the storyboard modifies the role-play graph the transaction event handler will be notified and get access to the `TransactionData` of the transaction. Thus, it could modify the specific instance in the role-play graph. However, when the role-play graph is changed externally, the adapter has to be notified. A *decorator* [Gamma et al., 1995] cannot be used, because we cannot change role-play graph element instantiations in the role runtime. A possible solution is to use AOP with AspectJ to implement the adapter to the role-play graph. However, this would have to be done for every role-runtime that should be used separately.

```
1 MATCH (robot : Player) -[:plays]-> (Autonomous) -[:in]-> (
    Collaboration)
2 WHERE robot.class = {"Robot"}
3 WITH robot
4 OPTIONAL MATCH (robot) -[c:carries]-> (object)
5 WHERE c IS null
6 RETURN robot
```

Listing 3.3: A Cypher query to find a robot that is in autonomous mode in the collaboration compartment and not carrying an object at this time.

[3]Neo4J Graph Database: http://neo4j.com/
[4]Cypher manual: http://neo4j.com/docs/stable/cypher-query-lang.html

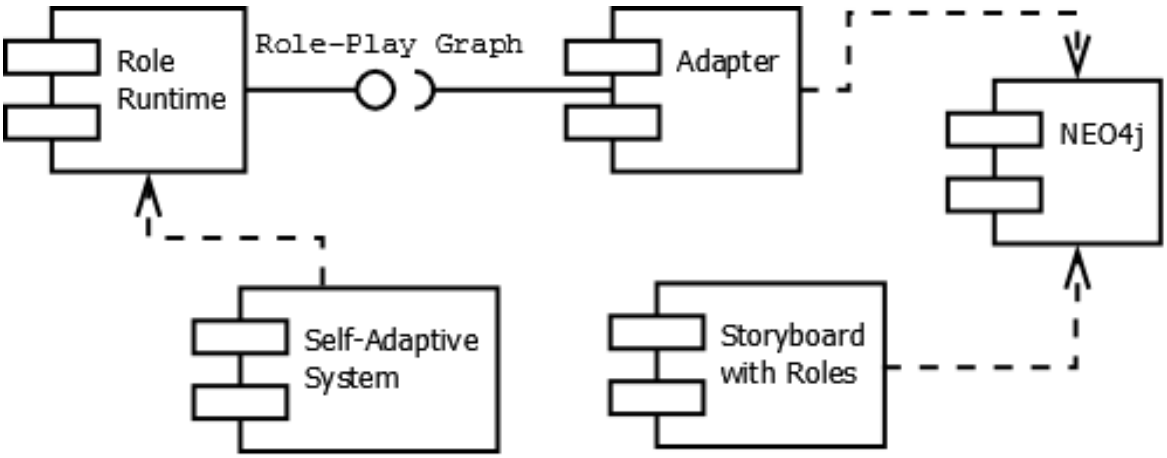

Figure 3.3.: To connect the role-runtime graph and the view on roles, compartments and their relationships an adapter [Gamma et al., 1995] has to be programmed.

QUERY AND MANIPULATION API

A query and manipulation API would be a third alternative to implement the graph search. Therefore, the overall search would be reduced to method calls on the query API for the LHS, and to method calls on the manipulation API for the RHS. Instead of manipulation API it will be called *adaption API* to emphasize the original means. Figure 3.4 shows the conceptual model to implement this scenario.

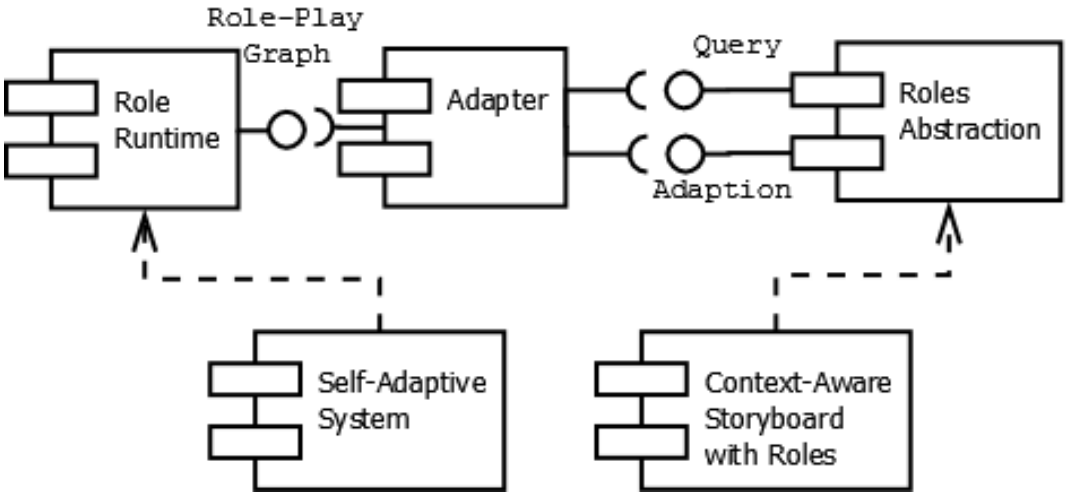

Figure 3.4.: To connect the role-runtime graph and the view on roles, compartments and their relationships an adapter [Gamma et al., 1995] has to be programmed.

In section 2.1.1 Compartment Role Object Model (CROM) and Compartment Role Object Instances (CROI) have been introduced. Both are part of the Role-based Software Infrastructures (RoSI) project and provide an interface for their respective meta-level. CROM offers the `IEvolution` interface, while CROI offers the `IAdaption` interface. It does not offer complete implementations w.r.t. these interfaces. The `IEvolution` interface offers methods to query the role model and ask for role types and their relations. The `IAdaption` interface works on the instance level. Hence, it offers methods to ask if a player plays a role in a compartment, or to add links between roles, and create or remove a role from a player. Both mix query and adaption. Because of separation of concerns, both interfaces for L0 and

L1 need to be divided into a query and an adaption interface. For further discussion about `IAdaption` please see the chapter 5.

```
List players = query.getPlayersOfClass("Robot");
List compartments = query.getCompartmentsOf("Collaboration");
for(c : compartments)
  List roles = query.getRolesInCompartment(c);
  for(r : roles)
    if(r.roleType == "Autonomous")
      for(p : players)
        if(query.plays(p, r, c)) then return p;
```

Listing 3.4: A Query and manipulation API pseudo code to find a robot that is in autonomous mode in the collaboration compartment.

Listing 3.4 emphasizes in pseudo-code, that we first have to find the right instance of `Robot`, the right instance of `Collaboration`, and have to ask if the player plays the role in that compartment. Please note, that in very big role-play graphs this could load to a complexity of $\mathcal{O}(\text{query}) = \mathcal{O}(\mathcal{C}) * \mathcal{O}(\mathcal{R}) * \mathcal{O}(\mathcal{P})$, where $\mathcal{C}$ is set of compartments, $\mathcal{R}$ set of roles played in each compartment, and $\mathcal{P}$ set of players of class `Robot`, which is cubic complexity at worst-case. However, section 2.2.3 introduced graph transformation as possible NP-complete problem. That does not apply in this case, as there are many fix points like specific instances of player classes, specific role types and type of compartments.

3.3.2. MODEL EXECUTION

Storyboards have always been with UML activity diagrams in mind [T. Fischer et al., 2000]. Since the storyboard has to be executable and its semantics is partially similar to UML activity diagrams the idea to transform the storyboard diagram into an activity diagram seems obvious. This emphasizes the Model-Driven Software Development (MDSD) approach where storyboards fit good into. However, previous approaches that implemented storyboards did not use MDSD approaches, but generated standard Java code. This thesis will generate a model which is subject to execution.

Petri nets have been evaluated as well, because they are grounded on formal models. However, there is no general approach on executing petri nets. Jungel et al. have defined Petri Net Markup Language (PNML) to describe petri nets [Jungel et al., 2000], that could be used as an interchange format. But there is no standard tooling on executing petri nets. Approaches like the petri net kernel date back to 2002 [Kindler et al., 2001].

Because of this reasons there were considered free, open source implementations for UML activity diagrams for their suitability. Solutions found are the fUML reference implementation, and the Eclipse Moka project both providing an execution engine for fUML models. The ALF reference implementation compiled Alf code to Eclipse UML2 XMI files (as interchange format for model instances) which could be imported and executed by Eclipse Moka, too. Therefore, there will be just a brief introduction of Alf. fUML and Eclipse Moka will be introduced in more detail.

FUML REFERENCE IMPLEMENTATION

fUML has the goal to develop a precise semantic specification for a subset of UML. This includes parts of the superstructure such as classes, behaviors, activities, and actions, but excludes state machines and sequence diagrams. The fUML reference implementation[5]

[5]fUML reference implementation: https://github.com/ModelDriven/fUML-Reference-Implementation

has the goal to implement an interpreter for fUML model instances and was created with the draft specification. It requires an XMI model file and executes the given activities. The runtime outputs the activities that have been executed (also called an execution trace). In order to be used as a compilation target for the storyboards developed in this thesis it needs the ability to accept events from outside (R-2), and execute arbitrary code from activities (e.g., call activities). Furthermore, transitions have to be guarded. However, the fUML standard does not allow guards on arbitrary edges, but if the source of the transition is a decision node [The Object Management Group, 2012, p. 58].

ALF REFERENCE IMPLEMENTATION

The ALF reference implementation[6] for the Action Language for Foundational UML standard is a textual DSL which semantics is defined by its mapping to fUML. It allows to program UML activity diagrams which can be executed later. The ALF standard allows in-line statements [Object Management Group, 2013, p. 106f] in actions, thus adding code in an external language than Alf. The reference implementation implements the in-line statement on the meta-model level but does not generate executable code but an exception. Listing 3.5 shows how Alf code can be used to specify activities.

```
1 namespace Roboter::Collaboration;
2
3 private import Collaboration;
4 private import RoboterWorkplace;
5
6 //self is bound to robot
7 activity collaborate(in w:Worker) {
8   self.collaborateWith = w;
9 }
```

Listing 3.5: Alf code representing an acitivity.

ECLIPSE MOKA

Eclipse Moka[7] as part of the Eclipse Papyrus project aims to provide a visual user experience for animating and debugging UML activity diagrams [Guermazi, Tatibouet, Cuccuru, Dhouib, et al., 2015; Seidewitz, 2015]. It contains a fUML execution engine. It allows to set breakpoints in the model code and is integrated into the Eclipse debug framework. Since Moka build on top of the fUML standard it will not have the ability to add external events either. Figure 3.5 shows a screenshot of Eclipse Moka and tags the different views that are available during a debugging session. Since Moka is integrated into Eclipse, it allows to use the Eclipse debug framework. Therefore, execution of an activity diagram can be stepped through.

3.4. SUMMARY

This chapter introduced a deeper understanding on how storyboards fit for role-binding in context-aware and self-adaptive systems. The problem domain and solution domain was explained. Therefore, it was shown how the system developed in the thesis can be integrated with the MAPE-K feedback loop, that many self-adaptive systems use. Furthermore, it was

[6] ALF reference implementation: https://github.com/ModelDriven/Alf-Reference-Implementation
[7] Eclipse Moka: https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution

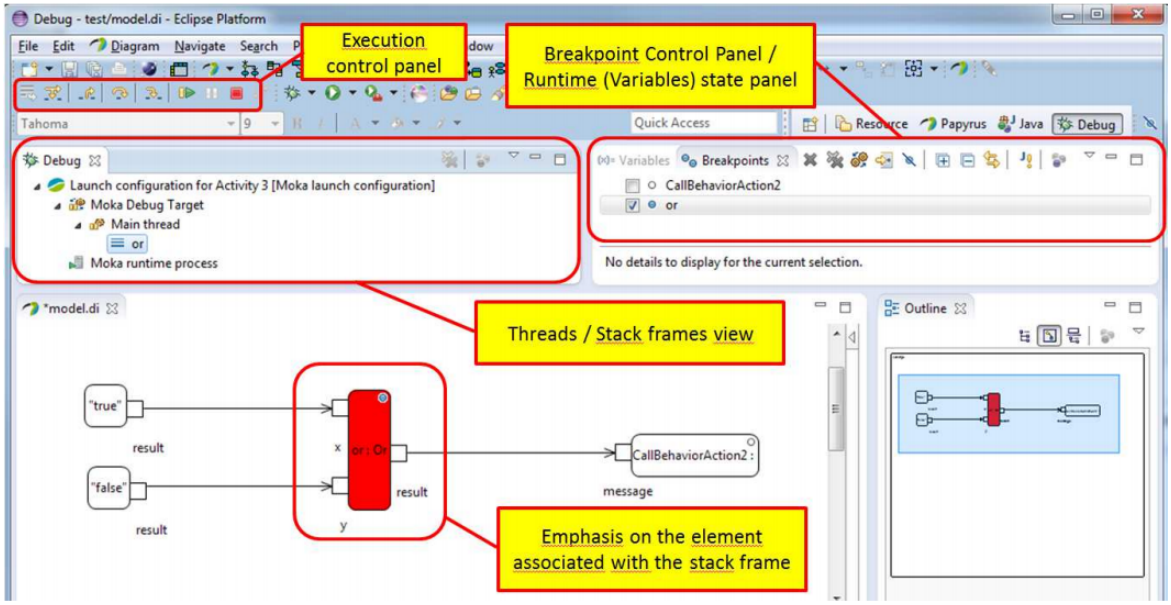

Figure 3.5.: A screenshot of Eclipse Moka taken from [Guermazi, Tatibouet, Cuccuru, Seidewitz, et al., 2015, p.10]

highlighted that prior approaches generated standard Java code to execute storyboards. This thesis approach is leveraged from MDSD and therefor generates an UML activity diagram which is subject to execution. Therefore, beside graph search implementations, there also have been introduced frameworks for model execution. The reason to exclude petri nets despite their formal foundation has been laid out.

In the previous subsections technology for graph search and model execution has been introduced. It was shown how the graph search implementations may be used, and how they may be adapted by this thesis. The outcome was that both, Guery and Neo4j require substantial work to be used within the thesis developed system. Furthermore, this work has to be repeated for every role-runtime and self-adaptive system that is used. Therefore, the interfaces provided by Role-based Software Infrastructures CROM and CROI have been investigated. Thus, a query and adaption interface approach has been derived which is easy to adapt, too.

As storyboards already have borrowed their semantics from UML activity diagrams, and since the thesis emphasize the MDSD approach it has been chosen to generate UML activity diagram instances from storyboard instances to be executed. Therefore, the fUML standard is chosen, as it delivers a semantic standard for UML activity diagrams. However, the standard does not allow or provide important aspects required in the system developed in this thesis. This encompasses guards at transitions and activities that execute code. Since there was no alternative for the diagram execution there have to be build an own implementation to run an adapted subset of UML activity diagrams. It must fulfill the needs expressed in chapter 3 on page 25.

4. CONCEPT FOR A ROLE-PLAYING AUTOMATON FOR SELF-ADAPTIVE SYSTEMS

In the previous chapters the backgrounds for this chapter have been laid out. Chapter 3 has shown that current concepts for separation of concerns regarding role adaption and business logic are not applicable in the context of self-adaptive systems, because the possibility to state context dependent role adaption is not possible. Therefore, this chapter introduces the concept of *Context-AwarE StoryboArds with Roles* (CAESAR). The original storyboards as well as their extension with roles have been introduced. However, as shown in chapter 3 they are still not sufficient to be used with SAS. Therefor, the changes made to existing storyboard concepts leading to the concept of context-aware storyboards with roles will be introduced in this chapter. Thereafter, the syntax and semantics of CAESAR are explained and presented in detail. Then the meta-model is shown and explained. Forelast, the reason to choose activity diagrams as basis for the deployed control flow elements is explained. At last a summary is drawn.

4.1. CONTEXT-AWARE STORYBOARDS WITH ROLES

Storyboards are used to describe role binding strategies. Like storyboards in film industry and user interface design they graphically describe situations [Luyten et al., 2010]. Storyboards are proposed by Fischer et al. to seamless integrate graph grammar languages with object-oriented design and implementation languages like UML and Java [T. Fischer et al., 2000]. Therefore, UML class diagrams are used to adopt the graph schemes, UML activity diagrams for the control structures and UML collaboration diagrams for the graph transformation rules (see section 2.2.3 for a more in detail description of graph transformation in storyboards). Thomas Kühn based his storyboards with roles [Kühn, 2011] on Fujaba's storyboards (cf. [Ira Diethelm et al., 2003]) which semantically represent graph transformation rules. These graph transformations can be applied on object-oriented programming languages, because of the object-graph-isomorphism. The storyboard approach is able to define complex match and replacement scenarios in a concise graphical way.

Context-aware storyboards with roles incorporate the storyboard with roles approach suggested by Thomas Kühn [Kühn, 2011], which applies graph transformations on the role-play graph. It extends the concept with context which consists of *context events* representing an incident in the real world, or in the system itself (e.g., w.r.t. the MAPE-K loop, an event fired in the analyze phase). Since the overall concept of storyboards it

still applied it uses control flow elements borrowed from UML activity diagrams (e.g., merge, fork, join nodes, and transitions with guards and events). This means, that context can be formulated as context events at transitions. Constraints on these context events will be possible using guards at the transitions. Role adaption is formulated using graph transformation in the reduced, compact syntax Thomas Kühn has suggested [Kühn, 2011]. Thus, situations described do not operate directly on the object graph but role-play graph. New to the concept is also the application of the new understanding of roles, which has evolved in the last years with the addition of compartments (cf. section 2.1.1). Thus, with the definition to add or remove roles from a natural in story patterns and the deployed control flow between the story patterns it allows to describe complex sequences of situations (the story patterns) and event occurrences which employ a meaning of context.

Transitions may be coupled with events, which will make the transition passable when the event occurred. Transitions may be further constrained by a guard, which is able to call operations on the context event. When the given event occurs, the runtime will check if the guard is satisfied. Otherwise, the transition is not fired, and the token at this transition will not move forward and get removed. For example, a transition could require the `entersArea` event, which employs the meaning of a person entering the working area of a robot. A token reaching the transition will be stalled until the event occurs. Thereafter, the transitions is fired. The guard for example could restrict the distance of the person entering the working area further `entersArea [enteringEntity().distance() < 100]` meaning that the transition can just be passed if the entity (i.e. person) is nearer than 100cm.

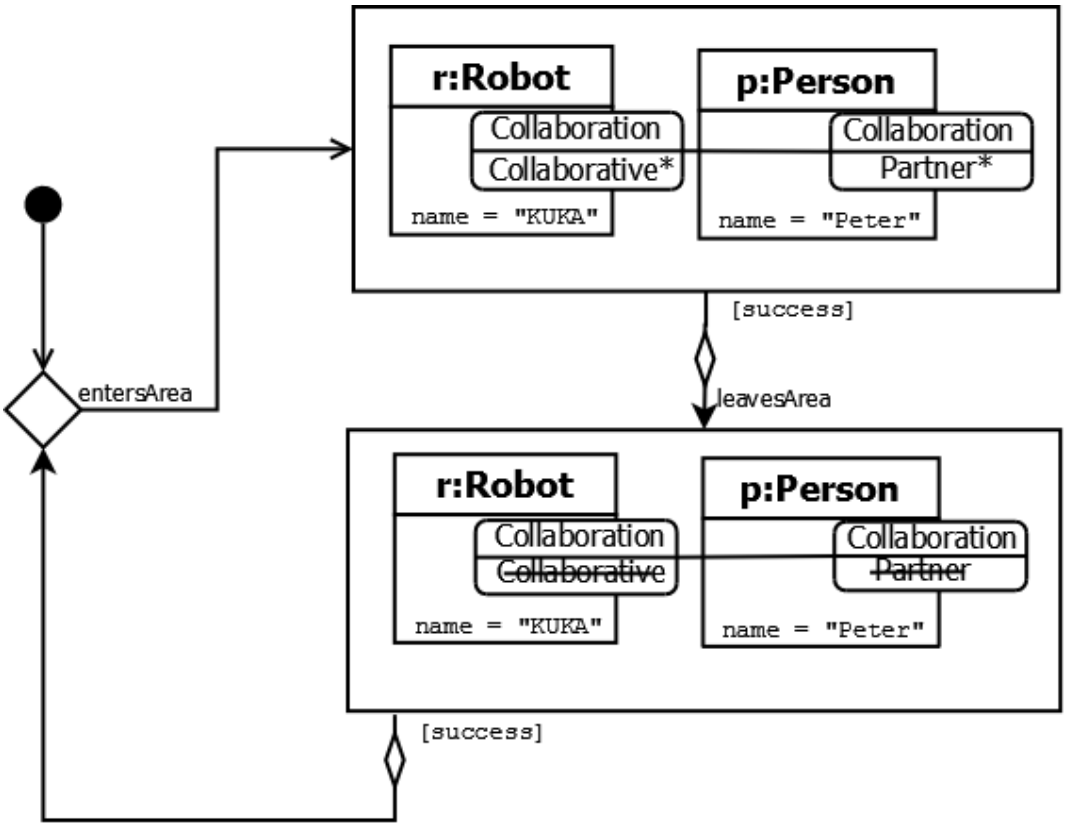

Figure 4.1.: A robotic co-working scenario modeled with CAESAR. A robot is taking part in a collaboration with a person.

So, context-awareness is achieved with two steps. First, you can match scenarios based on role bindings and collaborations of naturals and roles in compartments. Second, you

can connect those scenarios described in story patterns with a control flow which can be enriched with constraints and bound to events to represent incidents in the real world or the system.

4.2. SYNTAX AND SEMANTICS

In this section an overview about the syntax and semantics of context-aware storyboards with roles is presented. Thereafter, the syntax and semantics is presented in detail for the elements the storyboard is consisting of. Then the meta-model is shown. The similar aspects as well as differences between the CAESAR and storyboards, and between CAESAR and UML activity diagrams is exposed. At last, a summary is drawn.

4.2.1. OVERVIEW

In the big picture context-aware storyboards with roles consists of *story patterns* connected by a *control flow* (cf. figure 4.1 for an example storyboard). Following the control flow implies under what circumstances we came to a specific scenario. A story pattern describes a situation at a point in time. A situation is characterized by naturals collaborating over their roles in compartments. The story patterns provide a boundary for the actual elements describing the situation. The story pattern is the main element of the storyboard. It consist of role nodes which define graph transformation rules (i.e., queries on the role-play graph and role adaptions). Furthermore, it is stated how the roles have to be reconfigured. The matching of the story pattern can either succeed or fail. If it succeeds (roles are removed and added as described in the role bindings) the `success` transition is fired. If there was no match the `failure` transition is fired. These are different outgoing pins from the story pattern which can be connected with control flow elements to react accordingly.

Naturals are identified by the given class name. The role and compartment in the role node are identified by their role type name and compartment type name in the role model. All types are identified by their name. The role node is a rounded rectangle where the role's name and its compartment are stated. Story patterns are used to group role bindings into one single scenario.

Figure figure 4.4 on page 40 shows different control nodes which are used to control the flow between the connected story patterns. The current state of the storyboard is determined by the location of the tokens in the net. A token at a specific element means that the element is about to get executed. The start node defines the entry point into the control flow as which the end node defines the ending of the execution. However, the sink node just consumes the token without ending execution if there are still tokens alive. The storyboard allows to model parallelism using fork and join node. The fork node forks the incoming token into n tokens if there are n outgoing transitions. The join node joins n tokens from n different incoming transitions into one token (subsequent tokens from the same transition are either ignored or queued for later).

4.2.2. STORY PATTERN

A story pattern consists of naturals which are enhanced with role nodes at the right side of the natural type (short for `RoleModelNode`) (cf. (a) of figure 4.2 showing a single natural with two role nodes) . There is no restriction on the amount and types of role nodes. Those role nodes are describing the graph transformation rule, where each belongs differently to the LHS and RHS of the transformation rule. Hence, these role nodes describe queries and role adaptions. Figure 4.2 (b - e) shows the four different possible role nodes that can be combined. The role node (b) is the *play node* which means that the role has to be played

in the compartment, and just represents a query on the role-play graph. The role node (c), the *add role* node, is altering the role-play graph as it requires the specified role not to be already played in the compartment. After application, the role will be played by the natural in the compartment. The role node (d) is called *remove role* node and requires that a role to be removed is already played by the natural in the compartment. Last is (e) the *forbidden role* node, which is a negative query. Thus, if the natural is playing a role which is forbidden there will be no match.

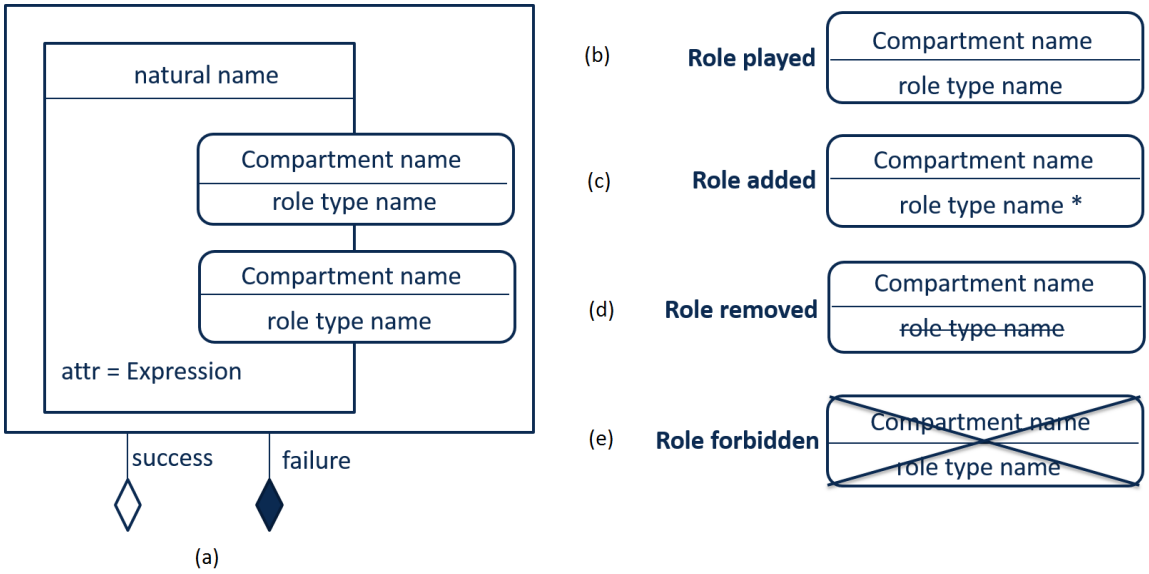

Figure 4.2.: Example of (a) a story pattern with querying role nodes, and (b - e) possible role node statements

Role types, compartment types and the natural are all concepts defined in the role-class model (i.e., the merged class and role model). The meta-model for the role types, compartment types and naturals is the Compartment Role Object Model. Because CROM is a meta-model family an instance of CROM is used (but will be referenced as CROM). The current instance of CROM disallows deep roles (which means that roles are playing roles) because every role is always assigned to a natural. Furthermore, one player could just play one instance of a role in a compartment. Otherwise, if playing multiple instances of the same role type in a compartment would be allowed, it could not be stated which of both need to be queried and modified. Even when constraints on role types are added (as Thomas Kühn suggested for storyboards with roles) both could remain indistinguishable.

In the following the single parts a role node consists of will be introduced in detail. The different parts and their interaction in the whole context are depicted in the meta-model figure Appendix A.2.

ROLE MODEL NODE

A role model node consists of many role model elements, which are the role nodes describing the graph transformation rule and one natural node representing the natural that is playing or will play the stated roles. It has a name which is the class name of the natural represented. Figure 4.3 shows an excerpt of the meta-model regarding the role nodes.

Natural Node The natural node is an abstraction representing the natural that is subject of transformation.

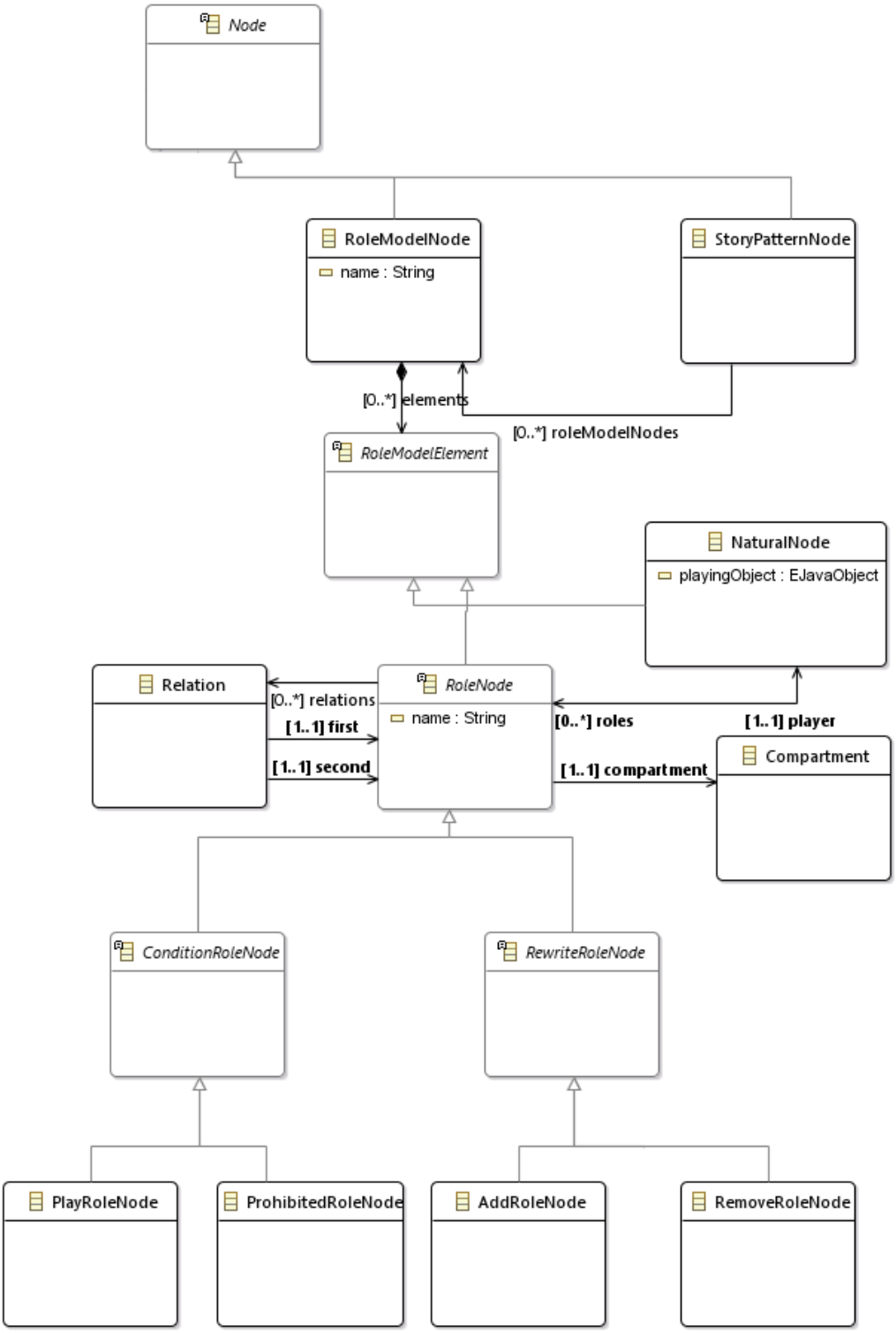

Figure 4.3.: The meta-model for story patterns of CAESAR

Role Node A role node can either be a condition role node, which means it is a structural invariant query on the role-play graph. Or it is a rewrite role node which means it is a role node defining a graph transformation rule. Roles that are played by the matched natural which are not part of the transformation rule are seen as *don't care* roles.

Condition Role Node A condition role node is an abstract type representing a role node which is just a query on the role-play graph without altering the structure. Therefore, there is just the play role node and the prohibit role node.

Play Role Node The play role node means that the natural has to play an instance of the specified role type in the compartment. If the specified role is not played by the natural in question the matching will fail.

Prohibit Role Node The prohibition role node means that an instance of the role type must not be played in an instance of the specified compartment type. If the role is played by the natural in question the matching will fail.

Rewrite Role Node A rewrite role node is a node which is altering the role-play graph if a natural can be matched. Thus, they are adding or removing roles from the natural in question.

Add Role Node The add role node is a composed role node. It requires that the role must not be played already. If the an instance of the role type is already played in an instance of the compartment type than the matching will fail and the role will not be added.

Remove Role Node The remove role node is a composed role node. It requires the role must be played and removes the play relationship between the role and the compartment. If the role is not played before the natural is not matched.

4.2.3. TRANSITIONS, EVENTS, AND GUARDS

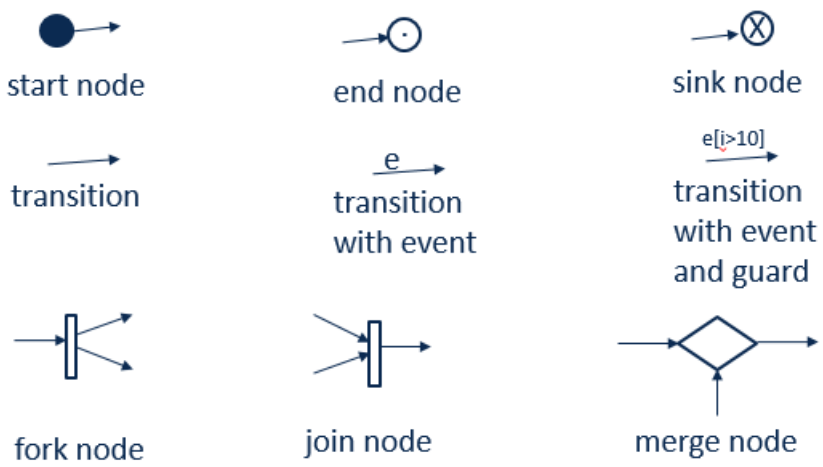

Figure 4.4.: Overview of the control flow elements of CAESAR

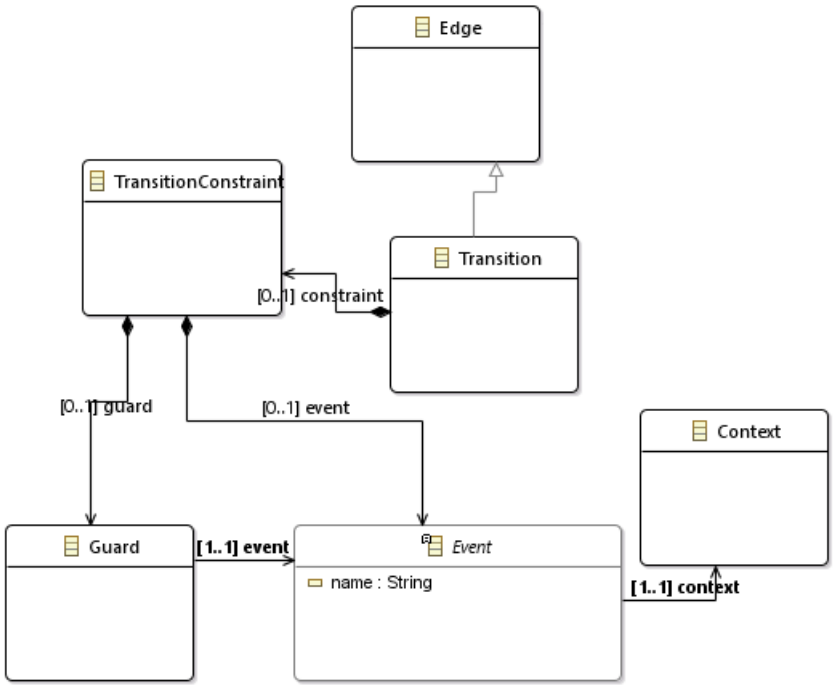

Figure 4.5.: The meta-model of the context-dependent part of CAESAR

Transitions are defined as edges that connect nodes. These nodes are either story patterns or control nodes. Events and guards are directly connected to their transitions. In the following the relation of these three is presented. Figure 4.5 shows an excerpt of the meta-model w.r.t. transitions, events and guards.

Token The token is part of the execution of the storyboard. A token represents current execution state. The set of all tokens represents the current active configuration of the storyboard. Beginning with the start node the token moves through the storyboard and activates activities and transitions it visits. A token gets stalled when the visited transition requires an event to be fired which not happened, yet. The token will be reactivated when the event occurs. Tokens can get removed from the execution when a guard is not satisfied (i.e., results to false), or when a sink node is visited, or when an end node is visited.

Transition A transition is a directed connection between two nodes. A node could be a story pattern or a control node (e.g., a merge node). It has one source and one target node. The source node is restricted to control nodes (which includes success nodes and failure nodes as outgoing pins from story patterns). As target node control nodes and story patterns are allowed. A transition may have an event associated with it. A guard which further employs constraints whether the transition can be passed is optional but requires an event.

Event The event is the abstraction of some incident in the real world or system. The actual event instance will be supplied by the external system using the storyboard. An event provides access to the context. Thus, it is also called the *context event*. The name of the event stated at the transition is a type reference to an event type. The event is responsible to carry the *context object* that is meant to be connected to the event as its trigger. A guard

can further state restriction on the context object. As shown in figure 4.4 on page 40 an event is directly connected to the transition stated with its name.

Guard The guard can constrain the transition. It accesses context to query if the stated condition is true. The condition has to be functions that can be called on the context object and the statement need to result in a boolean value (e.g., `[workItem.weight > 50]`). As depicted in figure 4.4 on page 40 guards are stated in square brackets directly attached to the transition next to the event.

4.2.4. CONTROL NODES

Control nodes employ the control flow between story patterns. Connected by transitions they allow to directly influence the execution of the storyboard. The control nodes comprise the control flow elements and are borrowed from UML activity diagram. They define semantics in order to control the forwarding of tokens. They have the ability to either delete a token from the storyboard, merge tokens or to fork a token into multiple tokens. Figure 4.4 on page 40 shows the syntax of those control nodes which are described in the following.

Fork Node The fork node functions as a token multiplicator introducing the possibility to model parallelism. It has one incoming transition and multiple outgoing transitions. The incoming token is cloned and fired on all outgoing transitions which may be executed in parallel.

Join Node The join node functions as a synchronization barrier. All incoming transitions have to be fired in order to fire the outgoing transitions. Therefore, the incoming tokens are merged into one token and the resulting token is emitted. The standard join node works as an `AND` between all incoming tokens. There could be an `OR` node which defines which incoming transitions are optional and which are mandatory. This is shown in figure 4.6 (a), while (b) shows an alternative modeling to achieve this using a merge node to join both incoming transitions unconditionally.

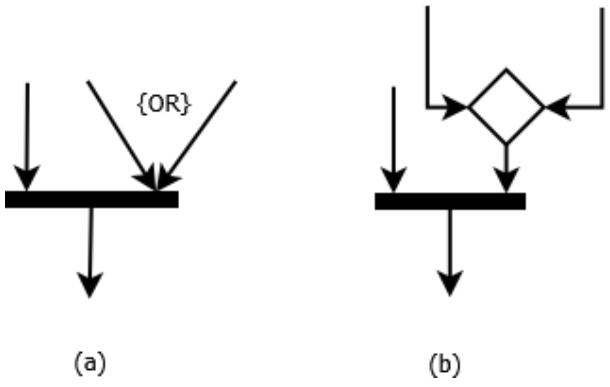

Figure 4.6.: An OR-join in (a) defines that the two right incoming transitions either can fire one or another. In (b) an equivalent alternative modeling approach is shown using a merge node.

Merge Node The merge node has multiple incoming transitions and one outgoing transition. If there is an incoming token it will get forwarded onto the outgoing transition. It can be used to unconditionally merge control flows (e.g., to model a loop as shown in figure 4.1).

4.2.5. VARIABLE BINDING

The idea behind variable binding has been stated in section 2.2.3. At the beginning of execution of an instance of a story pattern all referenced elements (e.g., naturals, roles, compartments) are not resolved. In storyboards [T. Fischer et al., 2000; Kühn, 2011] minimum one variable need to be bound at execution begin (e.g., as a method parameter). During execution a stack of bound variables will be build where following story patterns and guards can reference previously bound variables (e.g., the guard `r1.name = "KUKA"` would require that `r1` is already bound and will reference a `Robot` type).

However, CAESAR models a whole system and therefore does not allow to state parameters at execution begin. In CAESAR, there is no prior variable binding required as all patterns will be matched at runtime. Because of the parallelism in CAESAR, and the fact that there is no control over the instances of the role-play graph binding of variable references throughout the execution is not possible as practiced with storyboards. For example in figure 4.1, between the occurrence of the context events *entersArea* and *leavesArea* there could pass lots of time. Because storyboards do not require to model every possible aspect of the system just open-world assumptions [Keet, 2013] can be stated. This means, that in the mean time a mechanic could deactivate the robot which would make the second story pattern in figure 4.1 obsolete. Using the possibility to store resolved references would therefore result in faults (i.e., changing a robot which is not there anymore). Hence, story patterns have to be executed everytime again to check if they are applicable.

4.3. META-MODEL

In the previous sections of this chapter single parts of the meta-model have already been introduced. The whole meta-model is shown in Appendix A.2. Therefore, it has been pointed out that the meta-model is two-parted consisting of a part concerning story patterns, which is itself consisting of the graph transformation part and the role representing part, and a part expressing the control flow of storyboards which itself is divided into the control flow elements part borrowed from UML activity diagrams, and the part regarding context with transitions, events and context.

The context relevant part is shown in the left bottom quarter of figure A.2. A transition is constrained by an event and a guard. The event is the connection to the context, which is the reason it is also called *context event*. An event represents an incident in the real world or system and allows to be restricted by a guard. The guard formulates constraints over the event on the context. Thus, events should implement the methods called on them from the guard. Alternatively, the guard could directly state restrictions on the context without the need of an event. This would require the possibility to state assumptions over the context and is part of future work. Piechnick has introduced a context model query language (GRoCoMo-QL) [Piechnick, Püschel, et al., 2014] for the context model of SMAGs [Piechnick, Richly, et al., 2012].

The role representing part is located in the middle comprising the `RoleNode`, the `Compartment`, and `NaturalNode`. The role node is an abstract representation of a role. The compartment is referenced from the role to emphasize that roles are played in a compartment. The natural node holds a list of all roles that are subject of the story pattern. In the other direction a role knows its player. This is enough to state the graph transformations which are just adding if a role is for example added or removed.

The graph transformation related part is in the bottom of the middle. It comprises the `ConditionRoleNode` and its inheriting classes, and the `RewriteRoleNode` and its inheriting classes (see figure A.2). As the graph transformation rule is directly stated at the `RoleModelNode` it is modeled as a subclass of `RoleModeleElement` aligned with all

other elements which are part of story patterns.

At last the right side of figure A.2 which consists of the `ControlNode` and all its subclasses which are borrowed from UML activity diagram.

4.4. DIFFERENCES TO RELATED CONCEPTS

The context-aware storyboards with roles originate from other concepts, such as story diagrams and storyboards with roles. Despite their similarity it still host some differences in the concept itself. These will be laid out in this section.

4.4.1. RELATION TO UML ACTIVITY DIAGRAMS

UML activity diagrams have been chosen because of the idea was to transform the storyboard into an activity diagram, and execute it with existing solutions. Prior approaches on storyboards generated standard Java code. But this thesis emphasizes the Model-Driven Software Development. Thus, storyboards should be executed by an automata. Petri nets have been a discussed alternative, but was abandoned because of the lack of execution frameworks (see section 3.3). However, after intensive research there was no execution software for activity diagrams which would allow all necessary features. Even the fUML standard disallowed necessary requirements, such as guards at transitions. The reasons are discussed in section 3.3.2 on page 32.

Storyboards borrowed their control flow from UML activity diagrams [T. Fischer et al., 2000]. Guards can be used to constrain transitions. Transitions can be triggered by events. The story patterns can be transferred into activities where the graph transformation is distributed into two parts – matching and transforming. The matching part is executed at first, and if a match was successful the transformation will be executed.

Despite the context-aware storyboards borrow concepts from the UML activity diagram the semantics differs. The utmost similarity is in the appearance of control flow elements in the graph and their semantics. However, not every control node has been implemented. For example, the *decision node* is not implemented, because there is no access to context at this point. Context-aware storyboards and activity diagrams both differ in their semantics of the tokens floating around the net. Activity diagrams know two types of tokens: (1) control token, (2) object token (see section 2.2.1 for detailed explanation on UML activity diagrams). Context-aware storyboards only know control tokens. Objects (e.g., context) is introduced by events (i.e., context events). Hence, guards can state constraints w.r.t these context events.

4.4.2. DIFFERENCES TO STORY DIAGRAMS

The concept of context-aware storyboards with roles originates and extend the concept of story diagrams (section 2.2.3), but does not implement everything proposed in the initial approach. A lot of these things are future work (see section ??). In storyboards the `for_each` transition defines semantics to execute the story pattern for every match. If the story pattern cannot be applied anymore because there is no pending match the control flow will move further. The concept of this thesis instead takes the approach, that the first match even if there are multiple instances of the pattern found will be executed on this single match. The match will be chosen undeterministic. However, the `for_each` can be simulated by a transition from the *success* node to the story pattern itself. Then it will be matched again until there is no match anymore and at last the *failure* transition is taken.

Storyboards are introduced because in UML activity diagrams activities lack formal execution semantics [T. Fischer et al., 2000; Zündorf, 2001]. Activities model functions (i.e.,

methods of programming languages). Thus, the concept has been used to model functions. The concept of this thesis allows to model an automaton which abstracts away from single functions. It models context-dependent system adaption. Therefore, context is introduced as events, and can be accessed through guards. Such events can be system events, or incidents in the real world. However, there is no restriction that these events should happen in a timely manner. Beside using UML activity diagrams, the original storyboards did not introduce means to model parallelism as it omits *fork* and *join* nodes in the control flow. Because the concept of the thesis is used to model context-dependent system adaption parallelism is introduced. Both, the abstraction to a higher level view of the system, as well as the introduction of parallelism lead to omitting variable binding. The concept developed in this thesis has no authority over the role-play graph. External modifications are hidden. The original concept of storyboards allowed variable binding everywhere. A bound variable means that a reference is matched to an instance in the object graph (see 2.2.3). Thus, already bound variables can be used in subsequent story patterns in graph transformations, or be used to formulate guards on them. The concept of this thesis does not allow that kind of variable binding. Variables can be bound within a story pattern, but are not referable from outside. Thus, there is no global reference cache like in storyboards. However, the original approach needed at least one bound variable at the beginning of execution. This is because a graph search can be NP-complete [Eppstein, 1995], and there is no external view on the object graph which can be queried in order to find specific instances (e.g., instances of a specific class). The concept of this thesis does not require any bound variable a priori. This is, because the constraints to instances of specific player classes already counts as a fixpoint. The concept can be used with any role-runtime. However, these runtimes always hold a list of the *play* relationship.

Story diagrams propose *statement nodes* that can formulate algorithms with expressions. Furthermore, *call acitvities* are implemented, which can call other activities. This allows for structuring. Both are not implemented in the concept of this thesis, and the latter are seen as future work. Statement nodes need to be further evaluated if they are applicable to the concept.

4.4.3. DIFFERENCES TO STORYBOARDS WITH ROLES

Thomas Kühn proposed the storyboard concept to be used for explicit role binding [Kühn, 2011]. Therefore, he extended the storyboard approach to state graph transformations on the role-play graph instead on the object graph (see section 2.2.3).

The storyboards were able to further restrict the matching part of classes and roles depending on their attributes. Thus, a class `Robot` could be restricted to a specific color e.g., `color == "blue"`. Furthermore, they allowed to define external constraints, such that `robot.hasMoved()` which is tagged to the specific class node in the story pattern. This was defined as internal and external constraints [Kühn, 2011, p.76f]. Both are not part of the concept of this thesis, but are future work.

Furthermore, the story pattern was able to state associations between roles. This will not be part of the context-aware storyboard with roles developed in this thesis, but can be part of further investigation on how to implement the full storyboard approach.

Role migration is not part of the concept because many runtime systems do not allow role migration (cf. [Herrmann, 2007]).

4.5. SUMMARY

Context-AwarE StoryboArds with Roles (CAESAR) extends the idea of storyboards with roles [Kühn, 2011] with the ability to address context. Today the role community understands that

roles are played in a role context which is called compartment. Thus, CAESAR extends the role binding mechanisms of storyboards with roles with compartments. The story patterns allow to define role reconfigurations based on graph transformations. Context is introduced by allowing to add *context events* to transitions, which now require an event to be fired in order to be activated. These events are called context events, because they represent an incident in the real world (e.g., a sensor measures that a person is near the robot) or system events (e.g., a symbol has been clicked in an application). Following transitions employs some kind of context, too. Guards can be formulated to further restrict the forwarding through the transition. The guards can call functions on the events. The control flow elements used are borrowed from UML activity diagrams. Beside storyboards proposed by Fischer [T. Fischer et al., 2000] CAESAR allows to model parallelism using fork and join. However, CAESAR does not model methods but systems. This is a huge difference between storyboards and CAESAR. The result is that the variable binding as known in storyboards is not possible. Due to the fact that CAESAR has no control over the role-play graph and using open-world assumptions [Keet, 2013] it could not enforce references still to be valid.

As already said it allows to model systems instead of methods. Thus, the high level view of role reconfigurations allows it to model adaptions in a concise way. Furthermore, stating context as events and constraints (i.e. guards) and role reconfiguration in story patterns allow for separation of concern. The simple abstraction used to state role reconfigurations on the role-play graph allows to adapt role programming languages that define these concepts (e.g., roles, compartments and naturals or role-playing objects) themselves.

5. IMPLEMENTATION

This chapter presents the architecture of the developed context-aware storyboards with roles. It will outline how the requirements formulated in chapter 3 are fulfilled and how the concept shown in chapter 4 has been implemented. At first an overview about the system and its interfaces is given. Then the different aspects of the system are discussed in detail and their implementation is explained. After this the limitations with the current implementation is highlighted and last a summary is drawn.

5.1. ARCHITECTURE

This section introduces the architecture and design principles of CAESAR. Like already mentioned in chapter 4 the system developed is two parted. That means there is a tooling part to model the storyboards, and a runtime part to execute the storyboards. In figure 5.1 the top-level architecture of CAESAR runtime is shown. Therefor, it gives an overview about the environment of CAESAR and how CAESAR fits into the big picture. To use CAESAR three interfaces need to be implemented which are offered by the facade [Gamma et al., 1995]. Namely, the `IQueryModelInstance`, `IQueryModel`, and `IAdaptModelInstance`. Thereby, *model instance* stands for the CROM instance level, M0 (cf. meta-model hierarchy) and *model* for M1, respectively. The query interfaces offer functionality to query the role-play graph on both meta-levels M0 and M1. That interfaces have to be adapted by the system which wants to use the storyboard runtime. As figure 5.1 implies the adapters have to be implemented for every role-based programming language. Because they all maintain the role-play graph somehow the implementation effort for the adapters is limited. However, it offers architectural freedom, because the implementer can choose how to adapt the queries (e.g., a role database can adapt the query interface and map the Application Programming Interface (API) calls to actual database queries).

Figure 5.2 on the next page shows the dependencies of the implemented components where the arrow means that a component uses another. Beginning with the modeling part at the right, a DSL has been developed that is able to represent instances of the meta-model shown in the figure in Appendix A.2. Using a code generator an instance of the runtime model is generated which is subject of execution of the interpreter. The manager component consists of the `ExecutionManager`, `EventManager`, and `TokenManager`. The `ExecutionManager` is responsible for initialization of the other manager classes and controls the execution of the storyboard. The `TokenManager` is responsible to store tokens and their current state (e.g., their execution state in the storyboard). The `EventManager` is responsible to receive and distribute events to waiting tokens.

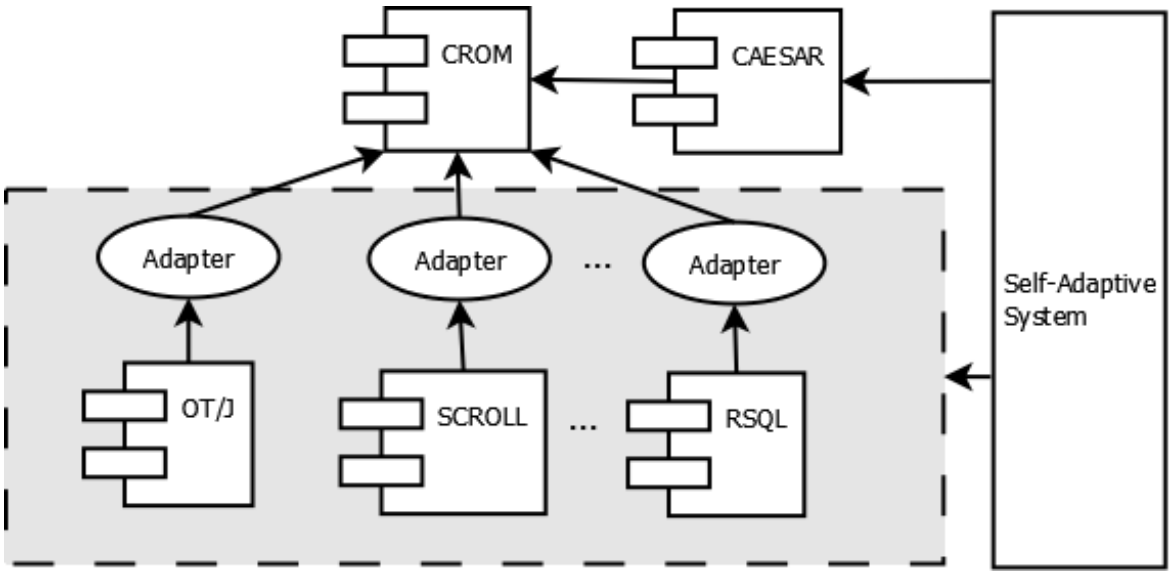

Figure 5.1.: Overview over the top-level architecture and its interfaces.

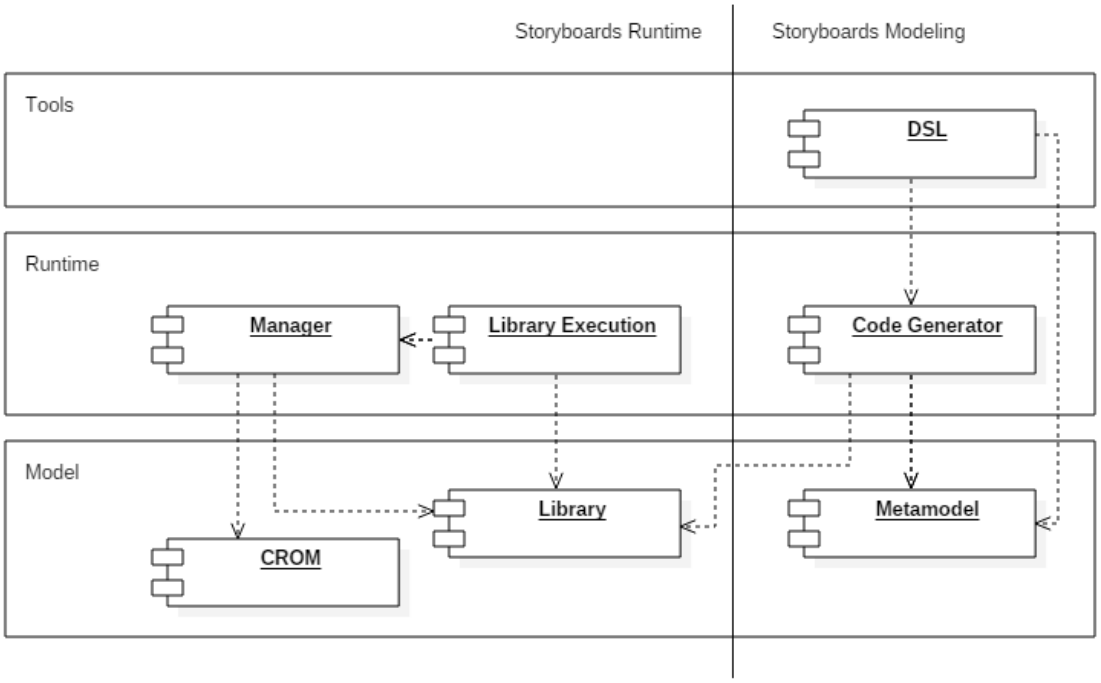

Figure 5.2.: This figure shows the dependency structure of the packages of the context-aware storyboard. They are labeled into three tiers for tooling, runtime, and the model.

5.2. IMPLEMENTATION

For the implementation Eclipse has been used as a framework. Thus, all components are implemented as Eclipse plugins which improves modularization, and encapsulation of internal parts of the implementation. For the models the Eclipse Modeling Framework (EMF) has been used. After the top-level architecture has been introduced this section will discuss the different components that compose the system, and explain their implementation in detail. At first the DSL is introduced, then how the storyboard instance is generated from it using a code generator. Next the pattern matching and role adaption implementation is explained. Afterward, the role model abstraction for the storyboard is shown. After this the implementation of the event architecture is explained and how it can be adapted by external systems. Then the model execution is discussed and last it is explained how the storyboard can be validated w.r.t. the role model.

5.2.1. GRAMMAR AND META-MODEL

Part of the motivation of the thesis was that textual role binding (e.g., role programming languages, see section 2.1.3) is not concise enough. For this reason Thomas Kühn has investigated and suggested storyboards with roles to model role adaption with storyboards [Kühn, 2011]. Therefore, in a first step textual modeling of CAESAR has been developed where a graphical modeling tool can be build on top of.

The meta-model (cf. Appendix A.2) presented in chapter 4 has been implemented as a grammar using Xtext[1] and Xbase[2]. Xtext is a parser generator framework which allows to define an attribute-grammar from which a meta-model, and a parser is generated. Xbase allows to integrate Java Virtual Machine (JVM) expressions (e.g., Java code) directly into the model. Thus, Xbase allows the external DSL to still be used beside existing implementations on the JVM. For example, classes can be referenced and used in the model, while Xbase delivers type inference and code completion.

The meta-model shown in the concept chapter was to show the structure and relations that make up a context-aware storyboard with roles. To implement the DSL it has been used as an inspiring example. Attributes and implementation specific stuff has been left out (e.g., the generated model includes Xbase specific model elements like `ImportStatement`). Appendix A.1 shows the grammar. Thus, storyboards can be formulated in a DSL which is then transformed into a runtime model inspired from UML activity diagram.

Listing 5.1 shows an excerpt from the use case. In order to see the complete story board see listing 7.1. The scenario is a robot which is working autonomously. A person entering the working area of the robot is sensed and the system adapts the robot to collaborate with the person. Listing 5.1 shows two story patterns which are connected by a transition. The transition fires when the `entersArea` event is issued. The first story pattern just asks for a robot natural and adds the `Autonomous` role. That means that the external system (e.g., self-adaptive system) will make the matched robot play this role. As told in 2.1.1 roles add or overwrite behavior. The second story pattern is matching a robot natural which is working autonomously (i.e. playing the role `Autonomous`) and remove this role. Furthermore, the `Collaborative` role is added to the natural. This role will overwrite the behavior how the robot is collaborating with the person in perception. Thus, the robot will hand out workpieces at a place where the person can take it instead of dropping work items into a bin. When the person is leaving the robots work area the `leavesArea` event is fired and the robot will continue autonomously. As you can see in listing 5.1 line 8 the source node of the transition uses `CASB`RobotIsAutonomous.success. That means, that the transition will be fired when

[1] Xtext: https://eclipse.org/Xtext/
[2] XBase: https://www.eclipse.org/Xtext/documentation/305_xbase.html

```
storypattern RobotIsAutonomous {
  class de.larsschuetze.usecase.robot.Robot {
    add Autonomous in Compartment
  }
}
transition PersonEntersArea {
  { entersArea }
  RobotIsAutonomous.success -> RobotIsCollaborating
}
storypattern RobotIsCollaborating {
  class de.larsschuetze.usecase.robot.Robot {
    remove Autonomous in Compartment
    add Collaborative in Compartment
  }
}
```

Listing 5.1: An excerpt from the use case: robotic co-working. A robot is being adapted to collaborate when a person enters the working area.

the pattern was found and the adaption could be applied. If there is a failure every story pattern has a `failure` node that could be used as a start node for transitions.

The entities (e.g., natural, role, compartment) that should be matched has to be named fully qualified in order to be be identifiable. That is because the type system is string based. Ehrlich proposed a type system for roles [Ehrlich, 2016].

5.2.2. MODEL TRANSFORMATION

A model transformation is the transformation from a source model into a target model [Mens et al., 2006]. In this case the modeling time model is transformed into an instance of the runtime model using Xtend[3]. The runtime model can be executed by an interpreter. The difference between both models is mainly with the story patterns. Everything else is almost the same in both models. A story pattern is transformed into an activity, but the query code is integrated and generated in this step. This means, that the code generator generated the code which is called against the query interfaces. As figure 5.3 shows the `RoleNode` and all its sub-classes as known in the model-time meta-model is not existent anymore. They are transformed into a `MatchActivity` which is despite its name responsible for matching and emitting of the adaption plan. Using an activity is because of the idea to use UML activity diagrams. However, to really use activity diagrams the querying and the adaption part need to be modeled in two activities. Figure 5.4 shows an example result of a transformation into an UML activity diagram. Furthermore, the object flow needs to be implemented between both activities. The CAESAR runtime model does not allow object flows.

Using a model at runtime (i.e. Model@Run.time) approach allows to improve the runtime e.g., with a Just-In-Time (JIT)-like compilation technique combined with interpretation.

[3]Xtend: http://www.eclipse.org/xtend/

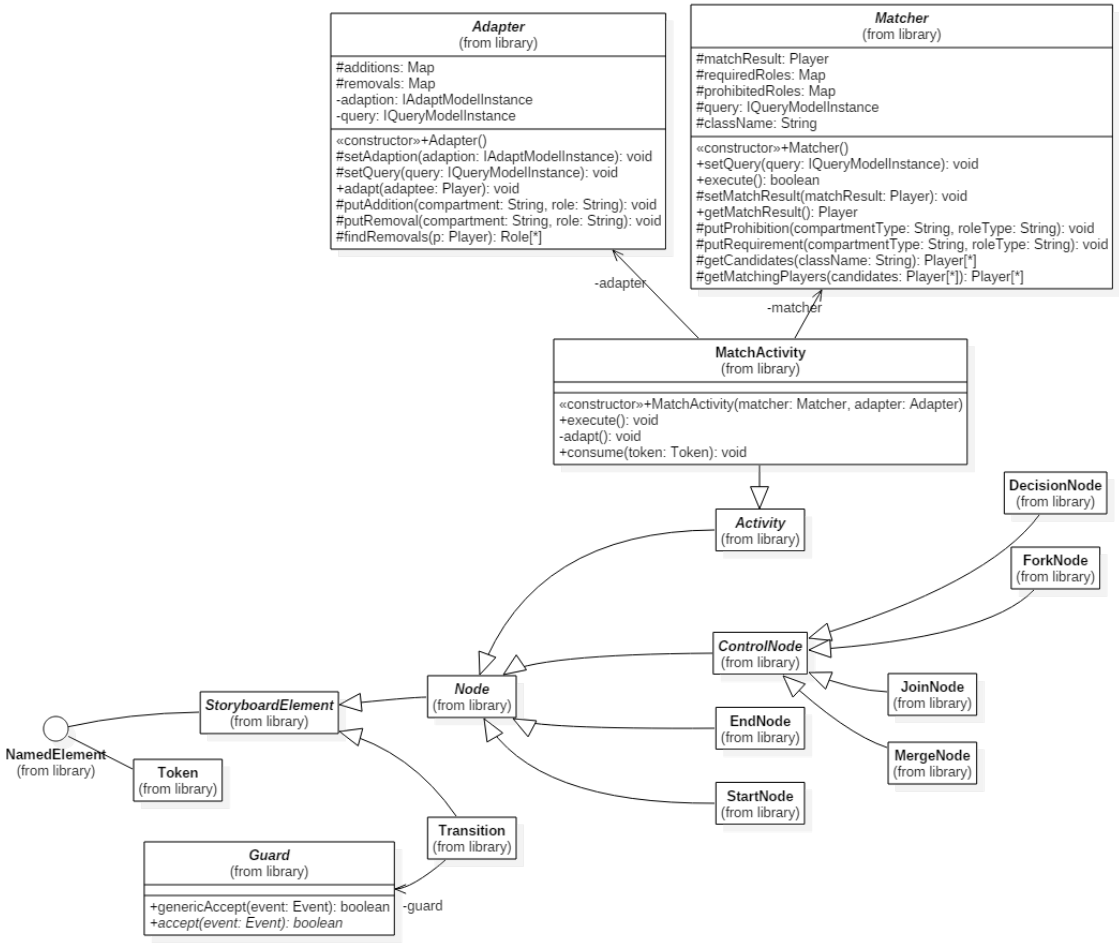

Figure 5.3.: This is the runtime meta-model showing the library part of CAESAR. Important classes are shown with more detail.

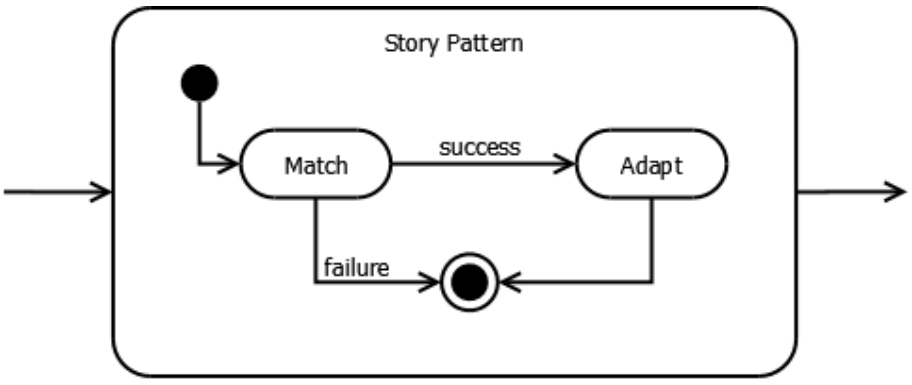

Figure 5.4.: This figure shows a possible result of a model transformation from a story pattern to an UML activity diagram. The story pattern will result in two activities, where one represents the matching part, and the other the adapting part.

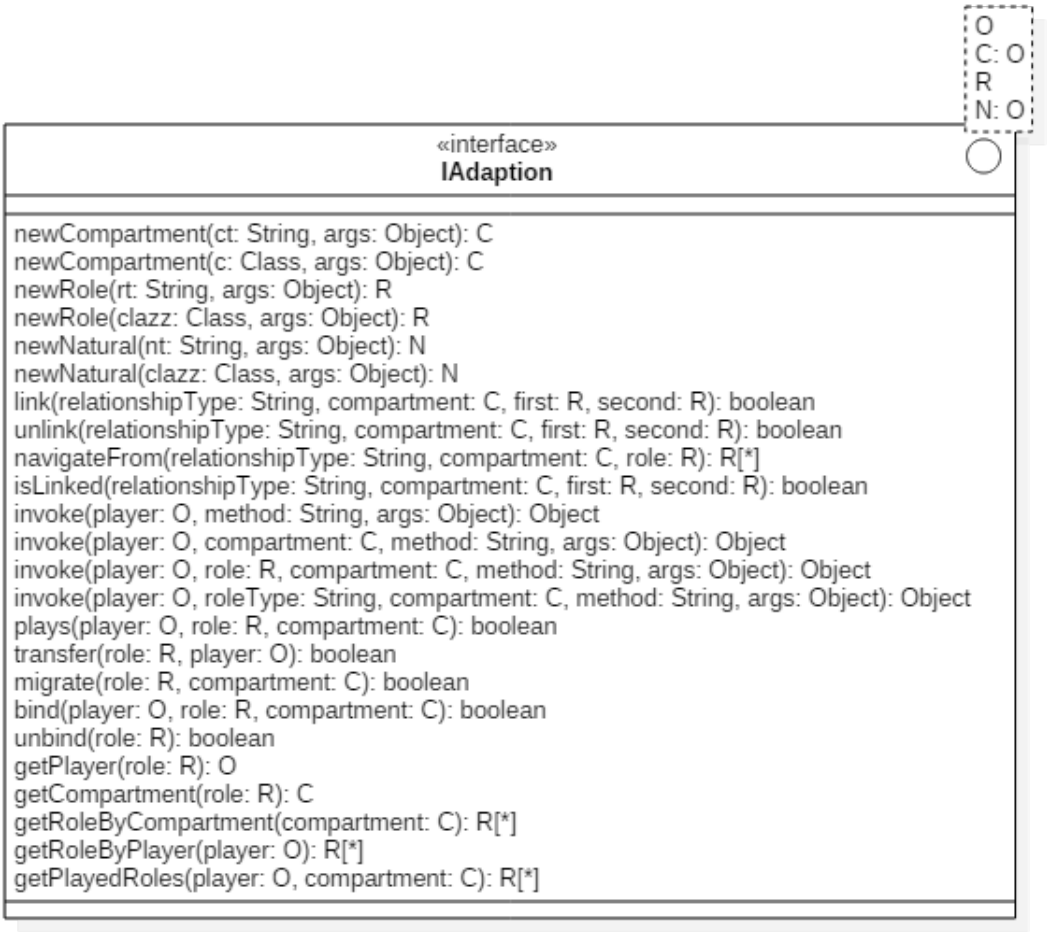

Figure 5.5.: Compartment Role Object Instances offers the `IAdaption` interface as part of the RoSI reference architecture. It allows to query and adapt the role instances.

5.2.3. GRAPH TRANSFORMATION

The graph transformation in CAESAR can be reduced to pattern matching and role adaption. The pattern matching and role adaption is part of the library component of the runtime (cf. figure 5.2 and figure 5.3). As shown in the previous section the story pattern are translated into their own instance of a `Activity` class which is responsible to find the formulated pattern in the role-play graph using `Matcher`, and an `Adapter` class which is responsible to emit the adaptation commands formulated in the story pattern. The model-time meta-model comprises `ConditionRoleNode`, and `RewriteRoleNode` where the first just is a query on the role-play graph. Thus, it does not contribute to the adapter. The second one contributes to the matcher and the adapter. For example, the removal of a role from a natural requires the natural to play the role beforehand. Thereafter, the adapter will issue a command to remove said role from the natural. These adaption commands are translated to API calls against the `IQueryModelInstance` and `IAdaptModelInstance` interfaces which are inspired by the `IAdaption` (shown in figure 5.5) interface of the RoSI reference architecture. IAdaption has been splitted because it does not separate concerns. It mixes query and adaption concerns (and execution concerns as it allows to execute methods on role instances). An alternative implementation approach is discussed in the following section.

ALTERNATIVE IMPLEMENTATION

An alternative implementation for the pattern matching would be to implement some internal understanding of roles, compartments and relations with the degree of detail needed to search instances of story pattern motifs in the role-play model. Then the user of the storyboard has to adapt the external role-play model to the internal role-play model, and all reasoning is done on the internal role-play model. The external role-play model could be informed about changes by using an observer [Gamma et al., 1995]. The pattern matching would be done with the Guery library. Therefore, an adapter need to be implemented to adapt the internal role-play model with the JUNG graph API to allow to query the role-play graph. Guery uses the MVEL expression language[4] to implement the query language (cf. section 3.3.1). The queries on the role-play graph would have been generated by Xtend from the described structure of the story patterns. The downside would be that there are two representations of the role-play graph and the user has to adapt his role model to the internal one. On the other side the user could implement the JUNG adapter directly on the external role-play model. But this requires deep understanding of internal technology.

5.2.4. THE ROLE MODEL

CAESAR uses an instance of the Compartment Role Object Model meta-model family. There have been involved some decisions to choose a not too restrictive meta-model instance which are shown in table 5.1. To generate an instance of CROM for L1 you need to answer the 26 questions. The meta-model defines what role models are allowed, or are able to be adapted. CAESAR itself just needs three parts of a role meta-model: naturals, compartments and roles. In the future relations could be added. However, *deep roles* which means roles are playing roles, and compartments playing roles is not allowed. This would mean, that the story pattern is not matching a natural which is playing specific roles and is subject to adaption, but a role. Furthermore, playing a role more than one time in the same compartment is not allowed either. Imagine an adaption requiring a role and removing the same role, too (which implies that the role is played, too). This would result in the role adaption being undeterministic. Same applies with a role which is part of many compartments.

5.2.5. CONTEXT AND EVENTS

The DSL allows to state events that are required in order to activate a transition. These events consists of a short name and their full qualified name which needs to be registered in the beginning of the storyboard DSL. Therefore, the `events` environment allows to import event types e.g., `import QualifiedName as EventName` will import the event which can be referenced in the storyboard. At runtime the `ContextEvents` interface allows to register instances of events to the system and to let the system know when an event happened. Therefore, the runtime offers an `Event` class which has to be adapted. The runtime events must be assignable to a class with the given qualified name. Figure 5.7 shows what happens in the system when an event is issued from an external system in an UML sequence diagram. At first stalled tokens that are waiting for the event are resumed. Otherwise, the event is added to the list of events that already happened (e.g., for events which are valid for a certain time). The `Event` class is a container which holds the actual *context object*. The concept already explained that guards are executed on the context object. These objects need to define methods which are used in guards. Figure 5.6 shows a class diagram depicting this. Aggregation is used over inheritance so that the context objects do not have to inherit storyboard specific classes or interfaces. Xbase allows to use

[4] MVEL: https://en.wikisource.org/wiki/MVEL_Language_Guide

Roles have properties and behaviors	●	Roles depend on relationships	○
Objects may play different roles simultaneously	●	Objects may play the same role (type) several times	○
Objects may acquire and abandon roles dynamically	●	The sequence of role acquisition and removal may be restricted	○
Unrelated objects can play the same role	●	Roles can play roles	○
Roles can be transferred between objects	○	The state of an object can be role-specific	●
Features of an object can be role-specific	●	Roles restrict access	○
Different roles may share structure and behavior	●	An object and its roles share identity	●
An object and its roles have different identities	○	Relationships between roles can be constrained	○
There may be constraints between relationships	○	Roles can be grouped and constrained together	○
Roles depend on compartments	○	Compartments have properties and behaviors	●
A role can be part of several compartments	○	Compartments may play roles like objects	○
Compartments may play roles which are part of themselves	○	Compartments can contain other compartments	○
Different compartments may share structure and behavior	●	Compartments have their own identity	●

Table 5.1.: Features that classify CAESAR's role meta-model, thus all role models that are adaptable. Extracted from [Kühn, 2011]

legend: ●- yes, ○- no

type checking at design time to check if the referenced class implements used methods. This is the reason guards can only be stated at transitions when an event is registered with this transition, too.

5.2.6. MODEL EXECUTION AND VALIDATION

The runtime model is subject of execution by the runtime model interpreter. The storyboard instance is generated using Xtend. To start execution, the storyboard, an instance of `IQueryModelInstance`, and `IAdaptModelInstance` is required. Then, a token is created and visiting the start node. Thereafter, the runtime visits (cf. visitor pattern [Gamma et al., 1995]) all storyboard elements that are active (e.g., can be visited and executed).

During developing the modeler has to state naturals and role adaption scenarios in story patterns. The stated role reconfigurations (e.g., add role) cannot be validated at compile time because the model can be used with multiple role models. Inspired from interfaces offered by CROM and CROI [Kühn, Böhme, et al., 2015] the `IQueryModelInstance`, and `IQueryModel` have been developed to query the role model on both model and instance level. Thus, an implementation may use these interfaces to verify that stated role reconfigurations are defined in the role model and are therefore valid. To emphasize fault tolerance it should be adjustable if the runtime should fail or use approaches like graceful degradation.

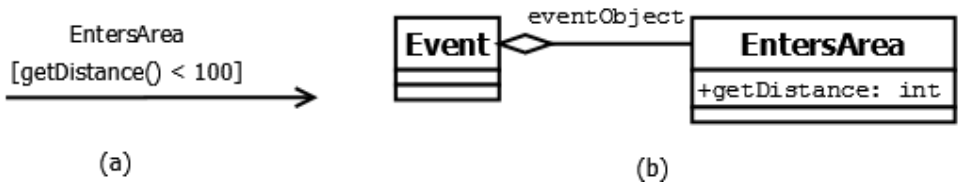

Figure 5.6.: An UML class diagram depicting the how context objects e.g., data objects, can be included into the event system to further formulate guards on the events. (a) shows a transition with event and guard, while (b) shows a class diagram of what is suspected to be issued as event to the storyboard runtime.

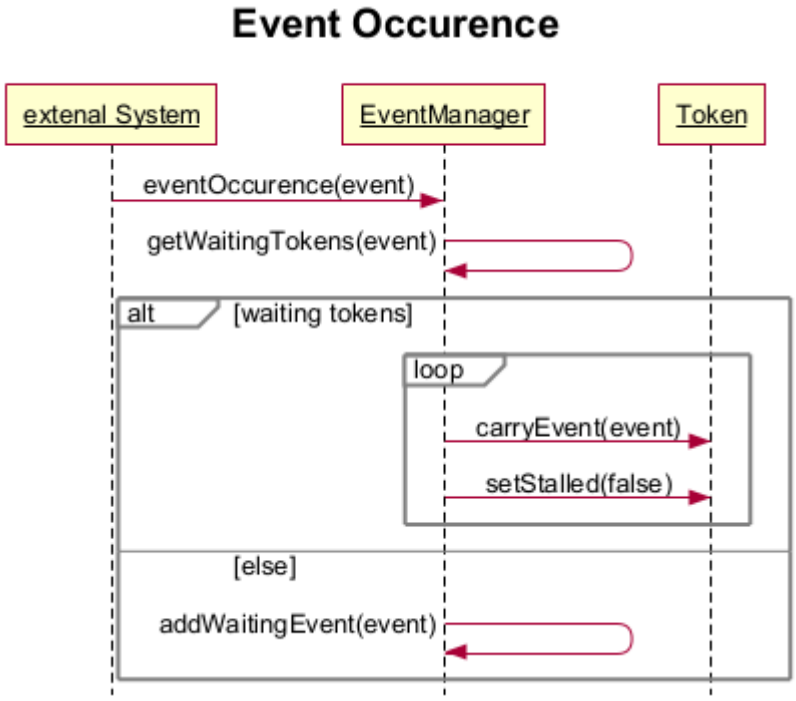

Figure 5.7.: UML sequence diagram showing system behavior when an event occurs.

Given a specific role type model (e.g., at instantiation time) the execution unit could verify the validity w.r.t. the constraints in the role type model. For example, it could check the role type set of each class in the story pattern if the stated role reconfigurations are allowed.

5.3. SUMMARY

This section introduced the implementation of context-aware storyboards with roles (CAESAR) which has been conceptually introduced in chapter 4. CAESAR takes up the suggestions of Thomas Kühn storyboards with roles [Kühn, 2011], and extends the concept with events, and guards. Furthermore, the concept has been changed to model a whole system instead single methods.

The implementation has been done in the Eclipse ecosystem using Java, the Eclipse Modeling Framework (EMF), and Xtext to implement a DSL. The system is layered into tools, runtime, and model and orthogonal there is the runtime and the modeling-time components.

The system is developed in way allowing that role programming languages can be adopted as soon as their role meta-model can be adapted to an instance of the CROM that is being used by CAESAR. Therefore, a query and a manipulation interface is provided which need to be implemented by the user of the system. The modeling-time storyboard model is transformed into a runtime model using Xtend. This runtime model is being executed (i.e. interpreted) instead of generating plain Java code. The role model used is an instance of the meta-model family provided by CROM. This meta-model defines what instances of role models can be adapted. To classify the design decisions for the meta-model of the role model 26 questions have been answered. At runtime type checking can be done for the actual role model that is bound to the storyboard. The analyzer can check if stated role reconfigurations are valid w.r.t. the role class model. For example it could be checked if all roles are contained in the role-type set of the specified natural.

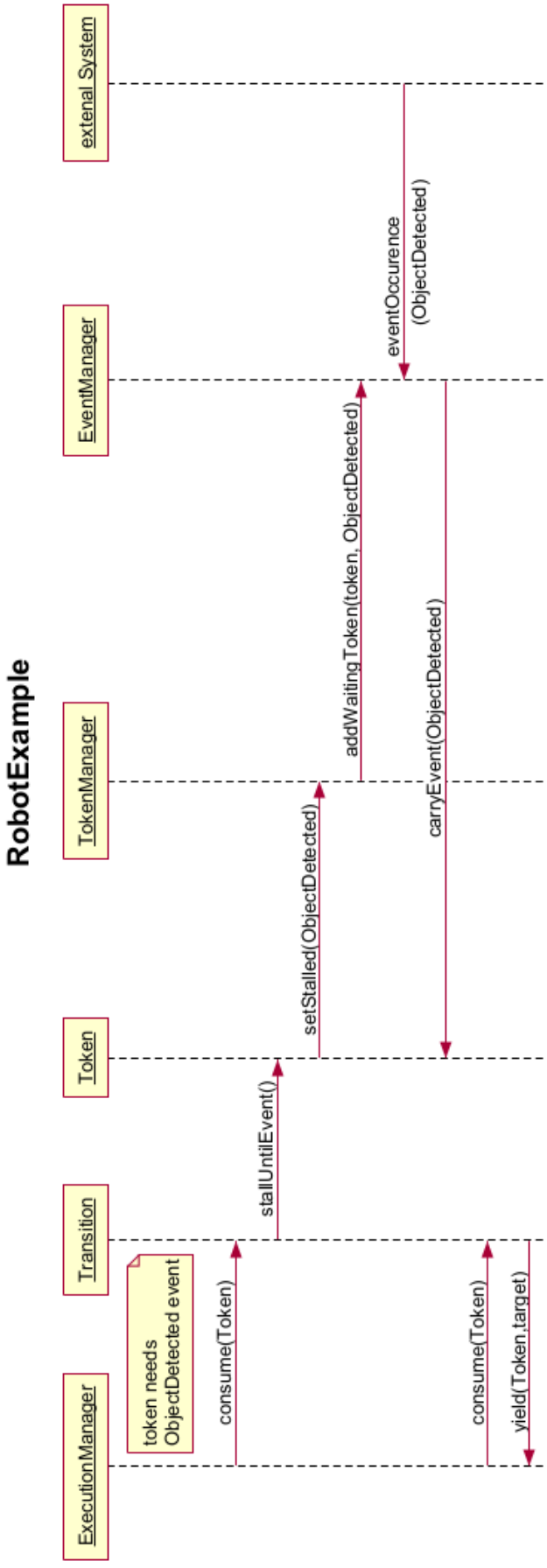

Figure 5.8.: A sequence diagram about the system behavior when tokens reach a transition before the event happened.

6. RELATED WORK

This chapter will present related work. Therefore, every approach is introduced, discussed, and compared to the result of the thesis. It is especially pictured, how every approach could be implemented with the concept of the thesis. At least a summary is drawn which includes a table comparing all approaches to context-aware storyboards with roles altogether.

6.1. CONTEXT-AWARE MIDDLEWARE FOR URC SYSTEM

Kim et al. have proposed a middleware to support context-aware services for robots [Kim et al., 2005] called CAMUS (Context-Aware Middleware for URC System). URC (Ubiquitous Robotic Companion) is a concept promoted by the Korean government which tries to improve services a robot delivers yet reducing the price of the robot. Therefore, the robots rely on external sensing, external processing power. The actual result of computation results in better services the robot should deliver. Kim et al. defined a *software robot* which represents the robot of the physical world in the cyber space. Beside formulating goals of the software robot being ubiquitously e.g., having the ability to dynamically adapt code parameters or code fragments, it should also be intelligent using knowledge to try out new things, or improve the knowledge with learning. Furthermore, the software robot should be capable to capture, represent and process context information, thus being context-aware. The paper just introduces context-awareness.

Therefore, they showed a conceptual model and explained how the physical world is mapped into the cyber space the software robot is interacting on. CAMUS consists of build-time components, and runtime components. The build-time components comprise the *sensor modeler* which models the mapping from physical sensors to sensor services in the cyber space. The *service modeler* describes input and output of service implementations such as device control. The *environment modeler* is used to model environments and available resources within them. The *user modeler* models user information, such as profiles and preferences. Context information is represented as Universal Data Model (UDM) which represent context information as nodes and associations between them. Tasks are implemented in a rule-based programming language which allows to react on events.

The approach allows to define the reactions of the robot in a context-aware manner. This is achieved by modeling tasks as event-action systems. However, the many models that have to be designed a priori prevent anticipated adaption. Not only the environment has to be known a priori (environment modeler), but also single elements that will be used by the robot as external sensor (sensor modeler), or are controled by the robot (service modeler). Even the people interacting with the robot have to be known (user modeler). Furthermore, the approach just concentrates on context-awareness. Adaption is future work. Roles are

not mentioned in order to achieve behavioral adaption of the robot (which is future work either).

6.2. CONTEXT PETRI NETS

Nicolas et al. introduced context petri nets (CoPN) to formally model the informal definition of context dependencies in Subjective-C, a Context-Oriented Programming language extension for Objective-C [Nicolás et al., 2012]. This dependencies are defined in a Context Definition Language (CDL). The example comprises a phone which has different behaviors related to the battery level. For example the `LowBattery` adaption could weakly include the `Ignore` adaption which would deny incoming calls other than incoming calls from VIP.

Therefor, a framework for defining context-definitions, mapping these into a petri net and executing the petri net have been developed. It functions as a bridge between the high-level definition of adaptions and context dependencies by the programmer and the orchestration of activation and deactivation of adaptions. Furthermore, context dependency definitions of Subjective-C have been formalized.

The adaption relationships are similar to relations between role types in a role model. The *weak inclusion* is similar to *role implication* in role modeling. Adaptions could be stated as roles in the context-aware storyboard with roles. The adaption dependencies need to be formalized in the role model and are out of scope of the context-aware storyboard with roles.

6.3. AGENT-BASED AND CONTEXT-ORIENTED APPROACH FOR WEB SERVICES COMPOSITION

Maamar et al. proposed an approach for agent-based and context-oriented web service composition [Maamar et al., 2005]. They base the term context related to the context of web services. The goal of the approach is to define a meta-model for web service conversation to integrate conversations and context for agent-based composition of web services. They define composite services as a composition of several services. Such composite services are described with a *service chart*, which is an extension to state charts. Beside *proactive composition*, which is already pre-compiled offline they concentrate on *reactive composition*, which is on-the-fly composition. They state an advantage of the reactive approach is that the compositions can be overseen and reaction can be taken when problems arise. The agents used in the approach are *composite-service agent*, *master-service-agent*, and *service-agent*, and there are contexts holding different views onto the system, namely $\mathcal{W}$*-context*, $\mathcal{I}$*-context*, and $\mathcal{C}$*-context*.

A web service is seen as a component which is instantiated with every composition. Related to the web service there is a $\mathcal{W}$-context. This context is maintained by the master-service-agent, and holds records like number of instances and maximum number of instances of a web service, the execution status of each web service deployed, and the request time vs availability of the web service. The master-service-agent is responsible to check if a certain web service is allowed to join a composition. This can be denied e.g., when the service has a high failure rate. With instantiation, the web service instance plus a new context, the $\mathcal{I}$-context is created. Furthermore, a service-agent is connected with the $\mathcal{I}$-context to keep the $\mathcal{W}$-context up-to-date. The $\mathcal{I}$-context holds information regarding quality of service (QoS) of the service e.g., time of completion, kind of completion (success, failure), execution status (in-progress, terminated). The composite-service-agent is used to trigger the specification and monitor the deployment of the specification. Composite service specifications are taken out of a store from the composite-service-agent. Thereby

the composite-service-agent initializes a $\mathcal{C}$-context. The $\mathcal{C}$-context consists of all information about the web services part of it and a combination of their $\mathcal{W}$-contexts.

The approach is using context in a much more narrow way as this thesis does. The different contexts hold different views onto the system. Where $\mathcal{I}$-context is fine grained, and $\mathcal{C}$-context is coarse grained, and $\mathcal{W}$-context is in between. Adaption is performed with the composite-service-agent preparing the next to be called web service. Thus, it can foresee problems regarding QoS and recompose the composed web service. This introduces some kind of self-adaptiveness w.r.t. recomposition of composite web services. Context-awareness is limited to the $\{\mathcal{W}, \mathcal{C}, \mathcal{I}\}$-contexts.

6.4. MODEL DRIVEN DESIGN OF SERVICE-BASED CONTEXT-AWARE APPLICATIONS

Grassi et al. propose an approach with the ideas of model-driven development and aspect-oriented design [Grassi et al., 2007]. The core business logic can be extended with context-aware adaption (i.e., context handling logic) without changing core logic. This context-aware adaption is scoped to the design process of SOA-based (Service-Oriented Architecture) applications. Grassi et al. adopt this separation of concerns approach because they define context-aware adaption as orthogonal concept to the core logic of the application designed. The paper proposes a meta-model for context and adaption, and showcases an UML framework which adapts the meta-model to implement context-aware adaption at design time using stereotyped UML activity diagrams for modeling. The paper does not propose an implementation of the proposed modeling methodology, but showcases how to implement the specific scenario with AspectJ, an Aspect-Oriented Programming (AOP) language for Java.

With this modeling methodology context has to be known at design time and cannot be implemented at runtime. Grassi et al. distinguish between *state-based context*, and *event-based context*. State-based context is defined as any attribute related to the entity (e.g., language, time, location) or functions of other contexts which are used as a source for the attribute. Event-based context is defined as events relevant to the entity (e.g., service invocation, internal events). Composite contexts are allowed. Furthermore, there are *state constraint* and *event constraint*. The state constraint is a logical predicate on some value, where the event constraint defines a pattern of events. *Context-aware bindings* and *context-aware inserts* are defined as association to values for the former, and the addition of structural elements or behavior for the latter. Thereby, two types of context-aware bindings are used: *structural insert* and *behavioral insert* where structural inserts correspond to intertype declarations in AOP, and behavioral inserts correspond to advices in AOP.

The UML framework for design models introduces the role of *Monitor* and *Adapter* where the monitors goal is to check if context constraints are satisfied, and notify the adapters. The adapter has the goal to adapt the application. The monitor uses *StateBasedContext*s to provide relevant data (where the StateBasedContext defines the actual source of the data), and *EventBasedContext*s to define triggering events and resulting signals of that event based context.

Behavioral Inserts are triggered by signals and add behavior to the core application. Thus, they allow for context-adaption at design time. State based context, event based context, event based constraint, and behavioral insert are modeled with UML activity diagrams.

The paper does not suggest a methodology to map the models to executable code, nor any special framework that has to be used. Despite Grassi et al. say that the methodology is used for SOA-based applications their own use case is implementing the model solely with

AspectJ[1] on simple method calls. The mapping from model to code has to be done by hand which shows that the approach concentrates on design models. The approach does not use roles for adaption and does not employ any kind of self-adaptiveness. If a context is not foreseen at design time the application will not be able to handle it. There is no knowledge inference or adaption in any way.

In a context-aware storyboard with roles the player will be the application itself. Therefor, the application has to be modeled and will be adapted which will add and alter its behavior. Event based constraints can be modeled as events, and the signals will be the adaption of roles to the application. Behavioral inserts have to be implemented in the roles themselves. Otherwise, the storyboards with roles need to be extended to allow statement activities (cf. section 2.2.3) which allows to inline code into activities.

6.5. SUMMARY

The related work presented was all about context and context-aware adaption. There have been different approaches starting from the Service-Oriented Architecture (SOA) based approach, over Petri Nets (PN) to Aspect-Oriented Programming. Table 6.1 compares every presented approach with the context-aware storyboards with roles. Therefore, the approaches are assessed regarding context-awareness, self-adaptiveness, if roles are used, and regarding the type of modeling methodology used to implement the approach. Every approach has been related to context-aware storyboards with roles and has been pictured how to be implemented with the concept of this thesis.

Approach	Context-Aware	Self-Adaptive	Roles	Type
CAMUS [Kim et al., 2005]	●	○	○	SOA
CoPN [Nicolás et al., 2012]	●	●	○	PN
agent-based and context-aware web service composition [Maamar et al., 2005]	●	◐	○	SOA
model driven design of service-based context-aware applications [Grassi et al., 2007]	●	○	○	AOP,SOA
context-aware storyboards with roles	●	●	●	SB

Table 6.1.: Comparison of the features of different storyboards approaches

●- supported ◐- partly supported ○- not supported

SOA (Service-Oriented Architecture), AOP (Aspect-Oriented Programming),
PN (Petri Net), SB (Storyboard)

CAMUS (Context-Aware Middleware for URC System) [Kim et al., 2005] is a context-aware approach, because context is directly included in the calculation of the service robot (e.g., temperature, the interacting user). However, it does not allow for adaption. This is marked as future work. Furthermore, the concept of roles is not used nor mentioned.

CoPN (Context Petri Nets) [Nicolás et al., 2012] is a context-aware and self-adaptive approach to model adaption relationships as petri net, and execute the petri net to orchestrate the enabling and deactivation of adaptions. It provides a framework in Objective-C and

[1] AspectJ website: https://eclipse.org/aspectj/ visited on 2016/06/06

implements the adaption in Subjective-C. Self-adaption of the application is achieved by adapting to specified contexts (e.g., `Ignore` in case of `LowBattery` context).

Maamar et al. have proposed an agent-based and context-aware web service composition [Maamar et al., 2005]. They define a meta-model over the conversations of web services among themselves. Therefor, web services can be composed to composite web services. Context is used to govern the adaption process and to choose from alternative web service components if necessary (e.g., if a specific web service downtime exceeds a specific threshold). Because of the protocol definition of conversations it can be foreseen that elements of a composite web service are unavailable to fulfill a task. The composite web service is recomposed, where the web service in question is exchanged.

Grassi et al. propose a modeling methodology to emphasize the orthogonal nature of context handling logic w.r.t. the core business logic [Grassi et al., 2007]. Therefore, they take the idea of Aspect Oriented Programming (AOP) and SOA to propose an UML framework for design models of context-aware adaption. The adaption of context-awareness happens at design time and has to be implemented. The framework does not constrain or propose any specific implementation strategy. However, the shown use case is implemented using AspectJ, an AOP language for Java.

There have been shown different approaches to showcase context-awareness for different use cases and at different places of the development cycle (modeling time vs. runtime). The difference in meaning of context come from the wide definition of context, which is defined as "any information that can be used to characterize the situation of an entity" [Dey, 2001]. However, context-aware storyboard with roles could always be used to adapt the scenario of each approach. This is because of the use of roles as abstraction and because context in context-aware storyboards with roles is a conceptual term that can be mapped in the implementation e.g., to an internal event, or a happening in the real world.

7. EVALUATION

This chapter will evaluate the developed context-aware storyboards with roles (CAESAR). Therefore, the system will be applied. The use case is a robotic co-working example. In the future, more and more robots will join to work with humans. Beside having full automatized assembly lines already today, they will begin to work side at side with humans. For such a scenario, Haddadin et al. have developed a robotic co-working approach to reduce the amount of errors, which result for safety reasons in a full stop of the production environment [Haddadin et al., 2009]. Therefore, escalation layers have been introduced before an emergency stop is executed. Thus, the robotic co-worker can work autonomously, but also in cooperation if humans are sensed in the working area. The overall goal is to minimize errors and thus a full stop of the robot or assembly line. Using Smart Grid Applications (SMAGs), a self-adaptive system using roles [Piechnick, Richly, et al., 2012], a robotic co-working scenario should be simulated cooperating with a KUKA robotic arm. The scenario is modeled, and the role reconfiguration is coming from CAESAR. Second the result of the thesis is evaluated against some criteria and compared to existing techniques.

7.1. USE CASE ROBOTIC CO-WORKER

Robotic co-working will become more and more important in the future. Using robots for production has been mainly driven by automotive industry, where robots are working caged off from humans [Sauppé et al., 2015]. But today's technology allows robots to work collaboratively with humans. Therefore, Haddadin et al. have developed the robotic co-worker [Haddadin et al., 2009]. Working autonomously the robot can sense when people are entering its working area. Therefore, it changes into a collaborative mode which enables further sensing capabilities, as well as the possibility to activate more fault tolerant behavior instead of ceasing to work and change into an error state [Lee et al., 1985].

Grasping the idea of a robotic co-worker CAESAR will be used in a scenario where a KUKA robotic arm is going to be working collaboratively. Using Smart Application Grids (SMAGs) (cf. section 2.1.3) the robot is able to work collaboratively when a human is near the robot, and autonomously otherwise. Collaboration will be manifested in a different location where the robot is dropping the working items as working autonomous. Figure 7.1 shows a storyboard depicting the scenario, where Listing 7.1 shows a CAESAR model that is modeling the described scenario. This model can be executed next to SMAGs, which will use CAESAR to plan the role reconfiguration for the different contexts.

```
events {
  import de.larsschuetze.usecase.robot.event.
     InPerceptionEvent as InPerception
  import de.larsschuetze.usecase.robot.event.
     OutOfPerceptionEvent as OutOfPerception
}
storyboard de.larsschuetze.usecase.robot.Collaboration
start StartNode
end EndNode
fork ForkNode
storypattern AutonomousRobot {
  class de.larsschuetze.usecase.robot.Robot {
    add Autonomous in Collaboration
    remove Collaborative in Collaboration
  }
}
storypattern CollaborativeRobot {
  class de.larsschuetze.usecase.robot.Robot {
    remove Autonomous in Compartment
    add Collaborative in Collaboration
  }
}
storypattern Error {
  class de.larsschuetze.usecase.robot.Robot {
    add Error in Collaboration
  }
}
transition Started { StartNode -> ForkNode }
transition Started1 { ForkNode -> PerceptingHuman }
transition Started2 { { Error } ForkNode -> Error }
transition PerceptingHuman { { InPerception } AutonomousRobot
   .success -> CollaborativeRobot }
transition NotPerceptingHuman { { OutOfPerception }
   CollaborativeRobot.success -> AutonomousRobot }
transition Error1 { AutonomousRobot.failure -> Error }
transition Error2 { CollaborativeRobot.failure -> Error }
transition Exit { Error.success -> EndNode }
```

Listing 7.1: A CAESAR model for a collaborative robotic co-worker

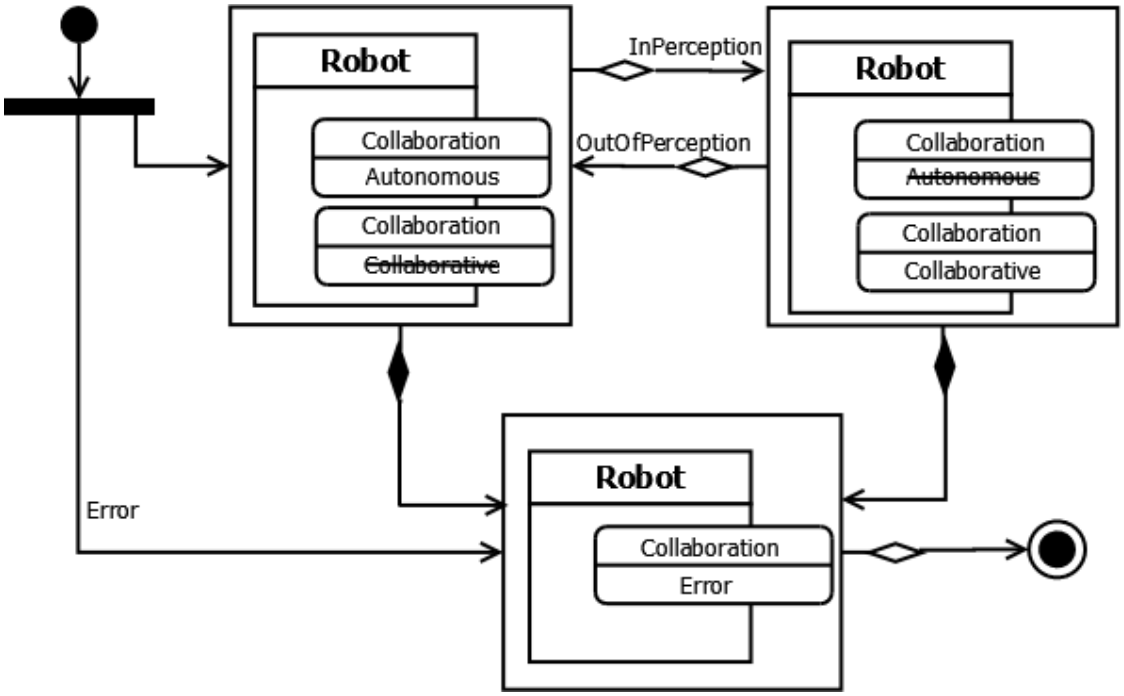

Figure 7.1.: A model of the scenario modeled in CAESAR.

7.2. RESULTS

In this section the developed context-aware storyboard with roles will be validated against the criteria formulated in section 3.2.

R-1 is fulfilled as the concept has been extended by compartments. R-2 is fulfilled as events can be received in order to forward the automata (e.g., stalled tokens that are waiting for events), and the context model can be called with guards. However, there is still room for improvement as the access to the context is very limited right now. The SAS needs to pass the right context objects to the storyboard in order to satisfy the guard. There is no type system just the `Event` needs to implement the interfaces the guard is calling. R-3 is fulfilled as the storyboard is implemented as a runtime model which is subject of execution by an interpreter. The system can be adapted to a SAS. Therefore, the runtime just needs a CAEASAR model that is going to be interpreted. To advance the state of the model, events can be passed from outside. R-4 requires CAESAR to work independent on the actual role runtime (i.e., role programming language) used. This is achieved, as the system defines interfaces that need to be adapted for every role programming language used. These interfaces allow CAESAR to query the role-play graph and to formulate adaption commands. This implies that R-5 is fulfilled already partially. Negative queries can be implemented such that CAESAR runtime iterates over a reduced set of naturals, roles, and compartments in the plays relation. If an instance is matched that for example plays a forbidden role it will not be matched.

The overall goal was the separation of concerns of business logic, context logic, and role adaption logic. Business logic remains outside of CAESAR encapsulated in the roles. This is, because roles override behavior, and thus define the business rules w.r.t. to themselves. For example the robot is autonomously taking an item and dropping it into the bin. If a person is near it will adapt its behavior and give the item to the person instead. The business logic is encapsulated in the roles. Role adaption is stated within the story patterns, and the context that leads to a situation is specified at the transitions (e.g., events and guards). Thus, the separation of concerns has been achieved.

Thus, all requirements stated in 3.2 could be met. Furthermore, the overall goal of separation of concerns has been achieved.

In the origin storyboard approach classes can be bound to variables (see figure A.1), which can be referenced in later executed story patterns or guards. This is possible, because a storyboard in their approach models a single method which is translated into a single method in Java code. Explicit parallelism is not allowed. In CAESAR, parallelism is explicitly allowed and modeled using fork and join. Furthermore, CAESAR models a whole system and is not in control of the role-play graph. Matched naturals and roles could therefore be already unbound and thus would cause failures if stored for later reference. Thus, variable binding is not allowed in the approach developed in this thesis.

7.3. SUMMARY

The implementation of the described scenario using context-aware storyboards with roles has shown that CAESAR is easily adaptable to existing solutions. Therefore, SMAGs has been used as self-adaptive system, as well as role runtime to model the adaptive behavior of a robotic co-worker. The separation of concerns approach used by CAESAR allowed to model the scenario in a concise manner. Furthermore, it allows to grasp the adaptive behavior of complex systems as well as the (extrinsic) influences that lead to an adaption.

It has been shown that all posed requirements are met. The system allows for separation of concerns for business logic, context logic, and role adaption logic. The business logic is encapsulated in the roles themselves. The context logic is defined with events and guards at transitions. The role adaption logic is contained in story patterns. Unanticipated adaption cannot be achieved as the system needs to be modeled a priori.

8. CONCLUSION AND FUTURE WORK

This chapter will summarize the presented work, and propose future work to extend or improve the context-aware storyboard approach proposed by the thesis.

8.1. CONCLUSION

This thesis developed a concept to model context-aware role adaption using a new and adapted concept based on storyboards. Motivation was that today's role programing languages all suffer from the problem of tangling w.r.t. different aspects i.e., context logic, adaption logic, and business logic.

With the importance of self-adaptive systems, and the relation between adaption and roles, building tomorrows large, self-adaptive software-systems lead to hard to understand, and unmaintainable code. The problem is at the root of the languages used and could not be circumvented with traditional methods used in programming language engineering. The need of a separation of concerns approach arises to separate the different aspects. This allows to handle and view the different aspects separately and leads to encapsulated, maintainable systems.

Chapter 2 introduced to the concept of roles and role programming, and the relation between self-adaptive systems, context-aware systems, and event-based systems. Modeling the change of software pieces (e.g., addition and removal of relations between objects) using storyboards is also presented. Together, they form the basis of the concept of the thesis to introduce a separation of concerns approach. In chapter 3 the current problems have been pointed out, as well as the goals to solve these problems. These goals led to the requirements the concept and implementation have to meet.

The concept developed in this thesis is presented in chapter 4. Therefore, the storyboard approach is extended and changed. The syntax and semantics for the different parts of the storyboard - story patterns to model role adaptions as part of the adaption logic, and transitions, events, and context to define the circumstances under which these adaptions are taking place i.e., the context logic - are introduced. The meta-model is presented to show how the defined goals are mapped into the concept. At last the concept is differentiated with the original concepts it is inheriting from.

The concept is the groundwork for the prototypical implementation presented in chapter 5 called context-aware storyboards with roles (CAESAR). Implemented as a set of Eclipse plugins mainly using the Eclipse Modeling Framework (EMF). There has been implemented the modeling part and the runtime part. The first consists of a meta-model based DSL to model storyboards as well as a code generator to generate a runtime model. The latter consists of an interpreter to execute the runtime model as well as interfaces. A plus of the

architecture is the separation of the storyboard and existing solutions (e.g., self-adaptive systems) which can adapt CAESAR. Furthermore, the solution developed is independent on the actual role programming language (or role runtime) used because of the query and adapter interfaces based on an instance of the meta-model family provided by Compartment Role Object Model (CROM). This allows to use any role programming language which is able to adapt its role concept to the used CROM instance.

Related work has been compared to the concept of context-aware storyboards with roles in chapter 6. Therefore, different self-adaptive and context-aware approaches have been conducted and evaluated. Then a relation with CAESAR is drawn. It has been shown that CAESAR is able to handle the same problems as some of the approaches presented in the related work.

In chapter 7 the prototypical implementation has been evaluated on the use case of a robotic co-worker where a robot's behavior is adapted to cooperate with human workers when they are near or working autonomously otherwise.

8.2. FUTURE WORK

The separation of concerns approach regarding business logic, context logic, and role adaption logic provides big potential to users of CAESAR. Despite the already gained usage this approach still allows to be further improved. Especially regarding the context logic definition, and role adaption logic which are the main parts contributed by CAESAR. Future work that has been identified is formulated in the following:

Visual Modeling As shown in the use case (see section 7.1) graphical modeling of role adaption w.r.t. extrinsic context is much more understandable, and maintainable (compare listing 7.1 vs. figure 7.1). The visual model is much more consise as the textual representation. This will be even more true, when story patterns are extended to match much more complex scenarios (e.g., using relations between roles). In a future version there could be implemented a graphical modeling editor for context-aware storyboards with roles which allow to build graphical models.

Context Constraint Language Since self-adaptive systems almost always implement a variant of the MAPE-K loop, where a knowledge base is shared across all four phases of the system it immediately suggest itself to be able to state constraints (i.e. guards) on this knowledge base. For example, Piechnick et al. defined a context model which offers contextual information in space and time, and introduced a context model query language (GRoCoMo-QL) [Piechnick, Püschel, et al., 2014]. This would allow the possibility to state assumptions over the context. Then the guard could directly state restrictions on the context without the need of an event.

Improved Meta-Model for CAESAR The meta-model for context-aware storyboards with roles is very limited currently. Beside the weak possibility to state context constraints, it does not allow to state relations between roles in story patterns. This has been left open for future work due to time constraints. Furthermore, the Story Driven Modeling (SDM) community developed a meta-model for storyboards as used in Fujaba [Heinzemann et al., 2011]. The meta-model has shown usable extensions to the current concept such as *optional nodes*, which could be transferred to *optional roles*. This would improve the expression of context-aware storyboards with roles tremendously, as well as support the reuse of existing story patterns.

Improved Story Patterns Currently story patterns just allow to model single, unrelated roles. Thomas Kühn has suggested to be able to formulate constraints on the natural, as well as on the roles. In addition, it should be possible to formulate constraint on the compartments as well. This would improve the current situation of pattern matching which is undeterministic. Especially when multiple matches are found.

The role adaption commands are currently a stream of calls to the adaption API. Thus, the external system does not know when the adaption as a result of a successful match begins and ends. This needs to be improved by providing a simple transaction API (e.g., `transactionBegin` and `transactinEnd`). This would add semantics for the adaption process, as well as the possibility to further improve the adaption in the external system. The same could be applied to the query part, where especially a database could improve execution speed of the query.

Furthermore, there could be allowed more freedom for defining story patterns. For example, an optional name for naturals would allow to state scenarios where just the roles matter, and not the natural playing the role.

A Standard for the Storyboard Runtime The current runtime model is plain Java code. In the future this runtime model can be implemented using modeling technologies (i.e., Eclipse Modeling Framework). Furthermore, lifting the execution of context-aware storyboards with roles to a standardized model (e.g., fUML explained in section 3.3) would improve quality as well as acceptance. Thus, in future work it should be investigated how fUML would allow to define and execute CAESAR runtime models. In [Mayerhofer et al., 2012] the fUML reference implementation was extended by AspectJ to introduce standard in-conform changes without changing the core application. It can be used to test UML activity diagrams [Mijatov, 2012].

A. APPENDICES

A.1. GRAMMAR FOR STORYBOARDS WITH ROLES

```
grammar de.larsschuetze.storyboard.Dsl
with org.eclipse.xtext.xbase.Xbase

generate dsl "http://www.larsschuetze.de/storyboard/Dsl"

Storyboard:
    'events' '{'
        importedEvents += Event+
    '}'
    'storyboard' name = QualifiedName
    elements += AbstractElement*
;

AbstractElement:
    Node | Transition
;

Node:
    ControlNode | StoryPatternNode
;

ControlNode:
    StartNode | EndNode | ForkNode | JoinNode | MergeNode
;

ForkNode:
    'fork' name = ValidID
;

JoinNode:
    'join' name = ValidID
;

MergeNode:
```

```
35      'merge' name = ValidID
36  ;
38  StartNode:
39      'start' name = ValidID
40  ;
42  EndNode:
43      'end' name = ValidID
44  ;
46  Event:
47      'import' eventType = JvmTypeReference 'as' name = ValidID
48  ;
50  Transition:
51      'transition' name = ValidID '{'
52          '{' event = [Event] guard = Guard? '}'
53          source = [Node]('.'sourcePort = StoryPatternPort)?
                '->' target = [Node]
54      '}'
55  ;
57  Guard:
58      '[' guard = GuardValue ']'
59  ;
61  GuardValue:
62      (ID | STRING | INT | '(' | ')' | '.' | OpCompare |
           OpEquality)+
63  ;
65  StoryPatternPort:
66      SuccessPort | FailurePort
67  ;
69  FailurePort:
70      'failure'
71  ;
73  SuccessPort:
74      'success'
75  ;
77  StoryPatternNode:
78      'storypattern' name = QualifiedName
79      '{'
80          'class' className = QualifiedName '{'
81              (roleReconfigurations += RoleNode)+
82          '}'
83      '}'
```

```
84  ;
86  RoleNode:
87      (ConditionRoleNode | RewriteRoleNode) 'in'
           compartmentName = QualifiedName
88  ;
90  RewriteRoleNode:
91      AddRoleNode | RemoveRoleNode
92  ;
94  RemoveRoleNode:
95      'remove' name = QualifiedName
96  ;
98  AddRoleNode:
99      'add' name = QualifiedName
100 ;
102 ConditionRoleNode:
103     PlayRoleNode | ProhibitedRoleNode
104 ;
106 ProhibitedRoleNode:
107     'prohibits' name = QualifiedName
108 ;
110 PlayRoleNode:
111     'plays' name = QualifiedName
112 ;
```

Listing A.1: DSL of Storyboards with Roles

A.2. EXEMPLARY OF A STORY DIAGRAM

Story diagrams (see section 2.2.3) model methods. In the example depicted in figure A.1 taken from [T. Fischer et al., 2000] the method `House::doDemo()` is modeled. `THIS` is the self reference. In case of execution the instance of the `House` that the method is called on is referenced. Classes are referenced by their name using the notation of class diagrams `instanceName:className`. This is, because during execution of the storyboard matched variables will be stored to be used and referenced later. Green means that a link, or object, or attribute will be added or changed. Red means that the link, or object, or attribute will be removed. Beside story patterns normal activites are supported, too. For example activity 7 and 8 are such a normal activity.

A.3. META-MODEL OF CONTEXT-AWARE STORYBOARDS WITH ROLES

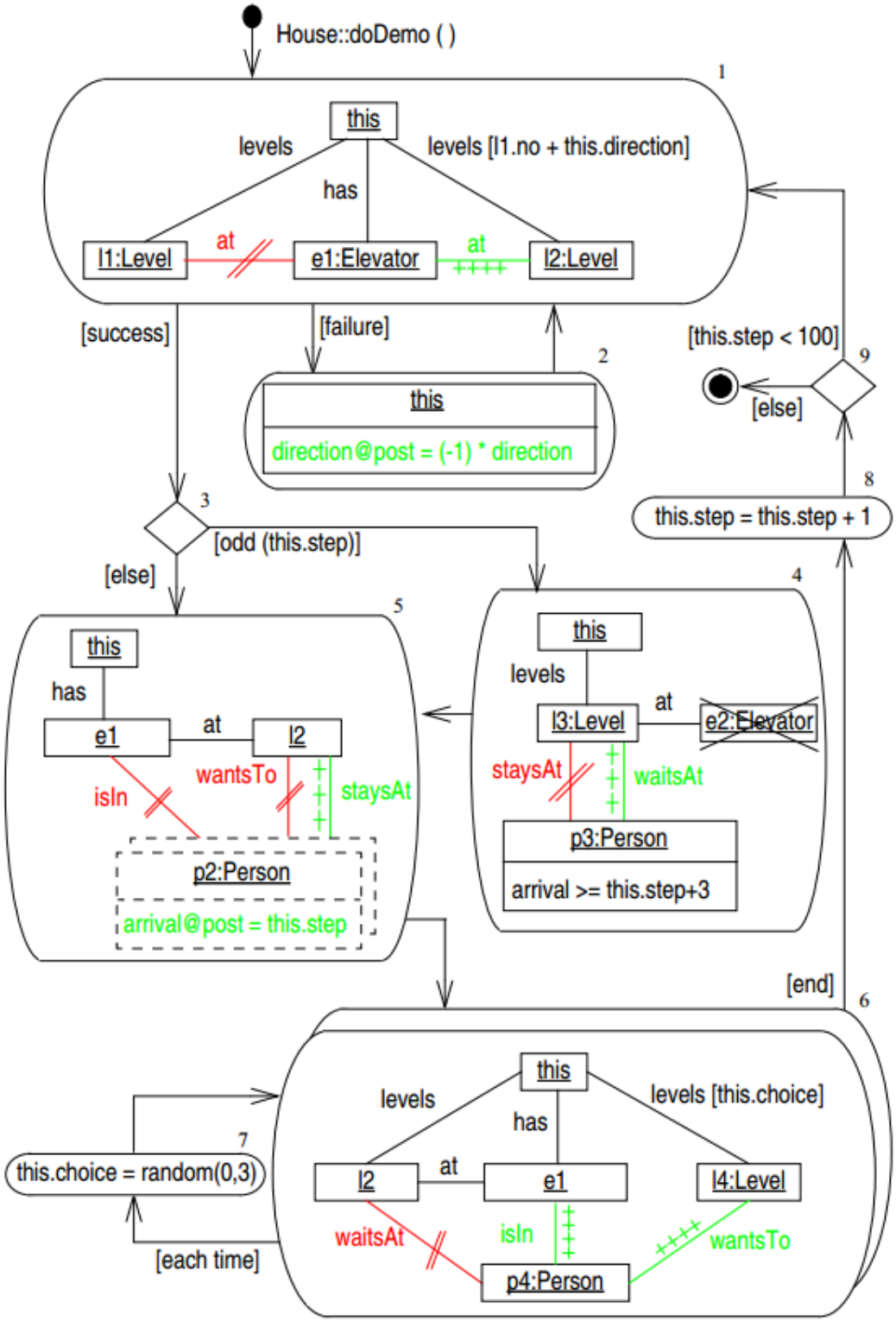

Figure A.1.: Example of a story diagram [T. Fischer et al., 2000]

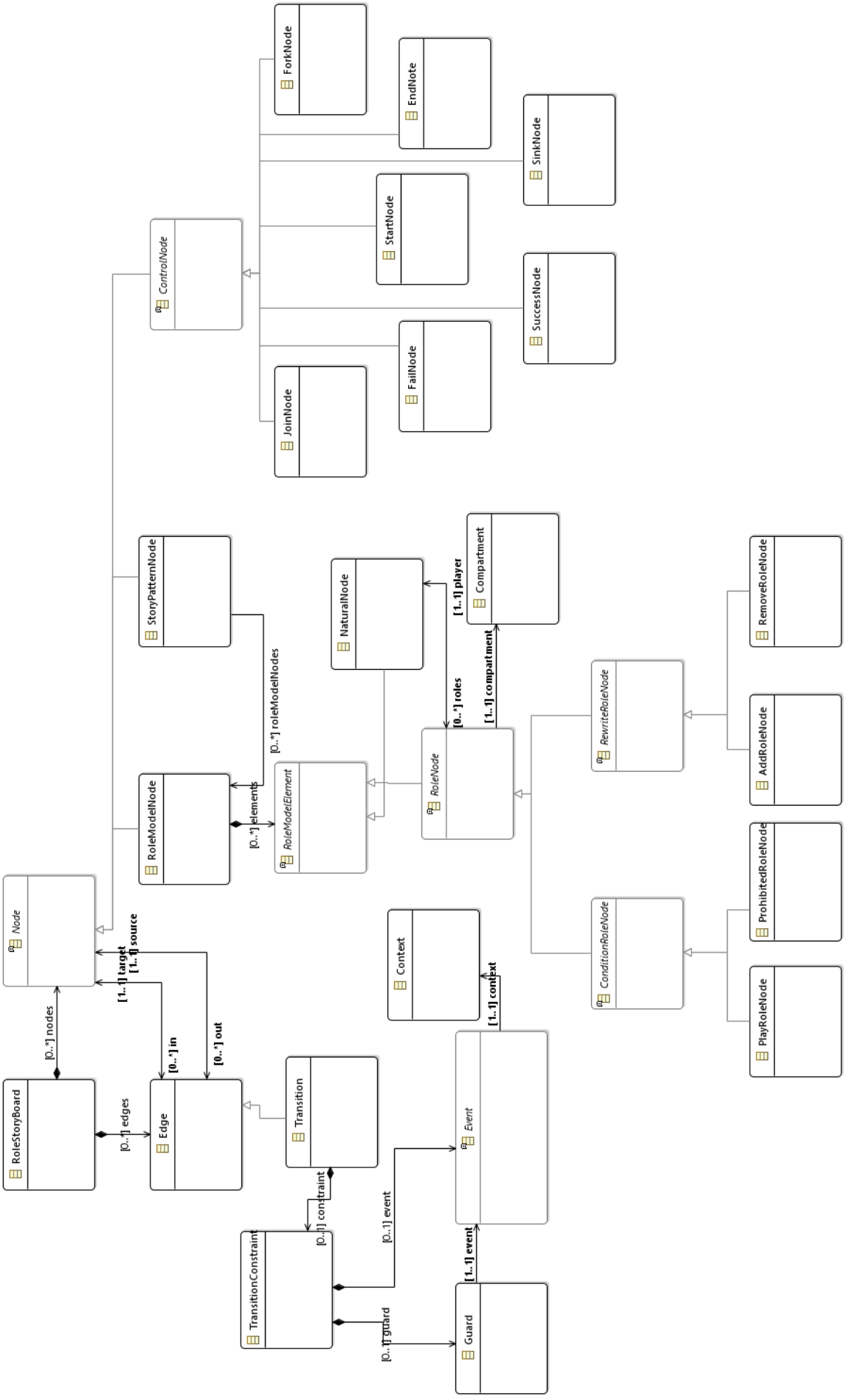

Figure A.2.: Meta-model of the Storyboard which is clearly three parted structured into context and events, role reconfiguration, and control flow elements.

ACRONYMS

fUML Semantics of a Foundational Subset for Executable UML Models

SAS Self-Adaptive System

CEP Complex Event Processing

CAAA Context-Aware Adaptive Application

ALF Action Language for Foundational UML

DSL Domain Specific Language

OMG Object Management Group

UML Unified Modeling Language

OCL Object Constraint Language

OOP Object-Oriented Programming

AOP Aspect-Oriented Programming

CROM Compartment Role Object Model

CROI Compartment Role Object Instances

PN petri net

FSM Finite State Machine

OT/J ObjectTeams/Java

DCI Data, Context, Integration

MOF Metaobject Facility

XMI XML Metadata Interchange

CPN Colored Petri Net

API Application Programming Interface

EBS Event-Based System

ROP Role Object Pattern

SCROLL SCala ROles Language

AD activity diagram

JVM Java Virtual Machine

RoSI Role-based Software Infrastructures

PIM Platform-Independent Model

PSM Platform-Specific Model

SMAGs Smart Application Grids

ECA Event-Condition-Action

UDM Universal Data Model

MDSD Model-Driven Software Development

EMF Eclipse Modeling Framework

OWL Web Ontology Language

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

BIBLIOGRAPHY

Andersson, Jesper, Rogério De Lemos, Sam Malek, and Danny Weyns (2009). "Modeling dimensions of self-adaptive software systems". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 5525 LNCS, pp. 27–47.

Antinyan, Vard et al. (2014). "Defining technical risks in software development". In: *Proceedings - 2014 Joint Conference of the International Workshop on Software Measurement, IWSM 2014 and the International Conference on Software Process and Product Measurement, Mensura 2014*, pp. 66–71.

Arcaini, Paolo, Elvinia Riccobene, and Patrizia Scandurra (2015). "Modeling and Analyzing MAPE-K Feedback Loops for Self-Adaptation". In: *Proceedings - 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015*, pp. 13–23.

Baldoni, Matteo, Guido Boella, and Leendert van der Torre (2006). "powerJava: ontologically founded roles in object oriented programming languages". In: *SAC {'}06: Proceedings of the 2006 ACM symposium on Applied computing*, pp. 1414–1418.

Barrenechea, Es, Psc Alencar, Rolando Blanco, and Don Cowan (2011). "Context-Awareness and Adaptation in Distributed Event-Based Systems". In: *Wwwtest.Cs.Uwaterloo.Ca*, pp. 1–24.

Bäumer, Dirk, Dirk Riehle, Wolf Siberski, and Martina Wulf (1998). "The Role Object Pattern". In: *Washington University Dept. of Computer Science*.

Bergmans, Lodewijk M J, Mehmet Akşit, Ken Wakita, and Akinori Yonezawa (1992). "An Object-Oriented Model for Extensible Concurrent Systems: The Composition-Filters Approach". In: *IEEE Transactions on Parallel and Distributed Systems* 92.87, pp. 2–12.

Brun, Yuriy et al. (2009). "Engineering self-adaptive systems through feedback loops". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 5525 LNCS, pp. 48–70.

CDIF Technical Committee et al. (1994). "CDIF–CASE Data Interchange Format". In: Extract of Interim Standard, EIA/IS-107, Electronic Industries Association.

Charles W. Bachman (1973). "The programmer as navigator". In: *Communications of the ACM* 16.11, pp. 653–658.

Chrszon, Philipp, Clemens Dubslaff, Christel Baier, Joachim Klein, and Sascha Klüppelholz (2016). "Modeling Role-Based Systems with Exogenous Coordination". In: *Theory and Practice of Formal Methods*. Springer, pp. 122–139.

Czarnecki, K. and S. Helsen (2006). "Feature-based Survey of Model Transformation Approaches". In: *IBM Syst. J.* 45.3, pp. 621–645.

Das, Sajal K., Dilip Sarkar, V.K. Agrawal, and L.M. Patnaik (1991). "Extended colored Petri net: An efficient tool for analyzing concurrent systems". In: *Information Sciences* 54.3, pp. 191–218.

David, Rene and Alla Hassane (1994). "Petri Nets for Modeling of Dynamic Systems A Survey". In: *Automatica* 30.2, pp. 175–202.

Davis, James P and Ronald D Bonnell (1999). "Role-playing : A Mechanism for Bridging the Object-Oriented Design-Level Gap". In: *OOPSLA-97: Workshop on Object Technology, Architectures and Domain Analysis*, pp. 1–8.

De Lemos, Rog??rio et al. (2013). "Software engineering for self-adaptive systems: A second research roadmap". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7475 LNCS, pp. 1–32.

Detten, Markus von et al. (2012). *Story Diagrams - Syntax and Semantics*. Tech. rep. tr-ri-12-320. Paderborn: University of Paderborn, Heinz Nixdorf Institute, Software Engineering Group, p. 106.

Dey, Anind K (2001). "Understanding and Using Context". In: *Personal and Ubiquitous Computing 5 (1)* 5.1, pp. 4–7.

Dey, Anind K and Gregory D Abowd (1999). "Towards a Better Understanding of Context and Context-Awareness". In: *Computing Systems* 40.3, pp. 304–307.

Diethelm, I, L Geiger, T Maier, and A Zündorf (2002). "Turning Collaboration Diagram Strips into Storycharts". In: *24th International Conference on Software Engineering (ICSE 2002), Workshop: Scenarios & State machines*, pp. 1–6.

Diethelm, Ira, Leif Geiger, and Albert Zündorf (2003). "Story Driven Modeling and programming with Fujaba". In: *Fujaba Days 2003*, p. 53.

Dietrich, Jens and Catherine Mccartin (2012). "Scalable Motif Detection and Aggregation". In: Adc, pp. 31–40.

Dustdar, Schahram, Harald Gall, and Manfred Hauswirth (2013). "Event-basierte Systeme". In: *Software-Architekturen für Verteilte Systeme*. Springer-Verlag.

Ehrlich, Franz (2016). "Towards a Type System for Roles". Diploma Thesis. Technische Universtität Dresden.

Eppstein, David (1995). "Subgraph Isomorphism in Planar Graphs and Related Problems". In: *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms* 3.3, pp. 632–640. arXiv: 9911003 [cs].

Fischer, Gerhard (2012). "Context-aware systems". In: *Proceedings of the International Working Conference on Advanced Visual Interfaces - AVI '12*, p. 287.

Fischer, Thorsten, Jörg Niere, Lars Torunski, and Albert Zündorf (2000). "Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java". In: *Theory and Application of Graph Transformations, 6th International Workshop, TAGT 1998*. Vol. 1764, pp. 157–167.

Fujaba (2005). "The Fujaba Tool Suite 2005: An Overview About the Development Efforts in Paderborn, Kassel, Darmstadt, Siegen, and Bayreuth". In: *FUJABA Days 2005*. September. Paderborn.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Garey, Michael R. and David S. Johnson (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co.

Genovese, Valerio (2007). "A Meta-model for Roles : Introducing Sessions". In: *Proceedings of the 2nd Workshop on Roles and Relationships in Object Oriented Programming, Multiagent Systems, and Ontologies*. Vol. 7. Berlin, p. 27.

Genrich, H. J. and K. Lautenbach (1981). "System modelling with high-level Petri nets". In: *Theoretical Computer Science* 13.1, pp. 109–135.

Grassi, Vincenzo and Andrea Sindico (2007). "Towards model driven design of service-based context-aware applications". In: *International workshop on Engineering of software services for pervasive environments in conjunction with the 6th ESEC/FSE joint meeting - ESSPE '07* 11, pp. 69–74.

Guermazi, Sahar, Jérémie Tatibouet, Arnaud Cuccuru, Saadia Dhouib, et al. (2015). "Executable Modeling with fUML and Alf in papyrus: Tooling and experiments". In: *CEUR Workshop Proceedings* 1560, pp. 3–8.

Guermazi, Sahar, Jérémie Tatibouet, Arnaud Cuccuru, Ed Seidewitz, et al. (2015). *EXECUTABLE MODELING WITH FUML AND ALF IN PAPYRUS: TOOLING AND EXPERIMENTS*. website visited at 2016/06/08. Model Driven Solutions. URL: http://www.modelexecution.org/media/EXE2015/presentations/EXE_2015_slides_paper_3.pdf.

Haddadin, Sami et al. (2009). "Towards the Robotic Co-Worker". In: *International Symposium on Robotics Research (ISRR2007), Lausanne, Switzerland* 1, pp. 261–282.

Heinzemann, Christian, Jan Rieke, Markus Von Detten, Dietrich Travkin, and Marius Lauder (2011). "A new Meta-Model for Story Diagrams". In: *8th International Fujaba Days*, pp. 2–6.

Herrmann, Stephan (2005). "Programming with Roles in ObjectTeams/Java". In: *AAAI Fall Symposium on Roles- an interdisciplinary perspective, 2005*.

– (2007). "A precise model for contextual roles: The programming language ObjectTeams/-Java". In: *Applied Ontology* 2.2, pp. 181–207.

– (2010). "Demystifying object schizophrenia". In: *Proceedings of the 4th Workshop on MechAnisms for . . .* Pp. –4.

Herrmann, Stephan, Christine Hundt, and Katharina Mehner (2004). *Translation Polymorphism in Object Teams Translation Polymorphism in Object Teams*. Tech. rep. Technische Universität Berlin.

Herrmann, Stephan, Christine Hundt, and Marco Mosconi (2008). *ObjectTeams/Java Language Definition*. Version 1.1.

IBM (2005). "Autonomic Computing White Paper: An Architectural Blueprint for Autonomic Computing". In: *IBM White Paper* June, p. 34.

Jungel, M, E Kindler, and M Weber (2000). "The Petri Net Markup Language". In: *Proceedings of AWPN 2000 - 7thWorkshop Algorithmen und Werkzeuge f{ü}r Petrinetze*, pp. 47–52.

Keet, C. Maria (2013). "Open World Assumption". In: *Encyclopedia of Systems Biology*. Ed. by Werner Dubitzky, Olaf Wolkenhauer, Kwang-Hyun Cho, and Hiroki Yokota. New York, NY: Springer New York, pp. 1567–1567.

Kendall, E A (1999). "Role model designs and implementations with aspect-oriented programming". In: *Acm Sigplan Notices* 34.10, pp. 353–369.

Kim, Hyun, Young Jo Cho, and Sang Rok Oh (2005). "CAMUS: A middleware supporting context-aware services for network-based robots". In: *2005 IEEE Workshop on Advanced Robotics and its Social Impacts* 2005, pp. 237–242.

Kindler, Ekkart and Michael Weber (2001). "The Petri Net Kernel: An infrastructure for building Petri net tools". In: *International Journal on Software Tools for Technology Transfer* 3.4, pp. 486–497.

Krupitzer, Christian, Felix Maximilian Roth, Sebastian Vansyckel, Gregor Schiele, and Christian Becker (2015). "A survey on engineering approaches for self-adaptive systems". In: *Pervasive and Mobile Computing* 17.PB, pp. 184–206.

Kühn, Thomas (2011). "Explizite Rollenbindung mit Story Boards". Major Thesis. Technische Universität Dresden.

Kühn, Thomas, Stephan Böhme, Sebastian Götz, and Uwe Aßmann (2015). *A combined formal model for relational context-dependent roles*. Tech. rep. Technische Universität Dresden, pp. 113–124.

Kühn, Thomas, Max Leuthäuser, and Sebastian Götz (2014). "A Metamodel Family for Role-Based Modeling and Programming Languages". In: *Software Language Engineering. Springer International Publishing*, pp. 141–160.

Kühne, Thomas (2006). "Matters of (Meta-) Modeling". In: *Software & Systems Modeling* 5.4, pp. 369–385.

Lee, M.H., D.P. Barnes, and N.W. Hardy (1985). "Research into error recovery for sensory robots". In: *Sensor Review* 5.4, pp. 194–197.

Leuthäuser, Max (2015). "Enabling View-based Programming with SCROLL Using roles and dynamic dispatch for etablishing view-based programming". In: *Proceedings of the 2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering*.

Liu, Yan and Dong Wang (2010). "Complex Event Processing Engine for Large Volume of RFID Data". In: *2010 Second International Workshop on Education Technology and Computer Science* 51, pp. 429–432.

Luyten, Kris et al. (2010). "On stories, models and notations: Storyboard creation as an entry point for model-based interface development with UsiXML". In: *1st Int. Workshop on User Interface Extensible Markup Language*.

Maamar, Z., S.K. Mostefaoui, and H. Yahyaoui (2005). "Toward an agent-based and context-oriented approach for Web services composition". In: *IEEE Transactions on Knowledge and Data Engineering* 17.5, pp. 686–697.

Mayerhofer, Tanja and Philip Langer (2012). "Moliz: a model execution framework for UML models". In: *. . . Class on Model-Driven Engineering: Modeling . . .* Pp. 3–4.

Mens, Tom and Pieter Van Gorp (2006). "A taxonomy of model transformation". In: *Electronic Notes in Theoretical Computer Science* 152.1-2, pp. 125–142.

Mijatov, Stefan (2012). "Testing of UML Activity Diagrams". In: *9th Workshop on Model Driven Engineering, Verification and Validation MoDeVVa 2012*.

Miller, Jj (2013). "Graph Database Applications and Concepts with Neo4". In: *Proceedings of the 2013 Southern Association for . . .* Pp. 141–147.

Nanda, Umang, Shrey Rajput, Himanshu Agrawal, Antriksh Goel, and Mohit Gurnani (2016). "On Context Awareness and Analysis of Various Classification Algorithms". English. In: *Proceedings of the Second International Conference on Computer and Communication Technologies*. Ed. by Suresh Chandra Satapathy, K. Srujan Raju, Jyotsna Kumar Mandal, and Vikrant Bhateja. Vol. 381. Advances in Intelligent Systems and Computing. Springer India, pp. 175–181.

Nicolás, Cardozo, Sebastián González, Kim Mens, and Theo D'Hondt (2012). "Context Petri Nets: Definition and Manipulation". In: *Université catholique de Louvain, Vrije Universiteit Brussel*.

Object Management Group (2013). *Action Language for Foundational UML (ALF), Version 1.0.1*. October.

Object Management Group (2014). *Object Constraint Language - Version 2.4*. Online available specification.

Object Management Group (OMG) (2011). *OMG Meta Object Facility (MOF) Core Specification*. August.

– (2012). *Unified Modeling Language*. Online available specification.

O'Madadhain, Joshua, Danyel Fisher, Scott White, and Y Boey (2003). "The jung (java universal network/graph) framework". In: *University of California, Irvine, California*.

Oxford English Dictionary (2016). *"meta-, prefix"*. website visited at 2016/05/30. Oxford University Press. URL: http://www.oed.com/view/Entry/117150?rskey=iA9XAX&result=4.

Petri, Carl Adam (1962). "Kommunikation mit Automaten". ger. PhD thesis. Universität Hamburg.

Piechnick, Christian (2016). *Smart Application Grids*. website visited at 2016/06/27. URL: https://st.inf.tu-dresden.de/smags/?p=44.

Piechnick, Christian, Georg Püschel, et al. (2014). "Towards Context Modeling in Space and Time". In: *First Workshop on Model-Driven Robot Software Engineering*. Ed. by Gerd Wagner and Uwe Aßmann, pp. 1–12.

Piechnick, Christian, Sebastian Richly, Sebastian Götz, Claas Wilke, and Uwe Aßmann (2012). "Using Role-Based Composition to Support Unanticipated, Dynamic Adaptation - Smart Application Grids". In: *ADAPTIVE 2012, The Fourth International Conference on Adaptive and Self-Adaptive Systems and Applications* c, pp. 93–102.

Reenskaug, Trygve (2011). *A DCI execution model*.

Riehle, Dirk and Thomas Gross (1998). "Role model based framework design and integration". In: *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* 33.10, pp. 117–133.

Rozenberg, Grzegorz, ed. (1997). *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations*. River Edge, NJ, USA: World Scientific Publishing Co., Inc.

Sama, Michele, David S Ds Rosenblum, Zhimin Wang, and Sebastian Elbaum (2008). "Model-based fault detection in context-aware adaptive applications". In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering - SIGSOFT '08/FSE-16*, pp. 261–271.

Sauppé, Allison and Bilge Mutlu (2015). "The Social Impact of a Robot Co-Worker in Industrial Settings". In: *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems - CHI '15*, pp. 3613–3622.

Schilit, B. N. and M. M. Theimer (1994). "Disseminating active map information to mobile hosts". In: *IEEE Network* 8.5, pp. 22–32.

Schürr, Andy (1994). "Progres, a visual language and environment for programming with graph rewriting systems". In: pp. 1–21.

Seidewitz, Ed (2015). "Tool Paper : Combining Alf and UML in Modeling Tools – An Example with Papyrus –". In: *OCL 2015–15th International Workshop on OCL and Textual Modeling: Tools and Textual Model Transformations Workshop Proceedings*.

Sekharaiah, K Chandra and D Janaki Ram (2002). "Object Schizophrenia Problem in Object Role System Design". In: *OOIS '02: Proc. 8th Int. Conf. Object-Oriented Inf. Systems*, pp. 494–506.

Smaragdakis, Yannis and Don Batory (2002). "Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs". In: *ACM Transactions on Software Engineering and Methodology* 11.2, pp. 215–255.

Steimann, Friedrich (2000). "Formale Modellierung mit Rollen". PhD thesis. Universtität Hannover.

Tamai, T., N. Ubayashi, and R. Ichiyama (2005). "An adaptive object model with dynamic role binding". In: *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* Pp. 166–175.

The Object Management Group (2012). *Semantics of a Foundational Subset for Executable UML Models (fUML)*. October, p. 441.

Trygve Reenskaug and James O. Coplien (2009). *The DCI Architecture: A New Vision of Object-Oriented Programming*.

Yang, Nianhua, Huiqun Yu, Hua Sun, and Zhilin Qian (2010). "Mapping UML Activity Diagrams to Analyzable Petri Net Models". In: *2010 10th International Conference on Quality Software*, pp. 369–372.

Zhu, Haibin and Rob Alkins (2006). "Towards Role-Based Programming". In: *Computer Supported Cooperative Work*.

Zhu, Haibin and MengChu Zhou (2006). "Role-based collaboration and its kernel mechanisms". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 36.4, pp. 578–589.

Zündorf, Albert (2001). "Rigorous Object Oriented Software Development Draft". Habilitation. University of Paderborn.