Run-time Adaptation of Role-based Software Systems

Dissertation

submitted in partial satisfaction of the requirements
for the academic degree of Doktor-Ingenieur (Dr.-Ing.)

At
Technische Universität Dresden
Faculty of Computer Science

By
M.Sc. Martin Weißbach
Born on 23.09.1989 in Görlitz

First referee
Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill

Second referee
Prof. Nicolás Cardozo

Advisor
Prof. Dr. Thorsten Strufe

Date of Submission
October 25, 2017

Date of Defense
April 6, 2018

Acknowledgements

This thesis was written during my time as PhD student in the research training group "RoSI – Role-based Software Infrastructures for continuous-context-sensitive Systems". It was a privilege and pleasure to have had the opportunity to work within this group for the past three years in which I was allowed to completely focus on this thesis. I particularly want to thank my supervisor Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill for the constant feedback and advice. Furthermore, I would like to thank all members of the RoSI family for all the fruitful discussions resolving around Roles and for all the valuable feedback provided during the regular retreats of the group and all the meetings in between.

I especially want to express my deepest gratitude to Markus Wutzler, Thomas Springer and Nguonly Taing for all the discussions, feedback, support and their joint efforts in improving my papers. A special thanks to Ivonne Wittig who read the thesis from cover to cover several times for correcting all my grammar and spelling mistakes.

Last but not least, I want to thank my family, specifically my mother and my grandmother for all their love, support and encouragement.

Abstract

Self-adaptive software systems possess the ability to modify their own structure or behavior in response to changes in their operational environment. Access to sensor data providing information on the monitored environment is a necessary prerequisite in such software systems. In the future, self-adaptive software systems will be increasingly distributed and interconnected to perform their assigned tasks, e.g., within smart environments or as part of autonomous systems. Adaptations of the software systems' structure or behavior will therefore have to be performed consistently on multiple remote subsystems.

Current approaches, however, do not completely support the run-time adaptation of distributed and interconnected software systems. Supported adaptations are local to a specific device and do not require further coordination or the execution of such adaptations is controlled by a centralized management system. Approaches that support the decentralized adaptation process [15, 16], help to determine a stable state, e.g., defined by *quiescence* [26], of one adaptable entity without central knowledge ahead of the actual adaptation process. The execution of complex adaptation scenarios comprising several adaptations on multiple computational devices is currently not supported. Consequently, inherent properties of a distributed system such as intermittent connectivity or local adaptation failures pose further challenges on the execution of adaptations affecting system parts deployed to multiple devices. Adaptation operations in the current research landscape cover different types of changes that can be performed upon a self-adaptive software system. Simple adaptations allow the modification of bindings between components [14] or services as well as the removal or creation and integration of such components or services into the system. Semantically more expressive operations allow for the relocation of behavioral parts of the system [18].

In this thesis, a coordination protocol is presented that supports the decentralized execution of multiple, possibly dependent adaptation operations and ensures a consistent transition of the software system from its source to a desired target configuration. An adaptation operation describes exactly one behavioral modification of the system, e.g., the addition or replacement of a component representing a behavioral element of the sys-

tem's configuration. We rely on the notion of *Roles* [28] as an abstraction to define the software system's static and dynamic, i.e., context-dependent, parts. Roles are an intuitive means to describe behavioral adaptations in distributed, context-dependent software systems due to their behavioral, relational and context-dependent nature. Adaptation operations therefore utilize the *Role* concept to describe the intended run-time modifications of the software system. The proposed protocol is designed to maintain a consistent transition of the software system from a given source to a target configuration in the presence of link failures between remote subsystems, i.e., messages used by the protocol to coordinate the adaptation process are lost on transmission, and in case of local failures during the adaptation process.

The evaluation of our approach comprises two aspects: In one step, the correctness of the coordination protocol is formally validated using the model checking tool PRISM. The protocol is shown to be deadlock-free even in the presence of coordination message losses and local adaptation failures. In a second step, the approach is evaluated with the help of an emulated execution environment in which the degree of coordination message losses and adaptation failures is varied. The adaptation duration and the partial unavailability of the system, i.e., the time roles are passive due to ongoing adaptations, is measured as well as the success rate of the adaptation process for different rates of message losses and adaptation failures.

List of Publications

The following peer-reviewed publications cover the main contributions of this thesis:

[47] Martin Weissbach. "Adaptation Mechanisms for Role-Based Software Systems." In: *OTM Workshops* 9416.Chapter 1 (2015), pp. 3–4

[49] Martin Weissbach et al. "Decentralized coordination of dynamic software updates in the Internet of Things." In: *WF-IoT* (2016)

[48] Martin Weissbach and Thomas Springer. "Coordinated Execution of Adaptation Operations in Distributed Role-based Software Systems." In: *SAC 2017: Symposium on Applied Computing Proceedings*. New York, NY, USA: ACM, 2017, pp. 45–50

[50] Martin Weissbach et al. "Decentrally Coordinated Execution of Adaptations in Distributed Self-Adaptive Software Systems." In: *2017 IEEE 11th International Conference on Self-Adaptive and Self-Organizing Systems (SASO*. IEEE, 2017, pp. 111–120

The following peer-reviewed publications are closely related but do not immediately contribute to this thesis:

[21] Tobias Jäkel et al. "Position Paper - Runtime Model for Role-Based Software Systems." In: *ICAC* (2016)

[52] Markus Wutzler, Martin Weissbach, and Thomas Springer. "Role-Based Models for Building Adaptable Collaborative Smart Service Systems." In: *2017 IEEE International Conference on Smart Computing (SMARTCOMP*. IEEE, 2017, pp. 1–6

Contents

1. Introduction

In recent years, computing systems have become more pervasive, smaller in size and increased in their complexity. According to Gartner[1], in 2015 alone roughly two billion mobile phones have been shipped. Including other device categories such as tablets or notebooks, the number of shipped devices amounts to 2.6 billion. The majority of these devices is powerful enough to support fairly complex computations. Moreover, almost all mobile devices are basically able to collect information about the user's current situation, e.g., via motion or illumination sensors. Software applications running on those devices are in general distributed applications, i.e., parts of the application are executed directly on the device to interact with the user or to collect sensor information whereas other parts of the application are executed on remote servers performing other and possibly computationally more complex tasks. Smartphones, however, are not the only mobile device category that has become powerful enough to support the execution of complex software systems. Modern cars are likely to execute close to 100 million lines of code on approximately 70 to 100 microprocessor-based electronic control units [6] that control everything from breaks to transmission to engine control to airbags to GPS and the entertainment system. In the advent of autonomously driving cars, the software systems deployed to cars are likely to become even larger and more complex, especially if the cars are constantly communicating and collaborating with each other and with traffic control systems to conduct the autonomous driving task. Other application domains, e.g., the Internet of Things, indicate an increasing number of collaborating computing devices, too. Imagine, for example, a complex system in a smart home that determines which power source to use to prepare hot water for the household or the heating system based on the current weather conditions, the weather forecast, and the residents' routines.

Future computing systems will not only be complex and distributed across an arbitrarily large number of computing devices but will also rely on collected sensor information to adapt their behaviors or executed tasks in dependence of their computational or external environment or the user's momentary situation. Such adaptations have to be planned and coordinated across several layers of the software system's architecture. Moreover,

[1] http://www.gartner.com/newsroom/id/2645115 as of 02/08/17

the adaptations will have to be performed reliably and in a way that prevents the system from reaching undesired configurations that could be harmful for the system itself or, worse, the user(s). In this thesis, the phase of such an adaptation process that actually performs the modifications of a highly distributed software system at run time, e.g., modifying the system's structure, will be the research focus.

1.1. Motivation

Software systems developed nowadays are increasingly distributed across numerous different devices. Additionally, they are highly interconnected since parts of the system deployed to different devices collaborate with each other using networked communication channels. Classic *n-tier* or *peer-to-peer* architectures serve as intuitive examples for such highly distributed and interconnected software systems.

Future software systems will not only be distributed and interconnected but will also be able to monitor their computational environment and to modify themselves in response to changes in that computational environment. The information collected from different kinds of sensors will enable those software systems to adapt their exposed behavior, e.g., the offered functionality, in response to changed user situations or changes in their computational environment. In the following, two exemplary application scenarios for highly distributed software systems capable to monitor their computational environments and their users' situations in order to allow for behavioral adaptations will be presented. The third and last exemplary application scenario addresses the special case of updating already deployed and running software systems. Updating in this regard refers to the evolution of the software system, e.g., the introduction of new functionalities or behaviors or the removal of such in the case the functionality was rendered obsolete.

1.1.1. Application Scenario 1 – Autonomously Driving Cars

An autonomously driving car is not operating in isolation but has to cope with other autonomously driving cars. Arbitrarily many cars may group together while driving on a highway, thus creating a line of cars (*platoon*). Cars in this line expose the same driving behavior and frequently communicate while driving. As long as the external conditions, e.g., weather and road conditions, remain stable, no adaptation of the driving behavior of the platoon is required. Assuming rain to start falling, a different driving behavior might be exposed by the platoon, e.g., the distance between cars within the platoon is increased or the velocity with which the cars are driving is reduced, because wet roads can be slippery and it takes longer to decelerate. It may be dangerous for the passengers

of the cars if each individual car decides to switch its driving behavior from sunny to rain mode autonomously and without consensus from the other cars in the platoon.

In this exemplary scenario, each car is considered a software system realizing all the behaviors necessary for the car to drive. This also includes variants of specific behaviors that are ought to be performed only when certain conditions or situations are met, e.g., a specific driving behavior on wet roads for rainy weather. Although each car can be considered a software system on its own, the entirety of cars grouped together in a platoon is regarded as the software system that has to modify itself in response to changing weather or road conditions. In a first step, all cars would have to agree on the behavioral change in response to the changed environment. In a second step, the actual behavior of the overall system has to be changed in a coordinated manner. The behavioral adaptation of the system as a whole is conducted individually on each car. In order to ensure that each car performs the adaptation and activates the behavioral change at the same point in time, all participating cars have to collaboratively coordinate the adaptation process.

The most important property of the distributed software system described in this application scenario is its ability to execute several adaptations in a coordinated manner. The coordination is motivated by the mutual dependency of the adaptations to prevent inconsistencies within the software system.

1.1.2. Application 2 – Search-And-Rescue Robots

Search-And-Rescue Robots are deployed in locations too dangerous for humans to enter to perform certain tasks, such as rescuing people from partially collapsed buildings or contaminated sites. A group of robots is collaborating in such environments to perform their assigned tasks.

In this scenario, robots are searching in a partially collapsed building for survivors. At first, each robot is searching for victims. When a robot finds a victim or a casualty, the robot changes its behavior from searching to rescuing, which means the person or corpse will be carried out of the site as fast as possible. Sometimes, unforeseen obstacles occur during a rescue mission, which leads to unpredicted effort on the robot. This could cause the robot to run too low on energy to finish the retrieval task or the robot requires the help of other robots in order to overcome the obstacle. In such a case, application behavior could be transferred or cloned to other robots in the proximity to take over or aid the retrieval task.

The coordinated transfer of application behavior including internal state information to other parts of the distributed software system is evidently the major point addressed

within this application scenario. Coordination in such a case is required to ensure a safe and reliable transition of specific behavioral parts of the software system while it is running.

1.1.3. Application Scenario 3 – Evolution of Deployed Software Systems

In any domain related to the Internet of Things (IoT), such as Smart Cities, Smart Grids or a Smart Home, software applications are highly distributed across several layers of the system to generate or provide value-added services to the users or other parts of the system. Such software systems are likely to require updates that introduce new or updated functionalities in response to changes in the requirements of the application. Highly distributed and interconnected applications in the aforementioned domains cannot be stopped or shut down easily in order to allow the execution of the update. Consequently, updates have to be performed while the software system is running. Additionally, updates are likely to affect several parts of the application located on different devices, which requires the update process to be coordinated. The coordination is required to activate the updated or newly introduced system behavior in a consistent manner.

An energy broker scenario serves as use case for the evolution of such a highly distributed and interconnected software system in the domains of Smart Homes or the Internet of Things. Different energy-providing devices continuously propagate information about the current amount of produced energy via IoT gateways to a centrally organized cloud instance. Different energy-consuming devices can register their estimated future energy consumption with the centrally organized cloud instance. If enough energy is available, the cloud instance will grant the energy consumers the right to perform the tasks for which they registered the estimated amount of energy. Both energy consumers and producers are connected via IoT gateways to the centrally organized cloud instance. These gateways host several other parts of the application in order to translate and enhance the data delivered from the producers' or consumers' sensors.

An exemplary evolution of the distributed software system introduces a new data field, which is required to balance the produced and required energy. This new data field has to be introduced to components of the IoT gateways as well as to the centrally organized energy broker service in the cloud. To prevent data loss or false balancing decision, the new data field has to be available consistently in the system during and after the evolution process. Having the data field available only on the central energy broker and not on the gateways, for example, is undesirable since wrong balancing decisions will be made due to the expected but lacking information at the energy broker.

The coordinated execution of software evolutions while a highly distributed and in-

terconnected software system is running is the major issue outlined in this application scenario. Coordination in such a scenario is required to ensure a reliable and, most importantly, consistent update of the software system at run time. Consistent in this scenario means all devices and software parts affected by the update install and activate the changes in a bound timeframe.

1.1.4. Summary

The discussed application scenarios address different aspects of the adaptation and evolution of distributed software systems at run time. As a result of the rather broad scope of all scenarios combined, different properties for the respective software systems can be derived that are highly related to the adaptation and evolution of such systems. The most obvious system property that is also shared by all application scenarios is the distribution of the software system across multiple devices or computing nodes. The first application scenario described the adaptation of a group of autonomously driving cars. Adapting the system behavior of such a line of cars requires the execution of multiple interdependent adaptations, i.e., adaptations have to be performed on all cars in a synchronized and coordinated way. The second application scenario described the transfer of application behavior from one robot to another. In such a system, behavior including internal state information, can be freely transferred between devices. The transfer of the behavior, evidently, needs to be coordinated among the devices affected by this kind of adaptation. The third application scenario described the evolution of a software system at run time. Introducing new or removing old system behavior as well as updating already existing implementations of system behavior is a fourth property of highly distributed software systems this thesis is concerned with.

The summarized system properties are crucial to the management of run-time adaptations in distributed software systems and serve as a foundation for the requirements and research questions later presented in this chapter. As a first step in this direction, interesting research problems will be identified and discussed in the following section.

1.2. Problem Analysis

Different aspects of the previously outlined scenarios are already addressed by a large research body as Krupitzer et al. [27] have shown in a recent survey. Work has already been done in order to enable software systems to observe their computational environment and to represent the obtained information in computationally processable data structures. Abstracting from raw sensor data is crucial to be able to reason about the computational

environment or the user's situation and to detect changes in the environment or the user's situation. In order to react to those changes, the software system has to calculate appropriate steps to modify itself in response to the detected changes.

Future software systems in the domains of the Internet of Things or the Smart Grid / City will be distributed over a multitude of different devices. Thus, reasoning tasks about the system's computational environment and planning steps to react to the detected changes are likely to become a performance bottleneck due to the large amount of devices and their wide spacial distribution if performed on centralized computational units. The decentralization of such tasks is an essential advancement of the research. Approaches have already been presented that focus on a decentralized decision-making process, e.g., [32, 41], in order to generate plans that describe which parts of the system have to be adapted.

Proposed reference architectures focusing on the design and implementation of such adaptive software systems consider the adaptation of distributed applications but perform all the adaptation tasks from a central instance. The granularity of adaptations of the existing work is focused on the level of services or software components and rarely covers adaptations on the level of runtime objects that can directly be implemented using a programming language.

In summary, we discovered two major gaps in the current research landscape:

1. a platform-independent abstraction for adaptation prescriptions to adapt any given software system

2. a reliable and consistent execution of generated change plans in a decentralized manner across several devices

It is not to be expected that software systems in the IoT domain will be implemented using the same programming language and platform, e.g., runtime. To ensure the adaptability of a distributed software system that is executed in a heterogeneous environment, a platform-independent set of operators to describe the planned adaptations that are supposed to be performed is essential. Furthermore, a centralized execution and coordination of the adaptation process is likely to become a bottleneck in setups featuring a large logical and spatial distribution of software artifacts of the system. If the central coordinator fails during the adaptation process, the software system might suffer from service interruption due to incomplete adaptations. Therefore, performing the adaptation steps locally and coordinating the procedure in a decentralized manner is highly beneficial.

1.3. Terminology

We take a step back to introduce and discuss important terminology frequently used in the remainder of this thesis. A comprehension of the terms and concepts presented in this section will ease the upcoming presentation of requirements and research questions of this thesis.

In the first part of this section, terms related to the domain of self-adaptive software systems, towards which this thesis is highly related, will be discussed. Subsequently, the concept of roles as abstraction for dynamic system parts will be introduced rather briefly discussing only the most basic and general ideas of the approach. A more detailed discussion of the *Role* concept, especially in correlation to self-adaptive software systems, will be given in Chapter 2.3. Further information about the contribution of the *Role* concept to self-adaptive software systems developed within this thesis will be presented in Chapter 3.

1.3.1. General Self-Adaptive Software System Terms

Self-adaptive Software Systems (SASS) are software systems that are able to monitor their computational environment (*context-awareness*), and their own internal state (*self-awareness*). Furthermore, self-adaptive software systems are able to modify their own behavior or structure (*self-adaptation*) in response to changes in their computational environment.

A self-adaptive software system comprises two distinct subsystems [23]: the *autonomic manager* and the *managed element*. The autonomic manager is responsible for all tasks related to the adaptation process of the overall self-adaptive software system, which includes the self-awareness, context-awareness and self-adaptation properties of the SASS. The managed element is the actual application of which functionality or structure are modified at run time through adaptations. In the remainder of this thesis, we will refer to both terms as *adaptation management* and *managed application*, respectively, without any change to the responsibilities of both subsystems originally described by Kephart et al. [23].

The following terms are closely related to self-adaptive software systems, but may not necessarily originate from this domain.

Configuration

A *configuration* is the global state of the distributed application with respect to the implementation units of the software system. In a component-based software system, for

example, the configuration would denote the types and instances of all components, the deployment location of any given component with respect to computational devices, and the relation of the components amongst each other, e.g., binding information between components. A *consistent configuration* of the distributed application is therefore a configuration that satisfies a run-time model of the application for a given computational context.

Consistency & Transaction

Kramer and Magee define the *consistency* of an application by the "relationship between node application states [that] are usually described by a set of global invariants or constraints which must be preserved" [26]. This definition will be used throughout this thesis without any modification or restriction.

In [35], a *Change Transaction* is defined as a set of at least two operations. An operation modifies the managed application, e.g., modifying the internal structure of the managed application. The *Change Transaction* is furthermore defined to be atomic, i.e., either all operations complete without error or the managed application will remain untouched. As a consequence, intermediate changes have to be reverted if the change transaction is aborted due to occurred errors. We refer to this definition from here on forward as *Adaptation Transaction* with one minor adjustment: we require an *Adaptation Transaction* to contain only one operation at least.

Adaptation / Reconfiguration

The *adaptation* or *reconfiguration* of an application modifies the current configuration of the application (source) so that a different configuration (target) is reached. It takes an *adaptation script* or *reconfiguration script* as input that describes the steps or changes that have to be performed in order to reach a consistent target configuration of the distributed application from the current source configuration. The aim of the *adaptation* or *reconfiguration process* is the consistent transition of the application from the source to the target configuration described by the adaptation or reconfiguration script.

Distribution & Decentralization

In conformity with the definitions of distribution and decentralization given in [51], we define a *distributed* software application as an application of which parts are scattered across several different devices. All parts of such an application perform different tasks and collaborate to achieve the overall goal of the distributed application. If parts of the

application are replicated and distributed across multiple devices, e.g., multiple instances of one component type exist within the system performing basically the same task, and have to collaborate with each other in order to fulfill these tasks, we consider this part of the application to be not only distributed but also *decentralized*.

Coordination

The term *coordination* describes how the *autonomic manager* [23] performs its assigned tasks, e.g., the *self-adaptation* property. Performing adaptations at run time might involve different steps that need to be controlled and managed in their execution, which is referred to as *coordination*. For the decentralization of parts of the autonomic manager, coordination does not only describe the general execution of the overall procedure but also how the decentralized instances collaborate with each other in order to perform the assigned task. With respect to the realization of the self-adaptation property, coordination steps or a coordination mechanism therefore does not only include the general procedure how to modify the managed application, but also describes any kind of protocol, formula or other means to support the decentralization of parts of the autonomous manager responsible for the realization of the self-adaptation property. Of course, the same assumptions hold for the other properties of a self-adaptive software system.

1.3.2. The Role Concept in General

In the domain of self-adaptive software systems, the term *Role* is used as abstraction to describe different concepts. In multi-agent systems, for example, roles are used to describe responsibilities of autonomous agents in collaborations between them. Other approaches, e.g., as discussed in [44] or [24], use a role as abstraction for context-dependent system behavior in order to enable self-adaptation. In the following, the notion of a *Role* is introduced generally and the understanding of a *Role* assumed within this thesis will be briefly outlined.

The concept of *Roles* was first introduced by Bachman et al. [1] in the field of data modeling and has evolved to different other fields since. The evolution process, however, did not yield a unified definition for the term *Role*, which leads to an inconsistent terminology among different approaches. A shared commonality of all views on the role concept is the notion of the *Player* as a software entity that is able to play one or more roles. The correlation of role and player, however, may differ between different approaches and is partially related to the role's nature of the respective concept. In literature, three natures of roles can generally be distinguished:

Context-dependent: Roles have a context-dependent nature if they are active only within a specific scope. This scope can be an arbitrary situation, e.g., when a set of sensor readings reports values within a predefined range, or a specific configuration of the software application. Consequently, the software system's configuration changes when a situation change occurs or sensor readings report different values that would require different roles to be active.

Collaboration-dependent: Roles are collaboration-dependent if they can only exist in collaboration with other roles, e.g., a buyer depends on the existence of a seller in order to fulfill its intended functionality and vice versa. Software systems that rely greatly on collaborations of their software entities often make use of this abstraction to describe responsibilities of the single collaborators within the system.

Behavioral: A role is behavioral if it implements a specific behavior and/or has its own internal state. The provision of a behavior unique to the role allows the role to modify the behavior of its respective player, if desired.

Friedrich Steimann [40] compiled a set of features that characterizes roles at the design level (M1) and at run time (M0). This original set of role features was later extended by Kühn et al. [28] with context-capturing concepts, referred to as *Compartment*. The resulting set of 26 features of roles comprises a meta-family for feature models. Consequently, not every role-based approach may implement the same subset of the proposed features (the list of role features proposed by Kühn et al. [28] is presented in Table 3.2 on page 50 together with a comparison which features the approach proposed within this thesis requires). Some of the proposed features, especially within the first 15 proposed by Steimann [40], however, can be considered fundamental for role-based software systems, such as Feature 3 ("Objects may play different roles simultaneously") or Feature 7 ("Unrelated objects can play the same role"). Objects in Steimann's definition can also be referred to as players.

Within this thesis, roles and players can be compared to run-time entities such as objects known from object-oriented approaches, components or services. A role is coined by a *role type*, which describes the capabilities, e.g., attributes and methods denoting the behavioral nature of the role, as well as a *role instance*, which is the concrete instantiation of a role of a given type at run time. The same distinction holds true for players and compartments within this thesis with respect to type and instance. A comportment can be considered objectified context information [28] at design time to logically group related roles active in the same situation or involved in the same collaboration. Apparently, several instances of roles and players of a specific type can be present at run time. All

adaptations discussed in the remainder of this thesis are as well instance-based, i.e., only a concrete run-time representation of a given type is affected by an adaptation. In contrast, all existing instances of a specific type would have to adapt if adaptations were performed on the type level.

1.4. Requirements Analysis

Based on the previously described application scenarios and the conducted problem analysis, requirements are posed to the approach that is developed within this thesis. As already stated, the execution of adaptations at run time within a self-adaptive software system coined by a high degree of distribution of the managed application, is the main concern of this thesis. Consequently, the following requirements specifically address concerns of run-time adaptation in such systems.

1.4.1. Decentralized Execution of Adaptations

This requirement is mainly motivated by *Application Scenario 1*, but can also partially be related to the other two scenarios, especially to the software evolution application scenario. The major challenge of the first application scenario is the coordinated and reliable execution of multiple interdependent adaptations in the system. The adaptation management subsystem is not only required to control the execution of a single adaptation, i.e., changing the behavior of a single car, but has to control the adaptation of a group of cars.

The software system discussed in the application scenario is furthermore highly distributed and mobile, i.e., the cars are driving, thus constantly changing their physical location. Having a central instance coordinating and controlling the adaptation process of such a software system appears to be both undesirable and infeasible. A central coordinator introduces a single-point of failure to the system which may have a negative impact on the passengers' security. Additionally, the overhead required to coordinate the adaptation process from a central instance located afar is infeasible and the overall system would be prone to communication interruptions, which could hinder the adaptation process preventing timely adaptations.

The ability of the software system to coordinate the adaptations in this application scenario without a central coordinator is evidently a highly desirable and necessary requirement to pose on a self-adaptive software system applicable in both the first and second application scenario.

1.4.2. Ensure a Stable Application State before Adaptation

This requirement is motivated by all presented application scenarios. In all three application scenarios, the software systems are fully deployed and operational at the point in time the adaptation or evolution of the system is ought to take place. If parts of the system were updated or adapted, e.g., removed, without any precautions, the entire system could break because tasks might not have been finished yet or data important to the managed application might be lost.

Therefore, the coordinator of the adaptation process has to ensure that the adaptation is performed when the application is in a *stable state*. We define a *stable state* as a state of the application in which the parts of the application that are subject to the adaptation or evolution process do not carry out any computational tasks. This definition is generally in line with the more precise definitions given in [26, 7]. A more detailed discussion of application states that are safe for adaptation or evolution tasks to be performed will be given in Chapter 2.1.1. If necessary, the coordinator of the adaptation process should be able to restrict the managed application in a way that such a state can be reached.

1.4.3. Cope with Loss of Coordination Messages

All three application scenarios require a reliable coordination of an adaptation's execution process across device borders without any central instance, thus backing up this requirement. We assume to already have a mechanism in place that enables the decentralized coordination of the adaptation process for the decentralized management of the managed application. Having such a decentralized adaptation process, the responsible peers have to exchange messages to control the progress of the adaptation. In dynamic and possibly unstable environments, such as the one described in the search-and-rescue robots scenario, those control messages may never arrive at their destination. The mechanism coordinating the adaptation process is ought to be able to detect and tolerate lost coordination messages, i.e., the adaptation process should not immediately fail because of infrequently lost coordination messages.

1.4.4. Cope with Local Adaptation Failures

The last requirement is mainly motivated by application scenarios one and two, but applies to the third application scenario, too. During the adaptation process, locally conducted adaptations on the respective computing devices may fail at run time or may not be completed. In the case of the migration of application behavior in the search-and-rescue robot scenario, the adaptation would not complete correctly if one of the robot's

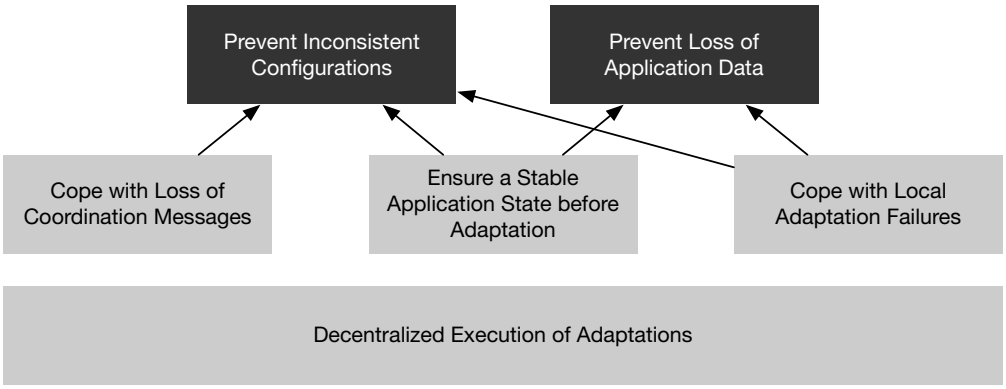


Figure 1.1.: Overview of requirements and their relation to the system's goals.

computing systems fails.

The adaptation management should be able to determine such errors during the adaptation process and handle them appropriately. At this point, we assume the coordination process of the adaptation to be decentralized, i.e., no central coordinator exists. Consequently, the handling of local adaptation failures during the adaptation process ought to be addressed by the decentralized coordination mechanism, too.

1.4.5. Summary

The requirements discussed in this section are supposed to be met by the solution proposed in this thesis, which aims to coordinate the execution of adaptations in a distributed self-adaptive software system. While the self-adaptive software system is undergoing adaptations at run time, two major goals exist that the adaptation management subsystem is required to meet: First, the managed application must be ensured to remain in a consistent configuration during the adaptation process, which is expected to always result in a consistent application configuration. This goal has often been mentioned implicitly as a system goal in the discussion of the requirements and in the problem analysis. The second goal of the adaptation management is to prevent the managed application from losing data either during or as a result of the adaptation process, since the managed application's performance can be severely affected adversely by such a data loss. In Figure 1.1, the posed requirements are summarized and their contribution to achieve the system goals is indicated by arrows. The requirement to execute the adaptation process without a central coordinating instance in the running system does not contribute to the fulfillment of any goal immediately. This requirement is a more fundamental one addressing single-point-of-failure and performance-bottleneck issues of a centralized coordinator in such highly distributed system setups as it was previously discussed.

1.5. Research Questions

This thesis is concerned with the investigation of three major research questions closely related to the previously discussed research problems and posed system requirements.

1. *How can a stable application state in a distributed application be reached in order to allow for multiple adaptations being performed on multiple computing devices simultaneously or in an otherwise coordinated manner?*
   The first research question is correlated to the second requirement and supports the goal to prevent data loss of the managed application during the adaptation process. The coordinated execution of multiple interdependent adaptations across device borders is immediately addressed by this research question.

2. *How can a decentralized management and coordination of an adaptation process composed of multiple interdependent adaptations be realized that*
   a) copes with the loss of protocol messages and network partitioning, and
   b) copes with node failures or adaptation errors during the adaptation process?
   The second research question is motivated by the decentralized coordination requirement necessary due to the possibly high degree and mobility of the managed application or parts of it. Requirements three and four are subsumed by this research question, since the handling of adaptation or node failures at run time or the tolerance to lost protocol messages are typical properties of a decentralized protocol.

3. *What is a suitable abstraction to describe the adaptations supposed to be performed on the managed application platform-independently and is that description able to support the execution of changes?*
   The issue of platform-independent descriptions of the adaptations that are supposed to be carried out at run time has not yet been raised in any of the application scenarios but was identified as an open issue in the current research landscape. This research question, however, is partially supported by the first and third application scenario, which were previously discussed. In an environment with a multitude of devices from possibly different vendors, application developers are likely to favor different platforms to use for the application's implementation. An adaptation approach that abstracts from the concrete platforms while still being able to support the decentralized coordination of the adaptation or evolution of a software system in a heterogeneous device platform environment is therefore greatly desirable and thus explicitly added as a third research question to the scope of this thesis.

1.6. Thesis Outline

The remainder of this thesis is structured as follows: In Chapter 2, foundations on adaptation aspects of self-adaptive software systems that specifically address the issue of finding a stable state will be the subject of discussion. Furthermore, existing research approaches addressing the decentralization of the execution of adaptations at run time, will be discussed as well as the relation of the *Role* concept to the research domain of self-adaptive software systems.

In Chapter 3, the contributions of this research work with respect to the utilization of the *Role* concept as abstraction for context-dependent application behavior in self-adaptive software systems in order to enable both collaborative and behavioral adaptations will be presented.

In Chapter 4, the developed approach to enable the decentralized adaptation of multiple correlated adaptations in response to changes in the system's operational environment will be discussed. A brief overview on the assumed system structure and failure models that the approach can handle during the run-time adaptation will be given first. Subsequently, the developed protocol to coordinate adaptations will be presented. At first, the supported modifications of the self-adaptive software system supported by the protocol will be outlined. In a second step, the execution of such modifications will be discussed from a local point of view. Having introduced the local adaptation management, the decentralized execution of adaptations with the help of the developed protocol will be introduced. A brief outline will be given how the approach can be used for the evolution of a self-adaptive software system before the contributions presented in this chapter will be summarized.

In Chapter 5, the previously presented work will be evaluated. The evaluation comprises a formal evaluation of the developed approach to be free of deadlocks for the assumed system structure and failure models, and of a qualitative assessment of the performance of the approach within an emulated environment.

In Chapter 6, the presented approach will be briefly summarized and compared to the posed requirements and research questions. A look ahead on interesting research areas that arise from the results of this work will conclude this thesis.

2. State of the Art

In this chapter, we present and discuss existing research approaches and results related to the work conducted in this thesis. The landscape of existing research work is split into two major subsets: First, we will discuss state-of-the-art approaches in the domain of self-adaptive software systems and correlated disciplines. Second, we will discuss existing work related to this research as well as approaches which serve as a possible application for the results presented in this thesis. An overview of the content of this chapter is given in Figure 2.1.

State-of-the-Art approaches comprise approaches that are competing with our proposed results or on which the work presented in this thesis immediately relies on, e.g., through the usage of the proposed solutions or the extension of existing work to fit our requirements. Sections 2.1 and 2.4 will focus on the state-of-the-art research landscape. *Related Work and Applications* covers approaches that follow a similar concept, but are lacking results on the level of the actual execution of adaptations at run time. The integration of results discussed in this section would be mutually beneficial, but is not subject to this thesis. Sections 2.2 and 2.3 will focus on related work and other possible application domains for the results presented in this thesis.

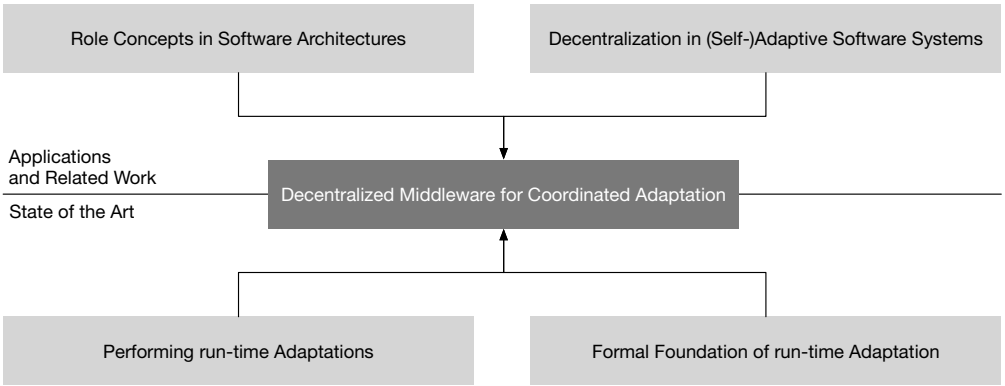


Figure 2.1.: Overview of the topic fields of state-of-the-art approaches and related work.

2.1. Formal Foundation

Research question one is concerned with finding a state of the managed application at which it is safe to perform the adaptation without negative side effects due to an inconsistent configuration of the application or data loss. The first part of this section discusses existing definitions of such a *stable state* for the managed application. The second part outlines a formal method present in literature that discusses how to reach such a stable state for any given managed application.

2.1.1. Stable Application States

Performing structural adaptations on a running application is not a simple task. The adaptable entities of the system that cause the structural adaptation, e.g., components or objects, are likely to execute certain tasks or to process method invocations. Removing such an adaptable entity at a randomly chosen point in time or as soon as the adaptation request is issued will therefore likely lead to message loss, inconsistent application behavior or a failure of parts of the system or the entire system in the worst case. The challenge to find or reach a state in which it is safe to perform adaptations has been widely acknowledged. In the field of *mobile code* and *object migration* [39] or in the CORBA component architecture [5] the first step of a migration is considered the suspension of the object or component that is supposed to be migrated. In the following, we discuss two approaches that enable a system to determine reliably when a safe state is reached in component-based software systems in order to modify its structure. The approaches discuss solutions to determine at which point in time a component, thus, the entire system, is in a stable state that allows the adaptation without data loss or inconsistent application behavior.

Quiescence

Reaching a safe state is an essential prerequisite to change an adaptive software system dynamically at run time and was already introduced by Kramer & Magee in 1990 as a concept called *quiescence* [26]. The concept was introduced in accordance with the adaptation of component-based applications and relies on the knowledge of the communication patterns and interconnections across components. A series of message exchanges between components is called a *transaction* [26].

A component in the approach of Kramer & Magee implements three states: *active*, *passive* and *quiescent*, which are depicted in Figure 2.2. In the *active* state, a component

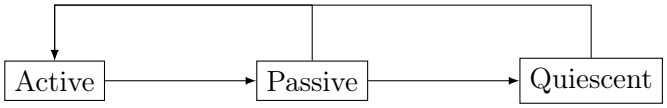


Figure 2.2.: State chart for components to reach the quiescent state.

can operate without limitation, i.e., the component can initiate, accept and service transactions. In the *passive* state, a component continues to accept and service transactions, but is currently not engaged in an ongoing transaction nor will the component initiate a transaction itself. Both states may occur naturally throughout the components life cycle. According to [26], a component is in a *quiescent* state, if (1) the node is not engaged in a transaction it initiated, (2) no new transactions are scheduled to be initiated, (3) an ongoing transaction is currently not served, and (4) no transactions of other components have been or will be initiated that require service from the component supposed to be updated. The *quiescent* state of a component is rarely reached naturally, but requires the restriction of the system's components. Since collaboration and binding relations between components can be arbitrarily complex, a possibly large set of components has to reach a *passive* state in order for one component to reach a *quiescent* state. The reason is condition 4, which requires all components not to initiate a transaction that requires the participation of the component supposed to be updated. Consequently, not only components that are directly connected to the component intended to be adapted, have to be passive, but all indirectly connected components that would require participation are prohibited to initiate new transactions. All those directly or indirectly connected components have therefore to be kept in a passive state, which requires additional means to restrict the system's behavior. Such means, though, are not part of the *quiescence* concept and, thus, not further discussed in the approach.

Tranquility

Vandewoude et al. [45] pointed out that the *quiescence* approach causes heavy system interruption because not only the component that is supposed to be modified has to reach a quiescent state but also all peer nodes have to be in such a state, because they are not allowed to initiate new transactions that involve the component under adaptation. They proposed a less strict approach called *tranquility* [45] and proofed it to be a sufficient condition for adaptability. A component is in a *tranquil* state if all connected components reach a passive state whereas *quiescence* required all directly as well as indirectly connected components to reach the passive state. The tranquility concept does not guarantee a safe state to be reached within a bounded timeframe to perform the ac-

tual update, which can be considered a major drawback of the approach [45]. However, the interruption of the running system is less severe using the tranquility approach than quiescence.

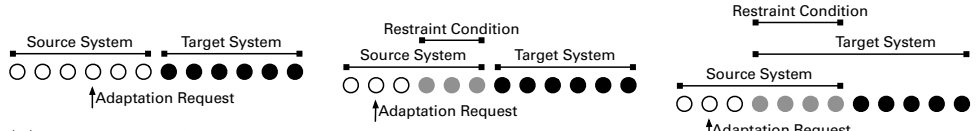
2.1.2. Semantics of Adaptation – Adaptation Models

The available adaptation operators to change the self-adaptive software system differ depending on the granularity of the adaptive entities in the system, e.g., different operators to perform changes on the level of services, components or objects are required. An adaptation describes the transition from one system (*source system*) to another system (*target system*) [7]. Zhang et al. [7] specified formal semantics for three different kinds of adaptations, the *One-Point Adaptation*, the *Guided Adaptation* and the *Overlap Adaptation*, which are displayed in Figure 2.3.

One-Point Adaptation The application adapts to the target system after receiving an adaptation request at a certain point during the execution. Reaching a safe state during the program's execution is a necessary prerequisite of this adaptation semantic to perform the adaptation.

Guided Adaptation In contrast to the *one-point adaptation*, a fundamental assumption of the *guided adaptation* is that the program is unable to reach a safe application state within a bounded timeframe. After receiving an adaptation request, the program's source functionality and behavior are therefore limited to ensure a safe state to be reached. The adaptation is performed immediately when a safe state is reached causing the program's functionality and behavior to change from one execution point to another.

Overlap Adaptation The *overlap adaptation* shares the same assumptions about the reachability of a safe application state as the *guided adaptation*. A further commonality is the program's source behavior to be restricted to ensure a safe state to be reached within a given amount of time in order to perform the adaptation. In contrast to the aforementioned *guided adaptation*, however, the program's target behavior starts to execute before the program's source behavior is fully stopped. Consequently, both source and target behavior coexist for a given timeframe (cf. Figure 2.3c). To prevent inconsistent application behavior, a restriction condition is applied on the program during the overlap time of both behaviors to safeguard the adaptation. The restriction condition is also used to restrict the source program's functionality and behavior to ensure the source program to eventually stop executing.



(a) One-Point Adaptation (b) Guided Adaptation (c) Overlap Adaptation

Figure 2.3.: Formal adaptation semantics according to [7].

A *safe state* [7] is described as a state in which the obligations of the source system are fulfilled and the source behavior can be terminated. Terminating the source behavior without having reached a safe state is prone to result in inconsistent behavior or data loss because ongoing computations of the source system could not be finished when terminated.

In a subsequent paper, Zhang et al. extended their approach to a model-driven approach to formalize adaptations in a software system and therefore described the adaptation as a transition from a *source model* to a *target model* using a specific *adaptation model* that comprises several operators that describe the transition [53].

A distributed coordination protocol to ensure a transactional adaptation behavior as it is envisioned in this thesis follows the semantics of the *guided adaptation*. Since our proposed coordination protocol treats the adaptable role-based application as a black box, we cannot derive a suitable point in time to alter the application without restricting the functionality of the role-based application. Logically, our approach cannot be a *one-point adaptation*. This observation furthermore holds for every adaptation operator we propose within this thesis.

2.2. Decentralization in (Self-)Adaptive Software Systems

self-adaptive software systems expose the desired properties of a software system that we described as a small collection of use-case examples in the previous chapter. Recent research surveys already summarize the current state of the art in the domain of self-adaptive software systems. In the 2009 [38] and 2010 [29] surveys, the research landscape of self-adaptive software systems was well outlined and important future research challenges were identified and discussed. The decentralization of the adaptation management was identified as one of the major concerns for future research efforts. A recent study conducted by Krupitzer et al. [27] focused on engineering approaches of self-adaptive software systems and a classification was developed that extends an original classification of self-adaptive software system approaches presented in [38]. Another recent survey fo-

cused on the application of self-adaptive software systems in the domain of cyber-physical systems [34]. The survey is concerned with the aspects how self-adaptation is applied to cyber-physical systems including adaptation concerns, evaluations and (dis-)advantages of surveyed approaches in the domain of cyber-physical systems.

In the remainder of this section, we will give a brief overview on the general architecture of self-adaptive software systems and how such systems allow for adaptive behavior. Subsequently, we will focus on decentralization aspects specific to the execution of adaptations. Other research results that contribute to other phases of the feedback loop are out of scope and will therefore not be discussed within this thesis. The relation of the *Role* concept to self-adaptive software systems and related fields that enable software systems to expose dynamic and adaptive behavior, will be covered in the subsequent section. Concrete approaches how adaptations are executed at run time will be discussed and compared to the set of requirements we posed on the envisioned decentralized execution phase afterwards. Finally, in the summary we will review the results of this section and briefly outline gaps in the existing research body that will be then addressed within this thesis.

2.2.1. General Architecture of (Self-)Adaptive Software Systems

In Chapter 1.3.1 we already introduced the two main subsystems of self-adaptive software systems: the *managed application* providing all the application's business functionality and the *adaptation management* that is responsible for the adaptation of the managed application at run time. If both subsystems communicate with each other through a well defined interface and are otherwise independent of each other, the architecture of the self-adaptive software system is usually considered to follow an *external control* approach, which is coined by both adaptation management and managed application being independent subsystems exchanging information through well-defined interfaces [38].

The *adaptation management* is not only responsible for the adaptation of the managed application (self-adaptation), but also for the monitoring of the computational environment of the system (context-awareness) and the monitoring of the internal state of the managed application (self-awareness). Moreover, the adaptation management implements mechanisms that enable it to abstract from and reason about the monitored data, which enables the adaptation management to detect when adaptations are necessary to be performed. A fourth task of the adaptation management is the calculation of adaptation plans, which describe the parts of the managed application that have to be adapted, added or removed in response to changes in the computational environment. This sequence of steps is often realized with the help of a feedback loop known from

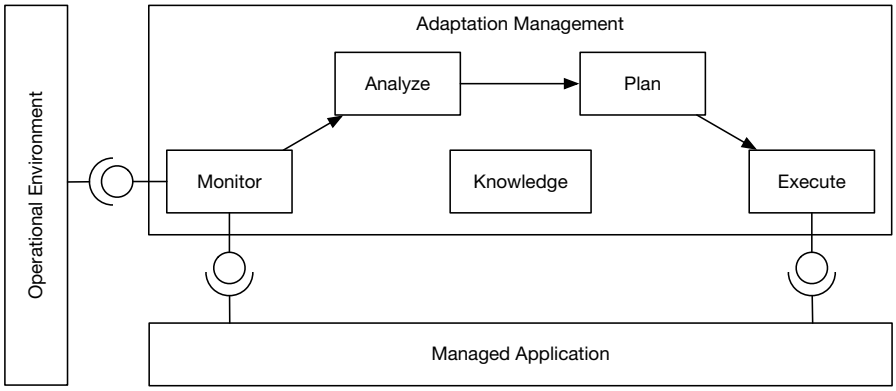


Figure 2.4.: A self-adaptive software system with the feedback loop as part of the adaptation management based on [23].

control theory. The feedback loop presented by Kephart et al. [23] is one typical example for such a software feedback loop to support context-awareness, self-awareness and self-adaptation in self-adaptive software systems. It consists of four phases to *Monitor* the computational environment of the system and the internal state of the managed application, to *Analyze* the monitored information and reason about it in order to be able to derive a *Plan* how to adapt the managed application, and to *Execute* the derived plan upon the managed application, finally. An overview of the feedback loop as a part of an externally controlled self-adaptive software system is given in Figure 2.4. Other feedback loops such as [11] and [35] exist, but provide basically the same set of features as the just discussed MAPE-feedback loop proposed by Kephart et al. [23]. One of the major differences of the feedback loop proposed by Oreizy et al. [35] is the notion of *consistency* that is explicitly introduced as one of the tasks of the feedback loop. The feedback loop of Oreizy et al. consists basically of two major parts: the *adaptation management*[1] and the *evolution management*. The evolution management is concerned with the execution of adaptations at run time and is explicitly deemed responsible to execute the adaptation process in a way that the desired run-time model of the application matches its configuration [35].

The reification of the feedback loop as a set of interconnected subsystems within the adaptation management is a commonly used approach to design and implement the management subsystem of self-adaptive software systems. A general example of such an architectural approach featuring an *external control* design is given in Figure 2.4. The Monitoring phase possesses interfaces to both the managed application and the

[1] Please note that the definition of *adaptation management* as a part of the feedback loop differs from the usage of the term within this thesis in general.

computational environment of the self-adaptive software system as a whole whereas the Execution phase shares an interface with the managed application only to realize the *self-adaptive* behavior. The *Knowledge* phase, which is also depicted in Figure 2.4, has not yet been discussed. It is not a phase in the sense of the already known MAPE-phases but serves rather as a shared knowledge base for all MAPE-phases. Run-time models, context-models, predefined adaptation plans etc. are typical examples for information and data structures stored in the *Knowledge* phase of the feedback loop accessible from all phases around it.

The subsystems of the adaptation management, however, are not necessarily aligned to the previously discussed phases of the feedback loop. Several approaches merge related parts of the feedback loop into a single subsystem of the adaptation management. Approaches that, for example, implement the self-awareness and self-adaptation properties, i.e., the internal state of the managed application is monitored and the managed application is adapted at run time, often merge the monitoring and execution phase into a single layer or subsystem of the adaptation management because both the monitoring and the execution require an interface to the managed application that enables both self-properties. Approaches following this track are Rainbow [13], 3L [25], SASSY [16], GRAF [10] or SMAGs [36], for example. Merging the monitoring and analyzing phase of the feedback loop into one layer or subsystem is often used when the computational environment is monitored (context-awareness) and requires a certain degree of abstraction from the raw context to determine situation changes, e.g., Hallsteinsen [18]. Approaches such as Rainbow [13] or GRAF [10] merge the analyzing and planning phase into a single layer, since the generation of change plans often relies on the kind of context change that is usually determined during the analyzing phase of the feedback loop. An overview how the just discussed approaches reified the MAPE feedback loop in their respective architectural approaches is shown in Figure 2.5.

2.2.2. Decentralization of the Adaptation Management

Many self-adaptive software systems implement the managed application using components or services as adaptive software entity. Naturally, service or component-based architectures can be distributed across several devices. With a growing distribution of the managed application, however, a centralized management of the entire adaptation process becomes increasingly difficult. Consequently, the decentralization of the adaptation management is an important aspect of self-adaptive software systems to cope with an increasing distribution, both logically and spatially, of the managed application.

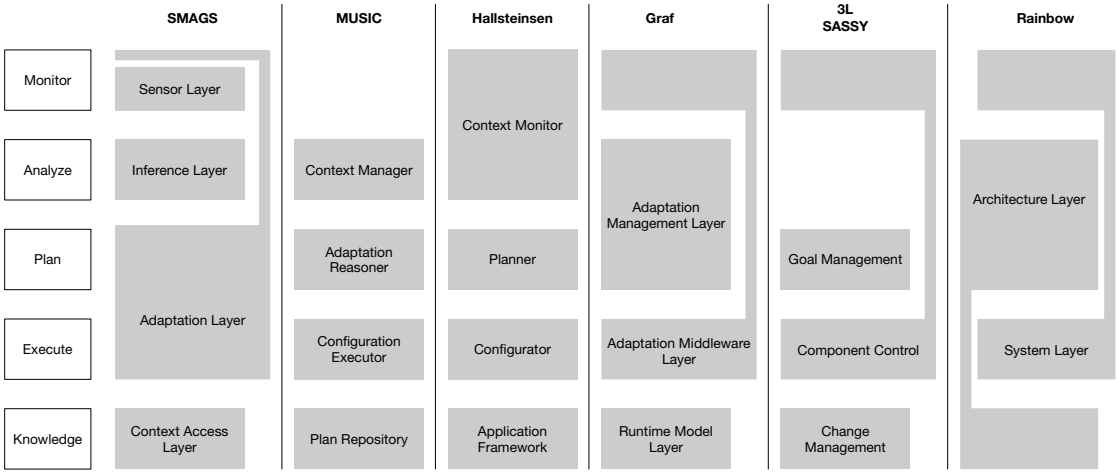


Figure 2.5.: Overview how different SASS approaches reify the MAPE feedback loop in their respective architectural approach.

Decentralization Patterns [51]

Weyns et al. proposed a set of different patterns to distribute and decentralize the feedback loop of the adaptation management. A phase of the feedback loop is distributed if multiple instances of it exist in the running system whereas a phase of the feedback loop is decentralized only if these instances also communicate and collaborate to perform their task. The definitions of distribution and decentralization by Weyns et al. have already been discussed in detail in Chapter 1.3.1.

Weyns et al. suggest a total of five different patterns to distribute and decentralize the adaptation management's feedback loop. We call four of those approaches hybrid patterns because only a subset of the feedback loop's phases is distributed or decentralized. The remaining pattern is called the *coordination control pattern* and is coined by a decentralization of all phases of the feedback loop. The first four mentioned patterns address different problems each: A decentralized context acquisition is addressed by the *information sharing pattern* in which only the monitoring phase is decentralized. The decision making process can be clustered using the *regional planning pattern* in which multiple collaborating planning phases exist. In cases where a centralized context analysis and generation of change plans is beneficial, the *master/slave pattern*, which moves the monitoring and execution tasks to the edge devices where the monitoring takes place or the adaptations are performed, can be employed. In large systems, the responsibilities to make decisions or the abstraction of the information to base adaptation decisions on may be hierarchically structured to cope with the system's complexity. The *hierarchical control pattern* addresses this issue of structuring adaptation cycles in different MAPE

control loops that operate on different levels of abstraction within the system.

Only the *coordinated control pattern* suggests the requirement for a decentralized coordination of the execution process. Although less evident, the concept can be applied most easily to the *master/slave pattern*, too. Having the monitoring decentralized is out of scope of this thesis, but even slave devices in a self-adaptive software system may communicate and collaborate with each other to perform the tasks assigned to the managed application deployed on them. Thus, the distributed execution phases of this feedback loop might be required to collaborate in order to coordinate the execution of the change plan generated by the central planning phase.

Interacting Control Loops [46]

Vromant et al. describe a self-healing[2] scenario for a traffic monitoring system that makes use of intra- and inter-control loop coordination to realize the self-healing property of the system. In the described scenario, a traffic monitoring system is clustered into groups of traffic monitoring cameras in which each cluster is comprised of a dedicated *master* camera and an arbitrary number of *slave* cameras. The system is supposed to recover autonomously from the random failure of a master camera.

In their approach, Vromant et al. describe two types of coordination: The first one is the *intra-loop coordination* that enables "MAPE computations within one loop to coordinate with one another." [46] In that case, sub-loops are created within a given feedback cycle to coordinate adaptation tasks. Such a sub-loop may contain only a subset of the phases of the original MAPE feedback loop, e.g., only the planning and execution might be required within a sub-loop to perform the coordination task. The second type of coordination is the *inter-loop coordination* that enables "MAPE computations across multiple loops to coordinate with one another." [46] This control type allows phases of different feedback loops to interact in order to perform their adaptation task, e.g., the planning components collaborate to achieve a joint agreement on an adaptation plan.

The failure of a master camera in the approach described in [46] is handled with a sequence of intra- and inter-loop coordinations. A first intra-loop consists only of the monitoring and analyzing phase and detects the absence of the failed master camera. In response to a master camera's failure, a first intra-loop is created on each camera device that changes the communication flow in a way that no information is sent unnecessarily any more to the failed camera. Subsequently, a new instance of an intra-loop comprised of a planning and execution component of the feedback loop is created on each camera

[2] A self-healing system has the "capability of discovering, diagnosing, and reacting to disruptions. It can also anticipate potential problems, and accordingly take proper actions to prevent failure." [38]

device. Within this intra-loop an inter-loop coordination is set up to elect a new master node from within the group of slave nodes of the respective camera cluster. In further steps of intra- and inter-loop coordinations, the system is brought back to a consistent state.

Our approach is neither concerned with the placement of phases of the feedback loop nor with the interaction of such phases in general. With respect to the proposed coordination types, our approach, however, would be perfectly applicable to inter-loop coordinations established between the execution phases of different control loops to handle the execution of adaptation steps. A tight interaction with the planning phase in separate local intra-loops coordinating with other intra-loops on remote devices could be used to react swiftly to adaptation errors that are severe enough not to be addressable by our coordination approach and thus would lead to a failure of the whole adaptation process. With the approach Vromant et al. proposed, such a scenario could probably be handled more gracefully with the aim to prevent the abortion or failure of the whole adaptation.

2.2.3. Decentralization of the Execution Phase

The software reconfiguration patterns proposed by Gomaa et al. [15] rest on component-based architectures. The work aims to determine the updatability of software components without central knowledge of the components' internal state and their participation in ongoing communications. Each component therefore implements a state chart based on the quiescence criteria, which is depicted in Figure 2.2. Different patterns for different application scenarios exist. In the decentralized control patterns proposed by Gomaa et al. a component [15] communicates with its predecessor and successor to determine the state of the component itself with respect to the quiescence criteria [26]. If a component is in a quiescent state, the adaptation of that component is deemed possible and adaptations can be executed.

The software adaptation patterns proposed by Gomaa et al. [16] are based on service-oriented architectures. The work aims to allow a service to be updated transparently to the other services of the architecture. Services in the system therefore communicate through *connectors*, which are comparable to proxies, with each other. The proxy implements the quiescence state chart presented in Figure 2.2 to determine when the service the proxy belongs to is in a state to be updated. Different connectors depending on the service's role in the architecture exist. As soon as a proxy determines its managed service to be in a quiescent state, the update of this service is allowed and can be conducted.

In none of their works do Gomaa et al. describe how adaptations are actually imple-

mented or coordinated. It is also not mentioned, which kinds of adaptations are possible. Since no notion of depending adaptations exists in either of the proposed approaches, it is save to assume adaptations to be local to the component or service, respectively. Both approaches focus on a way to determine a quiescent state for a component or service, respectively, to determine save application states to perform the actual adaptation.

The approach by Georgiadis et al. [14] intends to change the configuration of a component-based software system at run time. A component in the system comprises the component's application-specific implementation, a management agent that performs the adaptation tasks of its managed component and a view representing the system's current configuration. All component managers communicate via a reliable broadcasting mechanism with each other. The communication protocol basically comprises two types of messages: Join/Leave messages are used to advertise new components within the network or to notify peers about the graceful disappearance of components whereas Lock messages are used to satisfy a distributed locking scheme. This message type is especially important with respect to the run-time modification of the component-based software system. Only one component manager is allowed to modify the view of the system at a time, thus, the locking scheme assures no adaptation manager to be allowed to modify a view that has already been invalidated by another adaptation manager.

The approach by Georgiadis et al. [14], however, only allows the reconfiguration, i.e., rebinding, of the managed component's provided and required ports. Other operations, such as the explicit creation or deletion of component instances is not supported by the protocol.

In summary, existing approaches to distribute and decentralize the execution phase of the adaptation process lack means to coordinate several adaptations performed in parallel or in a (at least partially) sequential order. Existing solutions to determine a stable application state in a distributed, managed application, especially [16], are a good starting point for an integration in our work and an extension to meet our imposed requirements on the envisioned solution.

2.3. Roles in Adaptive Software Systems

The concept of roles has already been applied to the design and implementation of self-adaptive software systems or software systems in general. However, existing approaches only cover certain aspects of the *Role* concept, e.g., their collaborative and behavioral nature (MACODO), their contextual and behavioral nature (HELENA) or mainly their collaborative nature as in the field of Multi-Agent Systems. This section investigates

the utilization of the *Role* concept in the existing research body and compares the existing notions of roles with our understanding of the concept, highlighting differences and similarities.

2.3.1. MACODO [17]

A large software system is often comprised of many smaller subsystems that exchange information amongst each other to perform their assigned tasks. Specific names are often assigned to these subsystems, e.g., Message Broker, Provider, Booking System etc. These names are basically *Roles* the respective subsystem plays in the overall system and, most importantly, in relation to other subsystems with respect to their mutual interaction. The respective responsibilities and "Roles" of interacting subsystems named Provider, Message Broker and Receiver are intuitively understandable.

Modeling these collaborations between subsystems on an architectural level, however, is a challenging task that the *MACODO* approach tackles. The approach aims to make the collaborations among subsystems explicit on an architectural level to better capture the system behavior that is a result of these collaborations. Therefore, *MACODO* proposes different architectural views to capture the collaborative system behavior utilizing the notion of *Roles* to describe responsibilities of subsystems and the interactions between them. The *Collaboration & Actor View* models the actors (i.e., *Players*) in the system and the concrete collaborations between the actors. This view describes the run-time architecture of a system in terms of actors as well as the collaborations between them and assigns responsibilities to actors omitting unnecessary information about the details of the collaboration. The *Collaboration View* models collaborations into reusable units and decomposes them into reusable subunits. This view describes collaborations in terms of implementation units and captures the type of collaborations, roles, behaviors and interactions. The *Role & Interaction View* models the run-time architecture of collaborations in detail.

The notion of the *Role* concept in *MACODO* is organization-centric [9] because roles are treated mainly as named places with players implementing the role-specific behavior through the adoption of the interface demanded by the role of the collaboration. In *MACODO*, the system behavior varies within the collaboration, i.e., the exchanging actors participating in a collaboration can alter the overall system's behavior. This variability, especially at run time, however, has not been closely investigated in *MACODO* since the approach focuses on the architecture of collaborative software systems at design time.

In summary, *MACODO* provides an interesting approach to capture collaborations in large and distributed software systems, but omits run-time aspects of behavioral adapta-

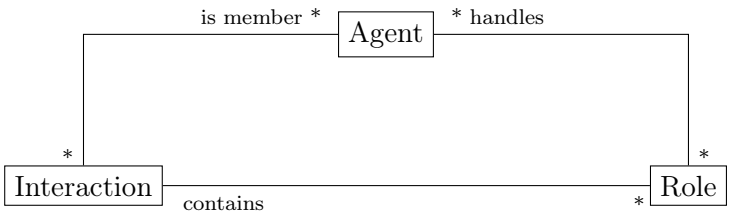


Figure 2.6.: General role model used in multi-agent-systems.

tions. Our approach, in contrast, focuses on the utilization of the *Role* concept to foster reliable run-time adaptations of highly distributed and interconnected software systems.

2.3.2. HELENA [24, 19]

The *HELENA* approach originally proposed a holistic engineering process for self-adaptive software systems using the notion of *Roles* to improve the modeling of standard and dynamic system behavior. *HELENA* follows an extreme player-centric [9] approach, which means that players are without specific behaviors of their own but serve as mere containers for the roles they are playing to exchange data. Any dynamic system behavior is implemented in the role, thus, the behavior of a player changes depending on the set of roles it currently plays. One of the major contributions of the approach is the description of run-time adaptations using formal methods to ensure system properties such as correctness, i.e., if adaptations will result in consistent and valid system configurations.

In summary, *HELENA* mainly focuses on the design of self-adaptive software systems using the notion of *Roles* to abstract from standard and dynamic system behavior. A formal method was presented to describe changes in the software system and to reason about these changes, but concrete solutions how adaptations are performed at run time that ensure a consistent transition from the system's source to its intended target configuration are missing.

2.3.3. Roles in Multi-Agent-Systems

Due to the large variety of Multi-Agent-Systems (MAS) and their diverse fields of applicability, we focus on the commonalities among several approaches with respect to their utilization and application of the *Role* concept. In Figure 2.6, a generalized role model applicable to many multi-agent-systems is depicted. Using an organization-centric [9] notion of roles is common in MAS as it is used in [12, 3, 4], for example. Agents communicate or collaborate through a defined interface that denotes their responsibilities (e.g., role) in the collaboration. A collaboration is generally not limited to a one-to-one

communication between two agents but allows several agents to participate. Due to the organization-centric notion of roles in multi-agent-systems, the system behavior is varied within the collaboration. Behavioral modifications of the system within a collaboration can be achieved by different players implementing the required interface or responsibilities of the role differently. A major drawback of the notion of roles in multi-agent-systems is the inability of players, i.e., agents, to pick up new roles at run time easily.

2.4. Perform Run-Time Adaptations

In the previous sections of this chapter, we discussed the general architecture of self-adaptive software systems, formal specifications of stable states of self-adaptive software systems to safely perform adaptations, and role-based approaches in the domain of self-adaptive software systems. An important aspect of self-adaptive software systems, however, has not been subject to a thorough discussion yet: how adaptations are actually carried out at run time and how the adaptations that are supposed to be executed are described. We will fill this gap in the remainder of this section.

Changes performed by a self-adaptive software system on itself during run time can generally be classified as *parameter adaptations* or *structural adaptations* [38]. The class of *parameter adaptations* usually only modifies publicly available variables of adaptive software entities in order to change their internal behavior. The setParameter operation of the 3L approach [25] serves as an example for parameter adaptations within component-based self-adaptive software systems.

The class of *structural adaptations* of self-adaptive software systems is the more powerful class of adaptations because the structure of the entire system or large parts of its behavior can be modified. In the case of implementing the adaptive entity on the level of software components or services, the modification of the binding of those components or services is an evident structural adaptation. The modification of binding information, however, is not a very powerful structural adaptation. Approaches such as 3L [25] or SASSY [16], for example, allow the creation and removal of adaptive software entities dynamically at run time. Such operations, for example, would allow the system to react to changing system workloads by creating or removing adaptive software entities in response. Together with the dynamic binding of these entities, it is possible to integrate new adaptive software entities easily into the existing infrastructure or to remove existing entities from the infrastructure if the respective behavior is not required any longer. The approach proposed by Hallsteinsen et al. [18] also suggests higher level adaptation operations such as the replacement of adaptive software entities with another one or

Table 2.1.: Supported component- & service-level adaptation operations of selected SASS approaches.

	SMAGS [36]	MUSIC [37]	Hallsteinsen [18]	Graf [10]	3L [25]	SASSY [16]	Rainbow [13]
Create	?	?	✓	–	✓	✓	–
Remove	?	?	✓	–	✓	✓	–
Connect	✓	✓	✓	–	✓	✓	–
Disconnect	✓	✓	✓	–	✓	✓	–
Higher Level	?	✓	✓	◆	–	–	–

Fully Supported: ✓ | Partially Supported: ◆ | Not Supported: – | Not Mentioned: ?

the relocation of a single adaptive software entity. From a design perspective, the sole creation and removal of entities lacks the ability to ensure the transfer of state information from the source to the target configuration, which would be allowed by higher level adaptation operations that were just mentioned. An overview of the supported adaptation operations is shown in Table 2.1. Approaches, such as Graf [10] or Rainbow [13] which are also displayed in Table 2.1, support operations different from the others. Rainbow relies on architecture-specific adaptation strategies, which cannot be mapped to any component- or service-level adaptation operations. Graf is one of the few approaches that relies on Java applications and the JVM to perform its adaptation tasks and uses Bytecode rewriting at run time to modify the managed application.

The execution of adaptation operations has not yet been closely investigated with the decentralization of the adaptation process in mind. Although some approaches, e.g., [16], determine a safe application state at which adaptations can be performed without a central control unit, the quiescence criteria [26] is applied only locally to a single adaptive software entity. In general, the proposed approaches do not support either requirement 1 or 2 or do support them only partially, e.g., SASSY [16]. A concise comparison of the previously discussed approaches with respect to our posed requirements on a self-adaptive software system is given in Table 2.2. Requirement 3 is evidently not applicable to approaches using a centralized adaptation management to execute adaptations of the managed applications. Other approaches, such as SASSY [16] or the work of Georgiadis et al. [14] and the later 3L approach [25] assume a reliable message channel between all participating computing devices in the network, which limits the applicability of their

Table 2.2.: Comparison of SASS approaches with respect to the imposed requirements on a self-adaptive software system.

#	Requirement	SMAGS [36]	MUSIC [37]	Hallsteinsen [18]	Graf [10]	3L [25]	SASSY [16]	Rainbow [13]
1	Decentralized Execution	–	–	–	–	◆	◆	–
2	Stable State before Adaptation	–	◆	✓	–	◆	◆	–
3	Loss of Coordination Messages	–	–	–	–	–	–	–
4	Adaptation Failures	–	–	–	–	–	–	–

Fully Supported: ✓ | Partially Supported: ◆ | Not Supported: –

approaches to environments that can offer such a stable network link. None of the surveyed approaches takes the possibility into account that ongoing adaptation may fail or assume this not to happen at all at run time. Consequently, Requirement 4 is not being addressed by the current state-of-the-art research approaches. Failed adaptations might be detected in the next cycle of the feedback loop and then corrected, but we argue that it is beneficial to handle possible errors within the execution phase to maintain a consistent system configuration at all points in time.

2.5. Summary

In this chapter, the research landscape of self-adaptive software systems with special focus on the execution phase of the adaptation management has been discussed. The discussion of the research landscape was aligned to the requirements and research questions introduced in the previous chapter. We first discussed formal approaches to describe a state in a self-adaptive software system that is considered safe to perform adaptation without inconsistent system behavior, e.g., loss of application data. Subsequently, we discussed how and to what degree the adaptation management of state-of-the-art research approaches is decentralized. In this discussion, we focused specifically on the execution phase of the adaptation management's feedback loop since this is the major field of interest of this thesis. In a third section, it has been analyzed how the *Role* concept is related to existing work in the field of self-adaptive and multi-agent-systems. Finally, the execution of adaptations at run time in state-of-the-art approaches has been outlined. The focus of the last discussion point was set to decentralization aspects, too.

In summary, the execution of adaptation groups, which are adaptations that depend on each other, has not been widely considered in the current research. This holds especially true in approaches coined by a decentralized execution phase of the adaptation management. Approaches that focus on the decentralized execution of adaptation operations concentrated on determining a quiescent state for the adaptive software entity under update without a central coordinator. The execution of several adaptation operations at a point in time is possible, however, it is not possible to express dependencies between these adaptations. Consequently, the first requirement ("*Decentralized Execution*") can only be considered partially fulfilled at best by the existing research landscape. The second requirement ("*Stable State before Adaptation*"), in contrast, can be considered as generally met. Existing work by Kramer & Magee [26] and Vandewoude et al. [45] describes a state in a potentially distributed application that allows the execution of adaptations in a safe way, e.g., preventing inconsistent system behavior or loss of application data. Cheng et al. [7] describe formal methods to reach the aforementioned state. The proposed adaptation [16] and reconfiguration [15] patterns by Gomaa et al. use the aforementioned definitions and methods to reach such a safe application state. The results are in general immediately reusable to our envisioned decentralized coordination approach for highly distributed self-adaptive software systems. Requirements three and four are generally not addressed by the current research landscape (Requirement 4: "*Adaptation Failures*") or assumed not to occur (Requirement 3: "*Loss of Coordination Messages*") because a reliable channel is assumed to be always available.

Multi-agent-systems have already adapted a notion of roles to describe collaborations between the agents of the software system. The organization-centric [9] point of view that most prominently relies on the collaborative nature of roles, is dominating in multi-agent-systems. Roles in organization-centric approaches are just named places to describe system functionality, which is actually implemented by the players of the respective role. This organization-centric view on roles, however, is not sufficient to describe and address all the application scenarios that were introduced in the previous chapter. The rescue-robots scenario, for example, relies greatly on the context-dependent and behavioral nature of roles and less on the collaborative nature of them.

Self-adaptive software systems introduce roles on the design and architectural level of the system to capture collaboration-specific features of the system (MACODO [17]) or to specify adaptations formally at design time (HELENA [24, 19]). The question how roles can contribute to the decentralized execution of multiple depending adaptation operations remains unanswered by these approaches and will be tackled in the following chapters of this thesis.

3. Employing Roles in Decentralized Self-Adaptive Software Systems

Self-adaptive software systems are usually coined by two different aspects of system behavior: static system behavior provides the basic functionality of the software system whereas dynamic system behavior can be added to or removed from the system context-dependently. Software abstractions, such as programming-level run-time objects [10], components [25] or services [33], do not cover the differences between dynamic and static system behavior explicitly, but use the same abstraction for both system parts.

The *Role* concept is utilized as fundamental abstraction within this thesis to capture static and context-dependent concerns of the *managed application*. As outlined in Chapter 1.3.2, a *Role* can be described through behavioral, collaborative and context-dependent natures and is always played by a *Player*, which represents a static part of the system. We make use of all three natures of this concept to fully incorporate the advantage of role-based abstractions for the execution of changes in self-adaptive software systems.

Kühn et al. [28] pointed out that no unified role model exists, but different approaches rather rely on different notions of a role. As a consequence, they developed a meta-model family from which the role model used within this thesis will be derived. In this chapter, this said role model is being developed. In a first step, we focus on the *Role Runtime* providing the execution environment for role-based applications that implement the self-adaptation property of the self-adaptive software system addressed within this thesis. Subsequently, we focus on concrete concepts required from such a role runtime to aid the distributed and decentralized adaptation of distributed role-based applications. Such mechanisms incorporate a dedicated life cycle for the run-time representation of a *Role* as well as a publicly accessible interface for the adaptation management to modify the role-based application. Finally, we compare the features required in the developed approach with the list of all role features developed by Kühn et al. [28] and with other approaches in the domain of self-adaptive software systems utilizing the *Role* concept.

3.1. An Adaptation Supportive Role Runtime

The notion of *Roles* is used as an abstraction to design and implement the *managed application* to explicitly distinguish between dynamic and static application behavior. In Figure 3.1, the relation of role-based application, role runtime and execution management is depicted. The managed application is represented by the *Role-based Application* and is distributed across several devices. Consequently, the managed application implements the dynamic system behavior using the *Role* concept and complies with the role model, which will be developed in the course of this chapter and summarized in Chapter 3.3. Every local application artifact of the distributed role-based application is embedded in a *Role Runtime*, which serves as execution environment for the role-based application and provides an interface to the *Execution Management* of the self-adaptive software system through which the role-based application can be modified at run time. The *Execution Management* refers to the execution phase of the previously discussed feedback loop. Other phases of the adaptation management, i.e., the monitor, analyze and plan phase of the feedback loop, are not important at this point and neither is their possible distribution across different devices.

The *plays* relation between players and roles is indicated with solid arrows in Figure 3.1 and represents the run-time structure, which will also be referred to as configuration, of the local role-based application. Hence, the modification of the relation between players and roles represents a context-dependent structural adaptation of the application. In order to be able to modify these bindings, but also to expose the correct system behavior when multiple roles are bound to a player, the role runtime has to maintain a run-time model of the device-local part of the managed application. This run-time model is furthermore expected to be retrievable and modifiable by the execution management through a well-defined interface, which we will discuss later in this chapter. Keeping such an externally retrievable run-time model is the first requirement we pose on the role runtime.

Besides the context-dependent nature expressed by the plays relation between roles and players, roles in our approach also possess a collaborative nature, which represents the interaction among roles. Collaborating roles may be located on different nodes, though. The local role runtime is required to maintain collaboration information of every roles' collaborations both locally and remotely. As an additional feature, each local role runtime does not only have to maintain a record about each role's respective collaborations but also requires a means to determine when a response for any given remote invocation of a role's method has been received. Generally speaking, the role runtime has to be able

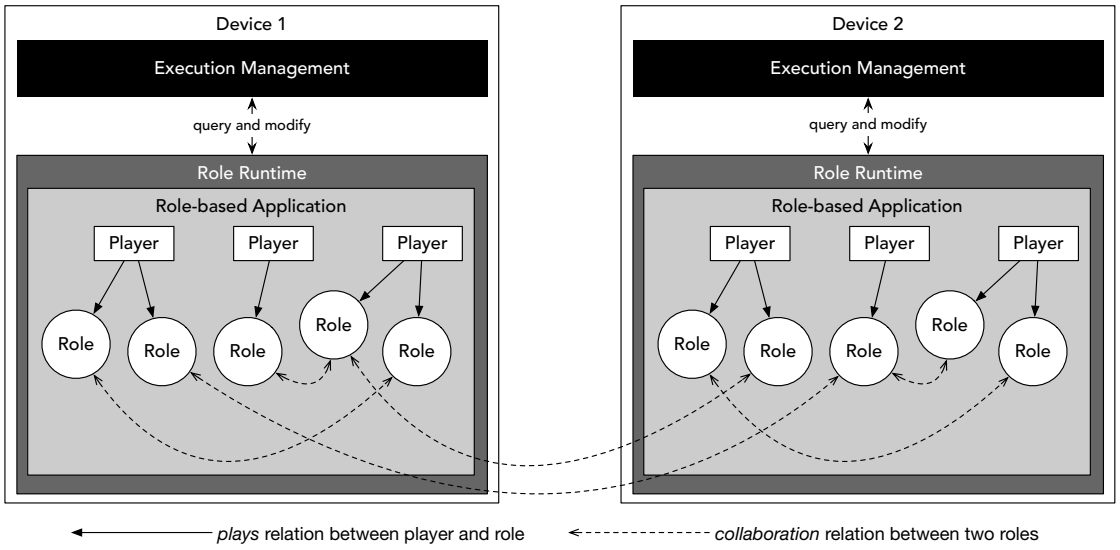


Figure 3.1.: Assumed role model of the role-based application at run time.

to determine when interactions among roles have finished. This information is essential in order to determine a stable application state for the distributed role-based application, which we will discuss in this chapter from the role-based application's perspective and in the next chapter from the decentralized adaptation's perspective. In Figure 3.1, collaborations between roles are indicated as dotted arrows representing both local and remote collaborations of roles. Compartments, which are not depicted in Figure 3.1, are used to capture application-intrinsic context-information, e.g., to group collaborating roles in a common scope that can aid the dispatch of method invocations or to aid expressing different concerns or layers of an application. In the remainder of this work, therefore, a player always plays a role within the scope of a compartment.

With respect to this run-time representation of roles and players, it is important to highlight that roles and players are perceived as indistinguishable from the application's perspective although maintained as distinct instances by the role runtime. Roles and players, therefore, can be considered as one entity when they are bound together from the application's perspective. Since players are able to play multiple different roles, which might provide the same interface, simultaneously, a *Dispatcher* is required to resolve the receiver of a method invocation on the player. This dispatcher redirects the invocation on the player's interface to the correct role. Naturally, method invocations to roles located on remote devices are resolved by the dispatcher in a similar manner to local role invocations.

Relying on the principles and mechanisms the role runtime is based on and has to

provide, the role runtime's support for local adaptations in response to changes in the overall system's computational environment will be discussed in the following.

3.2. Supporting Decentralized Run-time Adaptation

The *Role Runtime* is an integral part of the run-time adaptation process of the software system. It maintains run-time information about available role types, the plays relation between players and roles as well as the collaborations among roles. With respect to the support of the run-time adaptation process, the life cycle of roles will be discussed first. Subsequently, the adaptation interface of the role runtime required to support local adaptations will be developed. Next, the process of reaching a stable application state for a single role instance within a local run-time system will be explained relying on the previously developed life cycle for roles. The developed process will be extended for distributed, role-based applications finally.

3.2.1. The Role Life Cycle

The role life cycle is part of the *run-time model* of the role-based managed application and describes the states a role in such a software system can be in. A life cycle for roles is important especially during the adaptation of the software system at run time with respect to the first research question, which addresses the issue of reaching a stable application state before the adaptation can be performed. The goal of the decentralized execution phase to maintain a consistent configuration of the managed application at all times during the adaptation process also requires a precise definition of a life cycle for roles. In [21], we described a life cycle for roles taking different layers of role-based applications into account. Within this section, we will focus on the *Application Layer* life cycle of roles and will elaborate on the life cycle states to cover all parts important for run-time adaptations.

Application scenarios one and two described the adaptation of a fully deployed and running software system. The dynamic parts of the software system in these two scenarios are fully known at run time. Application scenario three, in contrast, described the introduction of previously unknown dynamic system behavior, i.e., roles. The resulting life cycle for a role-based software system, therefore, has to cover both type and instance level of the *Role* concept. In the remainder, we will use the term *role* as a short-hand for *role instance* and refer to the type of a role explicitly when necessary. Similarly, the term *player* always refers to a concrete *player instance*. Since we allow roles to be bound to arbitrary player types and assume players to be always present at runtime as they

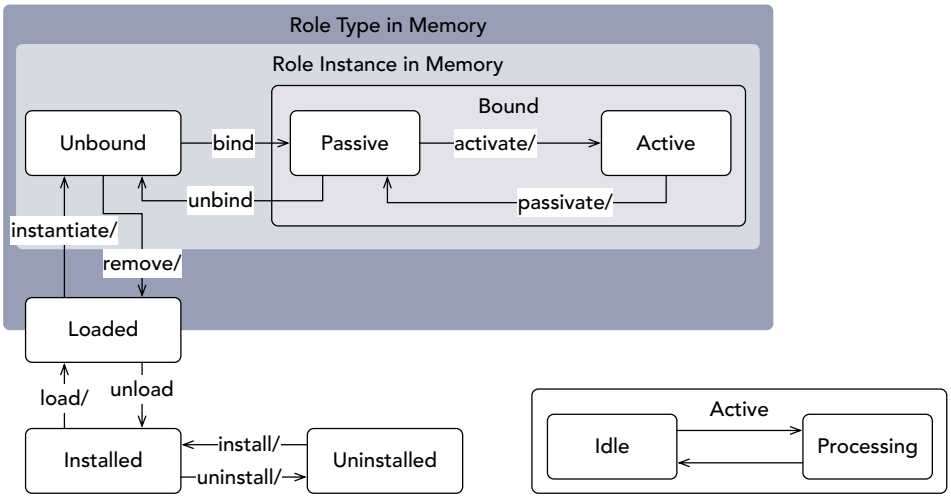


Figure 3.2.: Life cycle for the *Role* concept covering the type level (for evolution purposes) and the instance level (for adaptation purposes).

represent the static and context-independent parts of the software system, the type of any given player is not significant for the discussion of a role's life cycle.

In the life cycle, which is displayed in Figure 3.2, the states *Uninstalled*, *Installed* and *Loaded* are related to the maintenance of type information for roles and are motivated by application scenario three to introduce new or remove obsolete role types from the run-time system. An *Uninstalled* role type is not known to the respective node at all. Thus, using the *install* transition the type information is shipped to the nodes and installed on the respective node. Subsequently, role type information can be loaded to the run-time system and is then in the *Loaded* state. This state of a role type is at least required for our approach to perform run-time adaptations because a role type loaded to the role runtime can be used to create role instances via the adaptation interface, which will be discussed later in Chapter 3.2.2.

The most important part of the role life cycle is concerned with the creation and deletion of roles, their binding and unbinding process as well as their different execution states. In the following, all life-cycle states related to these concerns are briefly explained.

Unbound: Roles in an *Unbound* state exist as concrete instances of a specific role type within the local runtime. A role in such a state does not affect the behavior of the role-based application because behavioral adaptations are realized through roles being bound to players, which the respective role is not yet or not anymore. Without being bound to a player, a role can also not participate in collaborations with other roles. This state of a role can be considered a maintenance state, which

is required in the process of creating and destroying instances of roles.

Bound: A bound role does not only exist in memory but is also bound to a specific player. At this point, we want to highlight that a specific instance of a role can only be bound to exactly one player, but different instances of the same role type may be bound to different players. The *Bound* state of a role contains two sub-states: *Active* and *Passive*.

Active: A role in an *Active* state is considered by the dispatching mechanism of the role runtime for the behavioral modification of the player's behavior, i.e., methods implemented by the role are invoked instead of those provided by the player. If a role is currently serving any requests, i.e., a method invocation took place that is currently processed by the role instance and the method has not yet returned, the role is considered being in the *Active/Processing* state. If a role is active, but not serving any requests, it is in the *Active/Idle* state. Both last mentioned states are sub-states of the *Active* state and depicted as such in Figure 3.2.

Passive: A role in the *Passive* state is bound to a player, but not considered by the role runtime's method dispatching mechanism to alter the player's behavior. In this state, the role is consequently idle and does not perform any tasks. From an application perspective, the player appears as if the role would not be bound based on the player's exposed system behavior.

The transition of a given role through its life cycle shall be explained in the following with a scenario that evolves the managed application. A new encryption algorithm is supposed to be introduced to the smart energy broker example described as application scenario three (cf. Chapter 1.1.3). Before the update adaptation commences, the role is in an *Uninstalled* state. This state though, is merely virtual because the type information of the role is still missing in the local runtime, which is why no part of the system is able to maintain this state information. The first steps of the update adaptation introduce the type information to the local role runtime (*Installed*) and trigger the role information to be loaded into memory (*Loaded*). Conceptually, the execution management is then able to create new instances of this role type within the role runtime. The newly introduced encryption behavior is furthermore required to be bound to a given player within the system. An instance of the new encryption role type would be created and stored in the role runtime's memory. The role instance is therefore in the *Unbound* state. Immediately after the creation of the role instance the role will be bound to the respective player, thus entering the *Bound* state. In this state, the role is always in a *Passive* state and

remains in this state until it gets activated by the execution management. The execution management, however, needs to be able to ensure a consistent adaptation of the system if multiple adaptations are performed on the same or different devices, which is the reason for the role to be always in a passive state immediately after being bound. If the role was immediately active, an inconsistent configuration of the application can be reached in the case players collaborating through their roles with each other would expose different behavior, because one player has already bound the new encryption role while the other, for example, is still in the old configuration. As soon as the activation is issued, the role enters the *Active* state and is now able to perform its behavior. From this point in time, the role will constantly go back and forth between the *Active/Processing* and *Active/Idle* state depending on whether it currently performs computational tasks or not. The role instance can only be unbound and destroyed at a later point in time if it is in a passive state. All described transitions are depicted in Figure 3.2.

3.2.2. Adaptation Interface

The adaptation interface of the role runtime is a set of provided methods that allows the modification of the role-based application from a different process, e.g., the execution management of the self-adaptive software system as depicted in Figure 3.1. We will rely on this interface specification in the next chapter where we outline the overall contribution of this thesis, which addresses the execution of distributed adaptations of role-based software systems without a central coordinator. In the following, the methods provided within this interface to allow the modification of the role-based application by the execution management, will be introduced.

create(roleType)

The create operation allows the creation of a new role instance within the local role runtime. Newly created role instances are in the *Unbound* state. The create operation takes the type of the role to instantiate as sole input parameter and returns the identifier unique within the respective local role runtime of the newly created role instance.

remove(roleInstance)

The remove operation removes the given role instance from the local role runtime. If the role instance that is supposed to be removed is in the *Unbound* state, the removal of the role will be performed and a boolean value indicating the result will be returned. If the role is in any other state than the *Unbound* state, the operation will not remove the instance and return unsuccessfully.

bind(playerInstance, roleInstance, compartmentInstance)

The bind operation establishes a `plays` relationship between a given player and a role instance within a given compartment. The bind operation takes the unique instance identifiers of role, player and compartment as input and returns a boolean value that indicates whether the `plays` relation could be established successfully or not. If a role is already bound to a player and a bind operation is issued affecting this role instance, this method returns false without changing the underlying run-time model of the application, i.e., the `plays` relation between the role instance and its player is not modified. This constraint is a result of the assumption that a role instance can only be played by exactly one player at any given point in time. If a role is intended to be bound to the player it is already played by, the operation will succeed without changing the underlying run-time model. Consequently, the bind operation moves the given role from the *Unbound* state to the *Bound/Passive* state, thus, not modifying the player's exposed behavior yet, which is desirable if several adaptations have to be coordinated consistently, especially across multiple devices.

unbind(playerInstance, roleInstance, compartmentInstance)

The unbind operation is the inverse operation to the *Bind* operation and releases the `plays` relationship between a role and its player within the given compartment. The unbind operation takes the unique instance identifiers of role, player and compartment as input and returns a boolean value that indicates whether the `plays` relationship could be released successfully or not. This method will only succeed if the role is in the *Bound/Passive* state before the unbind operation is issued and thus moves the given role to the *Unbound* state.

activate(roleInstance)

A newly created role is in the *Bound/Passive* state, which means it does not modify the player's behavior since it is not considered by the role runtime to execute application-specific behavior. Roles that are in the *Bound/Passive* state are registered for the execution of application behavior by using the activate operation, which takes the unique instance identifier of the role instance to activate as sole input and returns a boolean value to indicate whether the activation was successful or not.

passivate(roleInstance)

The passivate operation is the inverse operation to the *Activate* operation and removes the given role instance from the list of roles considered by the runtime's

dispatching mechanism for the execution of application behavior of the respective player. It takes the unique instance identifier of the role to passivate as sole input and returns a boolean value to indicate whether the role could be moved to the *Bound/Passive* state or not. When the role to passivate is currently active and processing a request, i.e., if it is currently performing application behavior, the passivate operation will not return unless the role had finished its current task and went to the idle state. For any given role, it is only possible to go from the *Bound/Active/Idle* state to the *Bound/Passive* state as it was discussed previously.

getState(roleInstance)

This method is used to obtain the internal state information, which is returned through the return value of the method, of the given role. The internal state of a role describes application data stored within the attributes, for example, of the given role instance. A role of which state information is supposed to be obtained must be in the *Bound/Passive* life-cycle state at the time this operation is executed. Otherwise a defined error value will be returned to indicate that no state information could be obtained from the specified role instance. The criteria of the role to be in the *Bound/Passive* state for this method to succeed is also relevant to prevent the loss of application data. If a role was still allowed to perform computations that could possibly alter its internal state after the state information was obtained within an adaptation process, the subsequently made changes would be lost.

setState(roleInstance, stateData)

Obtained state information from a different role can be set to the role instance specified within this method. The successful execution of this method is indicated through a boolean return value. This method only succeeds if the role instance supposed to receive the state information is in the *Bound/Passive* life-cycle state at the time the operation is executed. The rationale for the required life-cycle state of the role to be in before this method can be successfully executed is the same as for the just discussed `getState` method.

connect(roleInstance, remoteRole)

The connect operation establishes a `collaboration` between two instances of roles, which do not necessarily need to be located on the same physical computational device. The connect operation takes the local role instance identifier as first input parameter and a tuple denoted as *remoteRole* consisting of $\{address, role\ instance\ identifier\}$ as second input parameter and returns a boolean value to indicate

whether the collaboration could be established successfully. Similarly to the *Bind* operation, an established collaboration between two roles is not immediately active, but remains in a preliminary state until an *Activate* operation is invoked on the role. Consequently, the role does not necessarily need to be in a passive state when the connect operation is invoked.

disconnect(roleInstance, remoteRole)

The disconnect operation is the inverse operation to the *Connect* operation and releases the `collaboration` between two roles. The signature is identical to the *Connect* operation, consequently. Since an ongoing collaboration between two different role instances is supposed to be released with this method, the local role runtime needs to ensure the role to reach the *Bound/Passive* state, i.e., a stable state to ensure no data in transit gets lost, in prior to the actual removal of the dispatch information.

Transitions between the life cycle states of a role (cf. Figure 3.2) denoted with a name that matches the name of the interface operation, are triggered by the invocation of the respective interface operation. Other interface methods are introduced to support the different scenarios introduced in Chapter 1.1.

3.2.3. Local Stable State for Roles

In Chapter 2.1.2, we discussed adaptation models that ensure the transition of a given program from a source to a target configuration. Those models proposed different approaches to change the application behavior ranging from an "instantaneous" adaptation of the application to a temporary co-existence of source and target behavior of the application. The core idea, however, in all three presented models is the notion of reaching a state in the program in which the behavior can be switched without data loss or any other harmful side effects. If such a state cannot be reached naturally, the source program's behavior has to be restricted in a way that such a state is guaranteed to be reached in bounded time. In [26, 45], two approaches were presented that aim to find such a state in which the source configuration can be replaced by the target configuration within a single adaptation step. Both approaches were discussed in detail in Chapter 2.1.1.

In our approach of a role model that supports context-dependent and collaborative adaptation, we do not describe a new approach comparable to *quiescence* and *tranquility*, but rather provide means within the role model to allow the implementation of either of those approaches if required. Evidently, the role life cycle and the adaptation interface aim at the *Guided Adaptation* model [7] (cf. Chapter 2.1.2). The role life cycle provides

two distinct states to distinguish the current working state of any given role instance: *Active* and *Passive*. The former is further subdivided into two states referred two as *Processing* and *Idle*. A role in the *Active/Processing* state is bound to a given player, thus, extending or modifying the behavior of the player. Furthermore, the bound role instance is currently serving a request, i.e., one of the role's provided methods was invoked and the execution is ongoing. If the execution of the behavior is finished and if no new behavior is invoked, the role instance is in the *Active/Idle* state.

Consequently, a role that is in the *Active* state cannot be considered to be in a stable state and is thus not ready for adaptation. Following this rationale, the *Passive* state of a role is considered the stable state of the role and a role in this state is thus ready for adaptation. Therefore, the *Active/Processing* state can be mapped to the *Active* state of the lifecycle proposed by Kramer & Magee in [26]. The quiescent state of [26] itself is represented by the *Passive* state in our approach. The *Active/Idle* state is comparable to the *Passive* state described by Kramer & Magee [26] since currently no computational tasks are performed, but can be started any time when one of the role's methods is invoked.

The respective interface operations of *Passivate* and *Activate* are used to enforce restrictions on the managed application's behavior or to release those restrictions in order to reach or leave a stable application state, respectively. Any additional implementation of a subsystem that is responsible to ensure quiescence, tranquility or a different approach to ensure a safe application state to be reached can utilize these interface operations and the provided role model to implement the desired criteria on the system during the adaptation process.

Having discussed the local mechanisms required to reach a local stable state of any given role instance, reaching such a state in a distributed environment in which the stable state spans across device borders involving multiple adaptations, will be presented in the following.

3.2.4. Distributed Stable State for Roles

In a distributed role-based application, adaptations may affect the configuration of the run-time model on multiple nodes. Imagine the basic scenario in which application behavior, i.e., roles, are supposed to be exchanged in response to a context change of the system. The different roles on the respective computational devices, however, collaborate with each other and with other roles that are not affected by the adaptation request. In order to perform the adaptation without violating the adaptation goals, i.e., to prevent an inconsistent configuration and the loss of application data of ongoing collaborations

during the adaptation process, it is essential to coordinate reaching a stable state for the roles affected by the adaptation request before the actual adaptation process starts. In the following, it will be discussed how this coordination can be achieved in order to fulfill the adaptation goals.

Our approach to determine a stable application state for roles in distributed environments relies on the collaborative nature of roles and the run-time information on role instances and their local and remote collaborations maintained by every role runtime. This information being present and retrievable by the execution management, enables the system to enforce a stable state for roles affected by the adaptation. The execution management applies the notion of *quiescence* [26] to determine this stable state. Additionally, we rely on the local runtime's dispatching mechanism to be able to keep track of ongoing invocations of role behavior.

Assume, two roles located on different devices and connected via a collaboration relation are supposed to be exchanged in response to a context change. Furthermore, the respective operations have been issued to the execution management instances located on the affected devices in order to exchange both roles. Before the actual adaptation can be performed, both roles are required to reach a stable state. In a first step, the *execution management* queries the respective local role runtime in order to obtain information about the collaborations of the roles affected by the adaptation. In a second step, *Passivate* messages are issued to all collaborating roles in order to move them to the `Bound/Passive` state temporarily. If all other execution management instances respond with a successful *PassivateResponse* message, the local role is moved to the `Bound/Passive` state. Collaborating roles on remote peers are moved to this `Bound/Passive` state as soon as they have reached the `Active/Idle` state. To enforce the respective remote roles to reach such a state, no new method invocations are allowed to be scheduled to these collaborating roles when the *Passivate* message was received. The description of reaching a distributed stable application state will be concluded by a brief explanation of the aforementioned messages, which have not yet been discussed:

PassivateRequest The *Passivate* message is used to inform remote execution managers that a given role has to be moved into a passive state in order for a given local role to be ready for adaptation. The *Passivate* message takes the instance identifier of the role located on the remote runtime. The remote role's identifier represents the role that is required to enter a passive state before the local adaptation can be performed.

PassivateResponse The *PassivateResponse* message indicates the successful transition

of the role to the `Bound/Passive` state. It contains the instance identifier of the role and a status flag that indicates a successful or unsuccessful transition of the role to the `Bound/Passive` state.

Activate The *Activate* message is the inverse coordination message to the *Passivate* message and triggers remote roles to resume their work after the adaptation process has been finished, i.e, the role instance is moved back to the `Active/Idle` state in which new invocations can be issued and served.

With this approach, we clearly follow the *quiescence* approach proposed by Kramer & Magee [26], which also results in a *Guided Adaptation* [7] of the decentralized adaptation process. Using the less disruptive *tranquility* [45] approach to ensure a safe application state would be possible if the internal communication scheme of every transaction, i.e., role collaboration, would be specified at design time as well as observable and interruptible at run time. This could be realized, for example, with the help of the MACODO [17] approach, which makes collaborations between roles explicit at design time, i.e., sequences of mutual invocations among collaborating roles are explicitly specified. Leveraging this knowledge at run time, aids the part of the coordination protocol to determine when a safe adaptation of collaborating roles is possible during an ongoing collaboration. Our proposed messages could still be of use though as a fallback, because *tranquility* [45] does not ensure a safe application state to be reached within a bounded timeframe.

3.3. Comparison with Role Features [28]

The application scenarios described in the first chapter of this thesis serve as foundation to define the capabilities and limitations of a *Role* within a role-based application. In the autonomous-car scenario, we described different behaviors of a car that depend on the current weather conditions. A *Role* thus may be *context-dependent*. In both the autonomous-car and the rescue-robots example, roles are used to modify the behavior of the system, but roles in these examples also collaborate with other roles to fulfill their task, e.g., cars that drive in a platoon or rescue-robots that work together to rescue victims. A *Role* thus may also be *collaboration-dependent*. The *behavioral* nature of roles is evident in all described application scenarios and is indeed the characteristic of a *Role* we always assume to be present. Our understanding of a *Role* encapsulates all three natures, which is the reason for the *Role* concept to be considered a suitable abstraction for dynamic system behavior with respect to the third research question. Roles, though, are not forced to always be context-dependent and collaborative besides having their own

Table 3.1.: Comparison of different role models.

Role Nature	MACODO	HELENA	MAS	Ours
Context-Dependent	✗	✓	✗	✓
Collaboration-Dependent	✓	✓	✓	✓
Behavioral	◆	✓	✗	✓

Supported: ✓ | Partially Supported: ◆ | Not Supported: ✗

state and behavior. Consequently, roles in our approach may be behavioral and either context-dependent or collaboration-dependent or both if necessary. Any of these combinations is supported for run-time adaptation by the proposed role model and adaptation interface.

With roles it is possible to capture context-dependent adaptations of behavioral and collaborating units of a self-adaptive software system without platform-specific knowledge about the implementation of these units. Other approaches, discussed in detail in the previous chapter, only support a subset of the role concept in terms of a *Role*'s nature. An overview what *Role* natures are supported by the previously discussed approaches, namely MACODO [17], HELENA [24, 19], and multi-agent-systems, e.g., [12, 3, 4], is given in Table 3.1. The collaborative nature of roles is the most often used part of the *Role* concept, which is not surprising since it is an intuitive mind model to describe the relation of two interacting computational units. The behavioral nature of roles appears to be less evident with respect to the collaboration of two computational units. Both multi-agent systems and MACODO implement the role-specific behavior as part of the player and just activate and deactivate the said behavior depending on the player being part of a certain collaboration or not. MACODO was graded with at least partial support for the behavioral nature of roles because a collaboration itself is abstracted as separate design unit prescribing the concrete interaction scheme and behavior of a role within the collaboration, which makes this nature a little more explicit than in multi-agent systems in general. So far, only the HELENA approach uses the concept of roles to capture context-dependent behavior of self-adaptive software systems.

HELENA and our approach include all three natures of roles. The HELENA approach uses *Ensembles* to make collaborations explicit [19] and an extension for self-adaptive software systems covers the context-dependent behavior of roles [24]. The two main differences between HELENA, especially including the recent extension for self-adaptive software systems, and our approach are the definition of players on the one hand and the focus on different parts of the life cycle of the system on the other hand. In the

HELENA approach, players are mere containers used for the exchange of data between roles whereas players in our approach possess their own state and behavior. HELENA proposes a methodology to design self-adaptive role-based software systems. We are concerned with the run-time adaptation of such role-based software systems in highly distributed environments in contrast.

Although other approaches also support all three role natures or a subset of them, the concrete role model used as a foundation differs even among approaches supporting the same set of features. Kühn et al. [28] performed an exhaustive study of existing role-based modeling and programming language approaches deriving a set of 26 features in total extending a seminal work on role features by Steimann [40]. This feature list is used as a foundation for a family of meta-models for role-based software systems. In the remainder, we compare our assumptions and requirements of a role-based software system and a role-based adaptation approach to these 26 features, thus, explicitly stating and summarizing the *Role Model* we rely on within this thesis. The list of the 26 role features including the degree of their support is given in Table 3.2.

The degree of support for the *Role* features was divided into four categories: Features that are fully supported (✓) are either fundamental assumptions we make on the role-based run-time system or are directly reflected in concepts presented in this chapter or the following. Not supported features (✗) are explicitly prohibited by our approach. On a given set of other role features, we do not make any assumptions about (❊). Such features are mainly related to the modeling level of a role-based application, but if such a feature is present at run time, the presented concepts in this thesis will remain unaffected.

Our approach supports the role features #1 through #5 since they postulate the foundations of the behavioral, collaborative and context-dependent natures of roles. In addition, features #15 and #26 are supported or rather required in order to address the concrete instances of roles, players and compartments at run time because run-time adaptations of the role-based software system would not be possible without these features. Feature #11 is explicitly supported to express roles to be able to extend the interface provided by the player, which allows a much broader range of context-dependent and collaborative applications in contrast to roles that are just specializing the already existent interface of the players playing them.

In our approach, we conceptually assume that only players can play roles. Allowing roles to play roles (#8) or compartments to play roles (#22 and #23) would render the current concept of reaching a stable state for roles impractical since much more relations to other run-time entities need to be considered. Furthermore, especially in the case in which roles play roles to further modify the player's behavior, we believe that such an

Table 3.2.: Our Role feature model using the initial 15 role featues by Steimann [40] and the 11 added by Kühn [28].

#	**Feature**	**Level**	
1.	Roles have properties and behaviors	M0, M1	✓
2.	Roles depend on relationships	M1	✓
3.	Objects may play different roles simultaneously	M0, M1	✓
4.	Objects may play the same role (type) several times	M0	✓
5.	Objects may acquire and abandon roles dynamically	M0	✓
6.	The sequence of role acquisition and removal may be restricted	M0, M1	✓
7.	Unrelated objects can play the same role	M1	✓
8.	Roles can play roles	M0, M1	✗
9.	Roles can be transferred between objects	M0	✓
10.	The state of an object can be role-specific	M0	✳
11.	Features of an object can be role-specific	M1	✓
12.	Roles restrict access	M0	✗
13.	Different roles may share structure and behavior	M1	✳
14.	An object and its roles share identity	M0	✗
15.	An object and its roles have different identities	M0	✓
16.	Relationships between roles can be constrained	M1	✳
17.	There may be constraints between relationships	M1	✗
18.	Roles can be grouped and constrained together	M1	✳
19.	Roles depend on compartments	M0, M1	✓
20.	Compartments have properties and behaviors	M0, M1	✗
21.	A role can be part of several compartments	M0, M1	✳
22.	Compartments may play roles like objects	M0, M1	✗
23.	Compartments may play roles which are part of themselves	M0, M1	✗
24.	Compartments can contain other compartments	M0, M1	✳
25.	Different compartments may share structure and behavior	M1	✳
26.	Compartments have their own identity	M0	✓

Supported: ✓ | Not Supported: ✗ | No Assumption: ✳

approach can be realized using appropriate dispatch information at run time. Another feature we explicitly do not support is #14, which states roles and players to share an identity. From an intrinsic perspective of the role-based application this assumptions holds true because roles and players are perceived as indistinguishable entities as soon as they are bound together. Our work, however, is concerned with the management and adaptation of (distributed) role-based applications and thus, needs to clearly distinguish between player, role and compartment instances.

All other features not explicitly mentioned are free of any assumptions and are mostly design-time-specific (M1). One exception is feature #10, which states the state of a player might be role-specific. Since we do not make any assumptions about players except for the fact that they are playing roles and have their own state and behavior, this feature is of no concern for our approach. Feature #21 allows a role to be bound to multiple compartments. Since we perform instance adaptation and require the concrete instance identifiers for roles, players and compartments, our presented approach is generic enough to allow this feature to be conceptually present at run time or not, which allows us to drop any assumptions about this role feature. Following this reason, compartments may or may not contain other compartments at run time (#24).

4. Decentralized Execution of Distributed Adaptations

The decentralized execution of adaptations is the first identified requirement for self-adaptive software systems supposed to be applicable in application domains coined by a high degree of distribution and decentralization and that require a context-dependent adaptation of their application behavior. In this chapter, the proposed approach to enable self-adaptive software systems to perform run-time adaptations without a central coordinator is presented.

The approach relies on the external control [38] approach, which is coined by a clear separation of adaptation management and managed application. Both subsystems of the self-adaptive software system are linked through a well-defined interface. This design approach was chosen to ensure the separation of adaptation-specific concerns from the role-based runtime hosting the managed application. A well-defined interface between adaptation management and managed application furthermore relieves the adaptation management from dealing with specific role features provided by the role-based runtime of the managed application. Hence, a reusable and widely applicable approach to coordinate adaptations in a highly distributed role-based software system is promoted. Additionally, we neither rely on a specific architecture of the managed application nor on a specific input set from previous phases of the feedback loop. Instead, the notion of *Roles* is used to (i) describe the possible adaptations of the managed application supported by the proposed approach; and (ii) to provide a specification how these changes have to be provided as input set from previous feedback loop phases. Consequently, using the external control approach results in a specification of well-defined interfaces between the adaptation management and the managed application as well as between the distributed instances of the decentralized execution phase of the feedback loop.

The resulting architecture of a self-adaptive software system that conceptually implements the proposed approach is depicted in Figure 4.1. Both adaptation management and managed application are depicted as two distinct subsystems of the overall software system. The managed application is distributed across several computational devices

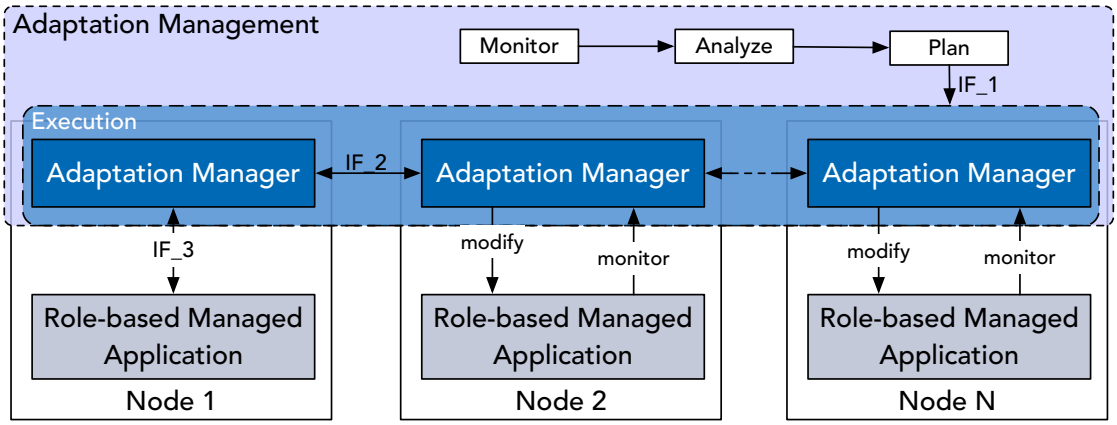


Figure 4.1.: System overview of a self-adaptive software system featuring a distributed and decentralized adaptation execution.

referred to as *Nodes* and comprises the role runtime and the actual application supposed to be adapted at run time. Both role-based application and role runtime have been discussed in detail in the previous chapter with respect to their conceptual contribution to the run-time adaptation (also cf. Figure 3.1 on Page 37). The *Adaptation Management* is divided into components representing the different phases of the MAPE-feedback loop. Since the actual execution of adaptations at run time is the primary research focus of this thesis, no assumptions are made about the distribution, decentralization or internal implementation of the phases for monitoring, analyzing or planning of the feedback loop. The proposed decentralized *Execution* phase realizes the execution of adaptations through *Adaptation Managers* distributed to all nodes of the system collaborating with each other to perform the adaptation process. Each adaptation manager controls and monitors its local role runtime of the distributed role-based application through a well-defined interface realizing the self-adaptation and self-awareness properties of the system.

The interface of the role runtime presented to the adaptation managers denoted as IF_3 in Figure 4.1 was discussed in the previous chapter. The adaptation managers shown in Figure 4.1 fulfill the tasks of the execution management shown in Figure 3.1 on Page 37 in addition to the decentralized execution of the run-time adaptations.

In the remainder of this chapter, we focus on the decentralized execution phase of the self-adaptive software system as the main contribution of this thesis. The system model and error models addressed by the approach will be discussed first. Subsequently, the interface between the feedback loop's planning component and the proposed decentralized execution component, which is denoted as IF_1 in Figure 4.1, will be discussed. This interface introduces role-based reconfiguration plans, which describe the steps that have

to be performed by the execution phase in order to migrate the managed application from the current configuration to the intended target configuration. Before the coordination protocol, which enables the adaptation managers to perform run-time adaptations without a central management instance (illustrated as IF_2 in Figure 4.1) is introduced in Chapter 4.4, the execution of adaptation steps will be discussed if these adaptations can be performed solely on a single node using IF_3. Having specified the protocol to enable the decentralized adaptation of the distributed managed application, a brief explanation will be given how the presented approach can be used to enable software updates of the context-dependent parts of the managed application. Finally, we will summarize the contributions presented in this chapter and relate them to the posed requirements and raised research questions.

4.1. System Model and Error Models

Both the decentralized execution phase of the adaptation management as well as the managed application are distributed to several *Nodes*. Nodes are assumed to be computationally powerful enough to host both the execution of a role-based application and the related execution component of the adaptation management. With respect to the underlying network, we assume all nodes to be connected to the same network. Furthermore, every node can reach every other node within one step. This is important for the presented coordination protocol since routing problems of coordination messages are of no concern for this work.

Neither the role-based application nor a single node nor the communication channel, however, is required to be reliable. In [31], three different error models for distributed systems and algorithms are defined: A *Link Failure* defines any group of failures in which messages put on any given channel can randomly get lost. A *Stopping Failure* defines any group of failures in which processes randomly stop during their execution. A *Byzantine Failure* defines any group of failures that are the result of faulty processes that exhibit completely unconstrained behavior, e.g., behavior that is not specified by any given protocol of the application.

Within this thesis, we address *Link* and *Stopping Failures* in the developed approach, i.e., the proposed approach will be able to consistently execute adaptations without central control despite *Link* and *Stopping Failures*. The coordination protocol, which is part of the approach and which will be introduced in Chapter 4.4, relies on the transmission of messages to distribute status and progress information during the execution of the adaptation. Link failures, therefore, denote the issue of such coordination messages be-

ing lost during transmission. Adaptation managers use this protocol to coordinate and synchronize local adaptation steps within the overall adaptation process. Stopping failures, in contrast, denote the issue of these adaptation managers going down ungracefully during the execution of an adaptation process. The developed protocol will be able to detect lost coordination messages and to react appropriately to cope with the link failure model. Furthermore, adaptation managers will have the possibility to recover from stopping failures. For adaptation managers that are still online, the protocol will specify the appropriate behavior of the adaptation managers to proceed with the execution of the adaptation process despite one or more adaptation managers having stopped suddenly.

In addition to link and stopping failures, we also address the special case of *adaptation failures*, which we introduce specifically within this thesis and which are not part of the failure models presented in [31]. Adaptation failures occur locally between adaptation manager and the role runtime providing the adaptation interface discussed in the previous chapter. An adaptation failure, for example, would be denoted by a `create` or `bind` operation invoked by the adaptation manager that returns a negative execution result, i.e., the role runtime was unable to instantiate the new role instance or failed to establish the play relation between role and player instance. The case of a randomly stopping role-based application during the execution of an adaptation process is also part of this failure scenario because local adaptations cannot be performed in this case as well.

Byzantine failures, which denote adaptation managers behaving maliciously to harm the system while an adaptation process is going on are not considered within this thesis. Rather, we assume all adaptation managers involved in the execution of an adaptation process to behave in accordance with the coordination protocol developed in this thesis.

In summary, link failures, stopping failures and adaptation failures will be addressed by the approach presented in this thesis. At this point, we want to highlight that every error model is only of concern for our work when the respective failure occurs during the execution of a concrete adaptation process. That means the solutions presented within the coordination protocol are not intended to serve as or replace a dedicated monitoring of the internal state of the self-adaptive software system within the feedback loop of the system to detect system failures or self-healing approaches, for example.

4.2. Adaptation Operations and Adaptation Transactions

In the previous chapter, the behavioral, context-dependent and collaborative nature of roles was discussed as well as the implications for the local role runtime with respect to the support of both local and distributed adaptations. Such an adaptation was described

(a) Structural Role-based Adaptation

(b) Collaborative Role-based Adaptation

Figure 4.2.: Structural and collaborative adaptations using adaptation operators according to Zhang et al. [7].

by Zhang et al. [7] as a transition of a program from a source system to a target system. With respect to distributed role-based software systems, the source system, which is also referred to as *source configuration* within this thesis, represents the entirety of binding relations between players and roles as well as the collaborations among roles. An adaptation, consequently, modifies the binding relations between players and roles or the collaborations among roles. The modification of the binding relation of players and roles is considered a structural adaptation [38] since the structure and composition of the system with respect to the behavioral entities, i.e., the roles, is modified. The relocation of a role from a player on one device to a player on another one as discussed in the second application scenario serves as an example for such a structural adaptation as well as the coordinated exchange of local system behavior (roles) discussed in application scenario one. In contrast, changing role collaborations does not effect the placement of roles but rather their interaction and is therefore considered a collaborative adaptation. The integration of a newly created role in existing system collaborations can be considered a collaborative adaptation of the application. Figure 4.2a depicts a structural adaptation of the system with the example of removing a role instance (represented by a circle) from a player (represented as a rectangle) and adding a new instance to a different player. A collaborative adaptation is depicted in Figure 4.2b in which one collaboration between two roles is released and a new one is established through an adaptation. Although each role runtime maintains its own local view of the role-based application, the overall run-time model of the distributed role-based application is considered as system or configuration that is reconfigured during an adaptation process, i.e., the depicted players in Figure 4.2 are not necessarily supposed to be located on the same device.

The modification of a binding relation between a player and a role or the modification of a collaboration between roles is regarded as a logical adaptation step to migrate the role-based application from its source to the desired target configuration. In the following, the concept of an *Adaptation Operation* will be discussed, which is used to describe such a single adaptation step. Subsequently, the properties and features of an *Adaptation*

Transaction in our approach to allow a decentralized adaptation of role-based software systems will be discussed. As it was previously mentioned, the transition of the managed application from the source to the target configuration may be composed of several adaptation steps, i.e., adaptation operations. We encapsulate several adaptation operations that denote a logical transition of the system within an adaptation transaction to ensure a consistent adaptation.

4.2.1. Adaptation Operations

An *Adaptation Operation* describes a logical adaptation step of the role-based application modifying either the binding relation between a player and a role or the collaboration between roles. Following this consideration, an adaptation operation has to capture information of both source and target configuration. The former is used to identify roles and players of the current configuration subject to the adaptation whereas the latter is used to describe the state of role-player bindings or role collaborations after the adaptation was performed. Therefore, an adaptation operation requires the following structure representing parameters that have to be filled with the concrete run-time information of roles and players etc.

Operation ID This identifier for the adaptation operation needs to be unique within the scope of an adaptation transaction to distinguish between individual adaptation operations.

Operation Type An adaptation operation to add or remove a new role instance to or from a player requires almost the same set of information in order to describe the target configuration of the system after the operation was performed, for example. The *operation type* is therefore used to specify the exact semantics of the adaptation operation, e.g., if the specified information for the target configuration of the system represents an added or removed role instance. We will first finish the discussion of the remaining parameters of an adaptation operation before we will focus on the concrete semantics our approach supports as adaptation step.

Target Node This parameter describes the future configuration of the adapted application. Within the target node, the player, compartment and role information are specified as well as the address of the device that is hosting the adaptation manager, which is then responsible for altering its co-located role-based application. The *target node* describes the configuration of the adapted application after the operation has been performed, e.g., if new behavior is supposed to be added to

a player, the target node would denote the concrete player instance and the role type to bind to the given player instance as well as the compartment instance in which the role is supposed to be played by the player. After the execution of the operation, a new instance of the specified role type is bound to the player, therefore, modifying the player's behavior. If behavior is supposed to be removed from a player, for example, the specified role instance is not played any further by the specified player after the operation was carried out.

Order This parameter is used to establish predecessor-successor relations between different adaptation operations. Adaptation operations with a lower value for this parameter will be executed before other adaptation operations configured with a higher value and the execution of subsequent adaptation operations will not be started until all previous operations finished successfully. Evidently, adaptation operations that carry the same value for this parameter are allowed to be executed in parallel.

Depending on the operation type denoting the execution semantics of the adaptation operation at run time, the following two additional parameters have to be specified in order to fully describe the adaptation step from the source to the target configuration:

Source Node This parameter describes the current configuration of the role-based application. The *source node* is composed of the same set of parameters as the target node parameter, but describes the current configuration of the application. More specifically, the tuple of role, player and compartment instances represent the starting point for the adaptation step represented by the adaptation operation.

State This property is only required if both source and target node are specified for a specific adaptation operation. It prescribes if internal state information of the involved role is supposed to be transferred from the source to the target configuration of the managed application.

The overall structure of adaptation operations is summarized in Figure 4.3 including a graphical distinction for mandatory and optional parameters.

All adaptation operators share the same set of configurable parameters as it was previously discussed. The semantics and, thus, the reconfiguration that is eventually executed by the adaptation managers is encoded in the *operation type* parameter. In the following, we briefly present and describe the operation types that are currently supported as adaptation steps by the proposed decentralized execution phase.

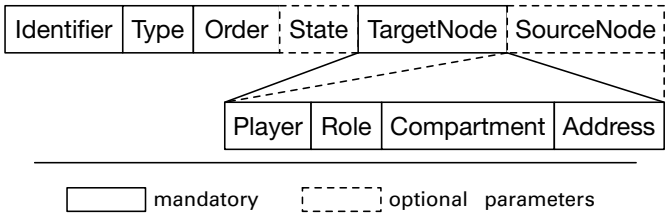


Figure 4.3.: Mandatory and optional parameters for *Adaptation Operation*s.

Add Using the *add* operation type, new behavior is added to a specific player instance, e.g., encryption behavior is added to two respective players using this operation type when the user is detected to enter an untrusted network. Consequently, only the target node parameter of the adaptation operation is required to perform this adaptation step. The target node requires the concrete type and instance of the player to add the new behavior to as well as the type of the role that represents the new behavior and the type and instance information of the compartment the role is expected to be played in. At run time, a new instance of the specified role type will be created and bound to the specified player instance, thus modifying the player's behavior.

Remove Using the *remove* operation type, existing behavior is removed from a specific player instance. For example, the previously added encryption behavior is removed from the respective players locally as soon as the user is detected to enter a secure and trusted environment. Since this operation can be considered the inverse operation to the `add` operation, only the target node parameter is required as well. In contrast to the `add` operation, however, concrete instance information on the player, role and compartment has to be specified in order for the adaptation management to remove the behavior at run time. During the execution of this operation, the adaptation managers ensure a stable application state before the role instance is unbound from the player and subsequently destroyed.

Exchange The two previous operations can be used to add or remove behavior to or from a player instance, respectively. In response to a context change, however, it might be reasonable to replace existing behavior with new behavior that fits the current context better. The exchange of concrete driving behavior of autonomous cars described in the first application scenario represents an example for replacing a role in response to changes of the application's operational environment. Such a scenario cannot be conveniently covered with just the `add` and `remove` operations since they are two distinct operations which makes it difficult to maintain a consistent

application state during the execution. More importantly, with the `add` and `remove` operation alone, the adaptation managers would not be able to transfer internal state information of the role from the source to the target configuration. The *exchange* operation addresses these shortcomings and allows to replace existing behavior with new behavior consistently. The *exchange* operation requires the full parameter set, i.e., both source and target node parameters have to be configured as well as the state parameter. Since the source node parameter describes the current version of the system, it is configured the same way as the `remove` operation's target node parameter because the role in the source configuration is supposed to be removed from the given player instance. Likewise, the target node of the *exchange* operation is configured like the `add` operation's target node parameter, i.e., only the role type needs to be specified since a new instance of the role type will be created at run time. During the exchange of a role, state information can be transferred from the source to the target role if indicated by the state parameter of the adaptation operation. The adaptation managers will synchronize reaching a stable application state before the transfer of the internal state information of the role.

Migrate The relocation of a role from one player instance to a different player instance (on a possibly different node) exposes the same shortcomings as the exchange of roles when realized using `add` and `remove` operations alone. The *migrate* operation is used to migrate a role instance from one player instance to another and requires the full set of parameters, i.e., both source and target node as well as the state parameter, to be configured. Source and target node of the *migrate* operation have to be configured with the same set of information as the `exchange` operation. Although a state transfer can be considered the default step for a migration of application behavior, the state parameter is required to cover also stateless migrations of roles since the coordination process, which will be presented later in this chapter, can be significantly simplified if no state information is supposed to be migrated. Reaching a stable application state for the role and player before the actual migration takes place is ensured by the adaptation managers for this operation as well. The relocation of the rescuer behavior described in the domain of search-and-rescue robots (cf. Application Scenario 2) serves as example for the purpose of the *migrate* operation.

Clone Using the *clone* operation type, a behavior similar to the `migrate` operation can be achieved. The role instance, though, specified by the source node remains

active after the adaptation process finished. This results in two different instances of the same role type being active, which possess the same internal state after the successful execution of the adaptation process. The mandatory information to denote the source and target node, respectively, is the same for this operation as for the `migrate` and `exchange` operation. Similarly to the relocation of application behavior in the domain of search-and-rescue robots, this operation can be used if the original role instance is supposed to be kept alive to allow the existing and the new role to collaborate for the rescue task. The replication of roles can also be used to achieve an effect similar to load balancing because roles are duplicated and can independently serve requests.

Connect Using the *connect* operation a collaboration relation can be established between two role instances, e.g. in order to integrate newly created and bound role instances into existing collaborations with other roles. Consequently, any method invocation intended for the collaboration partner is dispatched by the role runtime to the respective player playing the collaborating role. The *connect* operation uses the source and target node parameters of the adaptation operation to specify the two endpoints of the collaboration. In this case, the order which endpoint is considered the source and the target node is not important and does not relate to the otherwise valid assumption of source and target configuration specification of the respective parameter.

Disconnect The *disconnect* operation is the inverse operation to the `connect` operation and takes the same parameters as input to release the collaboration relation between the roles specified as source and target. Additionally, the adaptation managers will ensure that all ongoing communication between the affected roles have finished, i.e., both roles are in a passive state before the execution of the adaptation operation. Evidently, this operation is used if certain collaborations are not required any longer, e.g., if no further collaboration is required among search-and-rescue robots.

Except for the *add* and *remove* operations all operations require a source and target node specification to unambiguously determine the roles and players involved in the adaptation step describing the application's transition from the source to the target configuration. Using the *connect* and *disconnect* operation types, collaborative adaptations can be performed upon the managed application. All other operation types represent structural adaptations of the distributed role-based software system and are motivated by the role features #5 and #9 discussed in Chapter 3.3 stating that "*roles can be acquired and*

dropped dynamically" by players and that "*roles can be transferred between [players]*," respectively.

Please note that collaborations between roles are not considered in general when structural adaptations are performed, i.e., collaborations of a role that is supposed to be migrated have to be explicitly released using the *disconnect* operation before and reestablished with the help of the *connect* operation after the migration was performed. The `Order` parameter of the adaptation operation can be utilized to structure such dependent adaptation operations hierarchically and therefore express the execution order. Using the notion of an *adaptation transaction*, which will be presented in the following, such dependent operations can be described and ensured to be performed in the required order.

4.2.2. Adaptation Transactions

In highly distributed self-adaptive software systems, adaptations caused by changes in the system's computational environment are unlikely to affect only one role on one node. Rather, changes will affect several parts of the distributed application on multiple nodes including multiple changes on each of these nodes. If all of these changes were executed independently, the managed application would be in an inconsistent configuration during the time consumed by the execution of all these adaptations. Consequently, application data could be lost or the managed application might even fail. The main goals of the decentralized adaptation approach, however, are to be prevent the managed application from reaching an inconsistent configuration while being adapted, thus, ensuring no application data will be lost.

We introduce the term *Adaptation Transaction* to address these goals. An adaptation transaction defines a scope in which the collaborating adaptation managers ensure the managed application to remain in a consistent configuration preventing data loss. Adaptation operations required to be performed in response to a context change are bundled together and issued to the decentralized execution component comprised of adaptation managers by the planning component of the self-adaptive software system. An adaptation transaction, therefore, contains a set of different adaptation operations and a unique identifier to unambiguously identify the adaptation transaction at run time. Figure 4.4 provides a coarse-grained overview about the information transmitted from the planning subsystem to the decentralized execution component at run time in order to modify the managed application. While the identifiers of adaptation operations only have to be unique within the scope of an adaptation transaction, the identifier given to the adaptation transaction is expected to be unique with respect to the entire lifetime of the self-adaptive software system. A unique identifier for adaptation transactions is

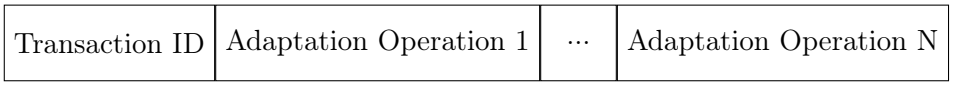


Figure 4.4.: Conceptual structure of an *Adaptation Transaction*.

crucial for protocol mechanisms addressing the handling of stopping failures, which will be discussed at a later point in this chapter.

To achieve a consistent adaptation of the managed application, several mechanisms are used during the execution of an adaptation transaction. A major principle to support a consistent adaptation is the *atomic* execution of an adaptation transaction, i.e., either all adaptation operations contained in the adaptation transaction are performed or none at all. Consequently, intermediate changes made to the managed application are rolled back using error handling mechanisms. This immediately ensures the *consistency* of the managed application, because the managed application is either kept in its source configuration if an error occurred or reaches its intended target configuration if the adaptation transaction could be finished successfully. In the remainder of this thesis, we focus on the execution of one adaptation transaction at a time in response to context changes. Mechanisms to ensure the *isolation* of an adaptation transaction if multiple adaptation transactions are issued in parallel or overlap each other are not considered and will remain an open issue for future work. The planning subsystem of the self-adaptive software system is thus required to ensure only one adaptation transaction being issued and executed at a time. An adaptation transaction is also not concerned with the *durability* of the changes that were performed, but only ensures the managed application to reach its intended target configuration or to remain in its source configuration. Other approaches, e.g., role-based databases [22], can be used when the execution of the adaptation process is finished to store the current system configuration persistently.

In contrast to the definition of a *Transaction* given by Kramer & Magee [26] and Vandewoude et al. [45], we do not consider the collaboration and exchange of messages between roles a transaction. To make the distinction clear, however, we will always refer to our definition of a consistent adaptation as *adaptation transaction*.

4.3. Adaptation Operations and the Role Runtime's Adaptation Interface

The execution of an *Adaptation Transaction* always includes local changes of the binding between roles and players or of the collaboration relations between roles. In this section, we focus on the interface between local adaptation managers and the co-located role

runtimes with respect to the execution of a single adaptation operation. The decentralized execution component currently supports the execution of seven distinct adaptation operations that could be derived from the features of the *Role* concept. Those seven adaptation operations can be divided into two groups: The first group only requires the `target node` parameter of the adaptation operator to be set (Add, Remove) whereas the second group requires both `source node` and `target node` parameters to be set in order to describe the desired change (Exchange, Migrate, Clone, Connect, Disconnect). We will refer to the first group as *Local Operations* and to the second group as *Distributed Operations*. Please note that distributed operations might also be solely locally executable if the address parameter for the target and source node points to the same node. Even in such a case, we refer to operations that require the specification of both source and target node as *Distributed Operation*.

Independent of the concrete adaptation operation, the execution of every adaptation operation is a two-phased procedure. We make use of the notion of bound but passive roles to perform changes transparently to the currently visible behavior of the managed application. The first phase is called *Pre-Activation Phase* and prepares the role-based application for the actual change of the behavior, e.g., role instances are passivated or newly created and bound. In the second phase, the *Post-Activation Phase*, the performed changes of the first phase are activated and necessary clean-up tasks are performed. An *Activation Event* starts the post-activation phase and is triggered when the pre-activation phases of all adaptation operations within an adaptation transaction were performed successfully. A distinction between both phases during the execution of an adaptation operation is required to reach a stable application state and to maintain a consistent system configuration. Until the activation event, changes of the software system do not affect the system behavior except for the passivation of role instances that are, for example, to be removed or migrated. The duration a role instance is passive denotes the period in which behavioral or collaborative parts of the managed application are unavailable during the adaptation process. Moreover, a synchronized activation of changes based on a coordinated activation event prevents the execution of contradicting behaviors or invalid collaborations among roles, thus, aiding in maintaining a consistent configuration of the managed application during the adaptation process.

The idea to split the local execution of adaptation operations into two distinct phases helps to maintain a consistent configuration of the application locally, especially when multiple adaptation operations are performed involving the collaboration of multiple adaptation managers. The approach resembles and is inspired by the commonly known commit protocols in which a consensus on the successful execution of a preparation phase

is reached before the local changes to the data become globally visible. The pre-activation phase can be mapped to the preparation phase and the activation event can be mapped to the reached consensus. In our case, also the new behavior of the system becomes active. Additional mechanisms necessary while performing the pre-activation phase on roles, which are performing computational tasks themselves and need to be moved to a stable state, go beyond the scope of commit and consensus protocols to agree on the success or failure of a distributed transaction.

Using the interface operations provided by the role runtime to the adaptation manager (cf. Chapter 3.2.2), the adaptation manager is able to detect local adaptation failures. The successful execution of interface operations of the pre-activation phase is indicated using the specified return value of the respective interface operation. If this return value indicates the operation to have failed, the adaptation manager will be able to react appropriately, i.e., the execution of the adaptation transaction can be terminated unsuccessfully and the source configuration of the system can be restored. The concrete failure handling behavior of the adaptation managers will later when the decentralized protocol to manage the execution of an adaptation transaction is presented.

In the following, we will first focus on the execution of local operations and subsequently on the special case of distributed operations that can be performed on a single node.

4.3.1. The Execution of Local Operations

Local adaptation operations are used to modify the structure of the role-based application in the most basic way, which is to create a new role-player relationship or to release such a relationship. The respective adaptation operations supported by the decentralized execution component are `add` and `remove`.

In Figure 4.5, the execution process of a single `add` and `remove` operation is displayed. In order to modify the role-based application, the adaptation manager uses the interface operations specified in Chapter 3.2.2. In the exemplary execution of both operations, the post-activation phase of both operations would be started as soon as the pre-activation adaptation steps were successfully performed to maintain a consistent configuration of the role-based application. The decomposition of local adaptation operations to interface operations provided by the role runtime for the respective activation phase is summarized in Table 4.1. If the adaptation transaction only consists of a single local operation, the adaptation manager immediately executes the post-activation phase after finishing all operations of the pre-activation phase. An adaptation transaction, though, may also be composed of multiple local operations and distributed operations local to a specific node,

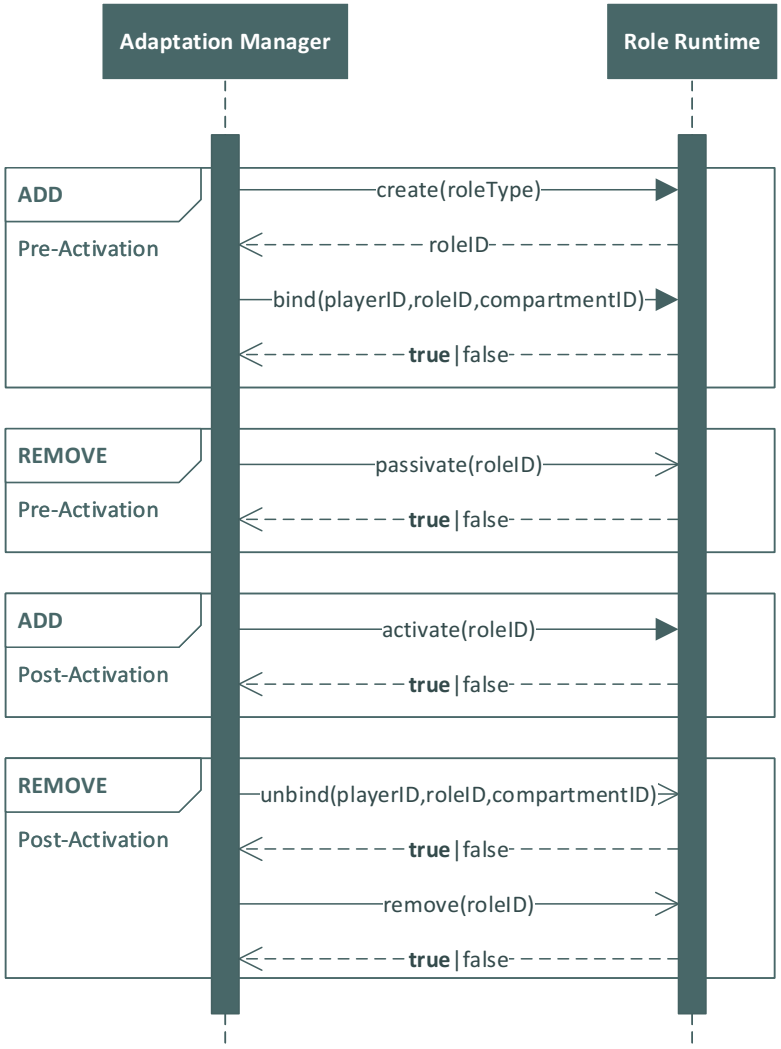


Figure 4.5.: The sequence of invoked role runtime methods of the adaptation manager for the execution of local operations (`Add` & `Remove`).

Table 4.1.: Mapping of local operations to interface methods of the role runtime for the pre-activation and post-activation phase.

Operation	Pre-Activation Phase	Post-Activation Phase
Add	Create, Bind	Activate
Remove	Passivate	Unbind, Remove

which may also be ordered in their execution by the planning component of the feedback loop. In this case, the adaptation manager first collects the results of all pre-activation phases of each adaptation operation and only continues with the execution of the post-activation phase if all operations succeeded without error. A specific execution order for adaptation operations configured with the same `Order` parameter in this case is not prescribed by the protocol since these operations can also be performed in parallel if the computational resources on the respective device are sufficient.

Since preliminarily changes of the pre-activation phase do not affect the system behavior, no inconsistent system configuration of the managed application can be reached, which is especially favorable in the presence of adaptation failures in which the adaptation fails. Due to the preliminary character of the pre-activation phase, passivated roles simply need to be reactivated to re-instantiate the valid source configuration's behavior. Other preliminary adaptations can be reverted in the background when the managed application has already resumed its tasks.

4.3.2. The Execution of Distributed Operations

Distributed operations are also translated into a sequence of interface operations provided by the role runtime similar to the previously discussed local operations, but applied to both source and target node. Table 4.2 provides an overview of the relation of the distributed adaptation operations and the execution of interface operations provided by the role runtime for the pre- and post-activation phase of the adaptation operation's execution cycle, respectively. Similar to the execution of local operations, the adaptation manager has to collect the results of the pre-activation phase for both source and target node first before the post-activation phase is executed. Assume the execution of a migrate operation without the distinction of two phases and a situation in which the binding of the created role instance fails for the target node specification, but the source node's role is passivated and immediately unbound and removed. In such a scenario, the role information will be lost and temporary changes could not be rolled back although the adaptation transaction had to be terminated and the source configuration has to be reestablished. Hence, a consistent application configuration cannot be maintained. With

Table 4.2.: Mapping of distributed operations to role runtime interface methods for the pre-activation and post-activation phase.

Operation		**Pre-Activation Phase**	**Post-Activation Phase**
Exchange	Source	Passivate	Unbind, Remove
	Target	Create, Bind	Activate
Migrate	Source	Passivate	Unbind, Remove
	Target	Create, Bind	Activate
Clone	Source	Passivate	Activate
	Target	Create, Bind	
Connect	Source	Connect	Activate
	Target		
Disconnect	Source	Passivate	Disconnect
	Target		

respect to the execution of multiple distributed operations local to a specific node within an adaptation transaction, the adaptation managers behave in the same way as described previously for the local operations. This holds also true if local and distributed operations are mixed within an adaptation transaction.

Each local role runtime maintains information about role collaborations alongside the current bindings of roles and players. Releasing (`disconnect`) or establishing (`connect`) collaborations between roles in response to context changes follows the previously described idea of a pre-activation and post-activation phase for the respective role. In the pre-activation phase for the disconnect operation, both roles specified by the source and target node are moved to a passive state to ensure all ongoing communications between the roles were stopped gracefully and the dispatch information is marked for deletion within the local role runtime. In the post-activation phase, the passivated roles are activated and the dispatch information is finally discarded, which releases the collaboration between both roles. If a new collaboration is supposed to be established using the connect operation, the behavior is almost inverted. First, a dispatch information is created in the local runtime, which is not considered for the dispatch of the method invocation unless the post-activation phase is performed in which this preliminary addition is finally activated. The post-activation phase in both cases is triggered by an activation event the same way as for adaptation operations performing structural adaptations described previously.

4.4. The Decentralized Coordination Protocol

The decentralized coordination protocol is represented by *IF_2* in Figure 4.1 on Page 54 and describes the execution of an adaptation transaction by the adaptation managers. This coordination protocol is composed of a set of messages exchanged by the adaptation managers to collaboratively execute adaptation transactions as well as a behavioral description how adaptation managers respond to received messages at different stages during the execution of an adaptation transaction. In this regard, the execution of distributed adaptation operations that require the collaboration of two adaptation managers will be discussed as well as the execution of multiple adaptation operations affecting parts of the managed application located on different nodes in general.

First of all, the coordination messages used by the adaptation managers to execute an adaptation transaction will be discussed. Subsequently, the general execution of an adaptation transaction will be introduced. As previously mentioned, adaptation operations can be grouped by the `Order` parameter to establish predecessor-successor relations for their execution. An *Adaptation Group* denotes the execution of adaptation operations configured with the same `Order` parameter, i.e., those adaptation operations can be executed in parallel and in a random order. The coordination of the execution of such adaptation groups as subunit within an adaptation transaction will then be discussed. Incorporated in this discussion, the coordination of distributed adaptation operations that involve the collaboration of two adaptation managers will be presented, because this aspect is closely related to coordination aspects within an adaptation group. The handling of link failures, i.e., lost coordination messages during the execution will be discussed alongside the general protocol behavior as outlined. Stopping failures, in contrast are more crosscutting affecting different parts of the coordination protocol and also require additional protocol behavior. The aspects of the coordination protocol addressing stopping failures, will be therefore be presented separately.

4.4.1. Protocol Messages

The protocol messages used for the execution of an adaptation transaction can be separated into two categories: *Transaction Control Messages* and *Execution Control Messages*. The first category is used for the distribution of general progress information, i.e., if an adaptation transaction's execution failed or if it could be finished successfully. Execution Control Messages are used explicitly for the coordination of adaptation groups and adaptation operations. All messages are solely exchanged between adaptation managers during the distribution and execution of an adaptation transaction and are introduced

here to relief further discussions of the protocol behavior from the discussion of used protocol messages. The parameters and information transmitted via each transaction control message and execution control message is illustrated in Appendix B.1 and Appendix B.2, respectively.

Transaction Control Messages

Transaction control messages are used to distribute adaptation transactions within the group of nodes affected by the adaptation process and to indicate a successful or failed execution of the adaptation transaction. Four message types serving that purpose can be distinguished:

Transaction Message This message is used to distribute the adaptation transaction that is supposed to be executed among the adaptation managers of the self-adaptive software system required to perform adaptation operations. The message, therefore, contains only one adaptation transaction, which includes the unique identifier for the adaptation transaction used by all other messages to relate them to a specific adaptation transaction.

TransactionAcknowledgement Message This message contains the identifier of the transaction and indicates that the transmitted adaptation transaction was received successfully.

TransactionRollback Message This message is used to indicate that the execution of the adaptation transaction has failed and that preliminary pre-activation-phase adaptations have to be rolled back in order to maintain a consistent configuration of the managed application, which is the source configuration. Besides the identifier of the adaptation transaction, the unique identifier of the adaptation operation that caused the rollback is transmitted as well.

TransactionActivation Message This message is used by the adaptation managers to indicate that all adaptation operations involving the respective adaptation manager's collaboration were executed successfully and that expected remote progress information, i.e., Report messages, which will be discussed subsequently, was received. Thus, the execution of the adaptation transaction was evaluated to be performed successfully, which is indicated by that message. Evidently, the identifier of the adaptation transaction that is acknowledged to be successfully executed is the sole parameter of this message.

Execution Control Messages

The execution control messages are most important for the coordination protocol with respect to the management of the adaptation transaction's decentralized execution. Different types of messages are used to disseminate progress and status information among adaptation managers as well as internal state information of roles to ensure the consistent execution of single adaptation operations. Additionally, a third message type is required to cope with cases in which one of the aforementioned messages was lost during the transmission.

Report Adaptation managers rely on this message to exchange progress information on the execution of distributed adaptation operations, i.e., the message is used by adaptation managers responsible for adaptations denoted by the source and target node, respectively. Furthermore, the *Report* message is used to distribute progress information on the successful execution of adaptation operations within the execution of an adaptation group. A *Report* message contains the unique identifier of the adaptation operation about which status information is exchanged or propagated as well as the identifier of the adaptation transaction, the respective adaptation operation belongs to. For the transmitted status information within the report, three mutually exclusive flags are required that indicate when set whether the respective adaptation operation could be successfully performed or not or if the operation is still being executed. The third flag is required especially if pre-activation-phase operations such as the passivation of a role, consume a larger amount of time that extends beyond the timeout period of the adaptation manager's execution behavior. In that case, the status value contains a manifestation representing the operation to be still going on.

StateTransfer The distributed operations allow to transfer state information of the role from the source to the target configuration. A *StateTransfer* message contains the identifiers of the adaptation operation and of the adaptation transaction in which it is executed as well as internal state information of the source node's role instance. We assume the state information of a role to be obtainable only when the role is in a passive state, i.e., no computations are being performed that may alter the role's internal state. Therefore, the *StateTransfer* message can be used as a replacement for the report message to reduce the number of messages an adaptation manager on the source node has to transmit to its peer on the target node. For example, transmitting a *StateTransfer* message from the source node of a `migrate` operation immediately means that the steps of the adaptation on the source side

could successfully be performed. Otherwise, a negative report message would be used.

RequestReportMessage During the execution of distributed operations and adaptation groups, *Report* and *StateTransfer* messages are exchanged to coordinate the adaptation progress among adaptation managers. In case of link or stopping failures, adaptation managers may not receive sent coordination messages or may not be able to send coordination messages, respectively. In that case, adaptation managers make use of the *RequestReport* message to request progress information on adaptation operations of which respective reports are missing. Depending on the concrete type of the adaptation operation a report is requested for, the peer answers with either *StateTransfer* or *Report* messages. A *RequestReport* message takes the identifier of the adaptation transaction as input and a number of adaptation operation identifiers for which reports are requested to be retransmitted.

4.4.2. Decentralized Coordination of a Transaction

In response to context changes of the self-adaptive software system, the planning phase of the feedback loop issues an adaptation plan to the execution phase. This adaptation plan is represented as an adaptation transaction in our approach and is executed by the group of instances of adaptation managers affected by the adaptation transaction. The set of adaptation managers affected by the adaptation transaction and supposed to participate in the execution process is determined by the collection of distinct address parameters of both source and target node parameters of the adaptation operations contained in the adaptation transaction. The planning phase is only required to transmit the adaptation transaction to one adaptation manager. This initial receiver will forward the received adaptation transaction to all other adaptation managers, which we also refer to as *peers*, included in the adaptation transaction.

As shown in Figure 4.6, the first step in an adaptation transaction's execution is its distribution among peers. The Transaction message is used for the distribution of the adaptation transaction. The receipt of an Transaction message is acknowledged using the TransactionAcknowledgement message. As soon as an adaptation manager received the TransactionAcknowledgement messages from every peer, the execution of the adaptation transaction commences. The initial receiver, furthermore retransmits the Transaction message after a specified timeout to all peers that have not responded with an TransactionAcknowledgement message yet.

In the case of lost TransactionAcknowledgement messages not all adaptation managers

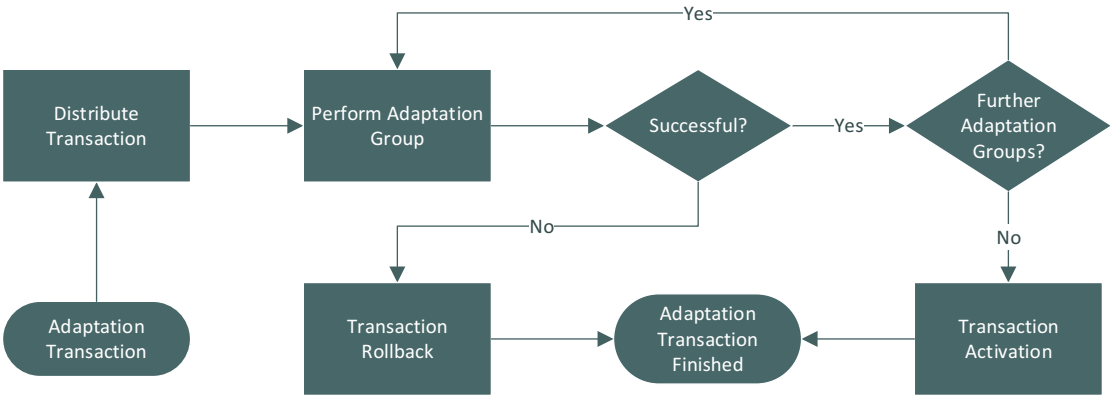


Figure 4.6.: Statechart / workflow of the execution of an adaptation transaction.

will therefore commence the execution of the received adaptation transaction at the same point in time. It is possible for adaptation managers to receive the first Report message before all expected TransactionAcknowledgement messages were received. The adaptation manager that received the Report message will consider the collection of acknowledgements successful and begins with the adaptation transaction's execution even though not all acknowledgements could be received. This behavior is valid because receiving a Report message confirms that at least one adaptation manager (the one that sent the Report message) was able to establish knowledge on the successful distribution by receiving TransactionAcknowledgement messages of every peer adaptation manager.

After successfully distributing the adaptation transaction among all peers, the adaptation managers preprocess the received transaction to determine the *Adaptation Groups*. In an adaptation group, all adaptation operations parameterized with the same value for the `Order` parameter are grouped together because those adaptation operations are allowed to be executed in parallel. After the set of adaptation groups was generated it is ordered and the execution of the group with the lowest value is started. Each adaptation group, therefore, has its own *Group Execution* state within the adaptation transaction. Progress management within an adaptation group will be discussed in Chapter 4.4.3. In general, however, Report messages are used to indicate the execution state of every adaptation operation contained within an adaptation group. The adaptation manager can therefore determine how many Report messages are expected to be received from the peers for the currently executed adaptation group. If all Report messages were received, the adaptation manager continues with the execution of the next adaptation group if there is another one or finishes the execution of the adaptation transaction with the emission of a TransactionActivation message to all peers, otherwise. If the local execution of an adaptation operation contained in the adaptation group fails or if a Transac-

tionRollback message was received from a peer, the adaptation manager terminates the current execution and enters the *Rollback* state. In the *Rollback* state all modifications made in the pre-activation phase of the execution of adaptation operations are reverted, i.e., passivated roles are reactivated and preliminary role creations or bindings are undone. TransactionRollback messages are also forwarded to other peers once after the first receipt to disseminate the information quickly among the group of adaptation managers.

4.4.3. Decentralized Coordination of Adaptation Operations and Adaptation Groups

The specification of the coordination management for adaptation transactions focused so far on the distribution of the adaptation transaction and the behavior of the adaptation managers if the adaptation transaction could be finished either successfully or unsuccessfully. Details on the actual execution for distributed adaptation operations and adaptation groups composed of different distributed and local adaptation operations have still not been discussed. In the following, emphasis is first put on the execution of distributed adaptation operations that truly require two adaptation managers to collaborate in order to perform the adaptation step. Subsequently, the protocol behavior for the execution of adaptation groups will be developed.

Coordinating Distributed Operations

We introduced *Distributed Operations* as adaptation operations that contain both source and target node parameters. If the address property of the target node parameter is different from the source node's address property, the distributed operation requires additional coordination since two adaptation managers (the one located on the source node and the one located on the target device) have to collaborate in order to perform the adaptation. Both adaptation managers execute the respective operations for the pre-activation phase as it was already discussed for distributed adaptation operations in Chapter 4.3.2. When the pre-activation phase of either node is finished, the adaptation managers involved in the execution of the distributed operation notify each other about the progress with Report messages if no state information is supposed to be transferred between source and target node (`state=false` in the adaptation operator's configuration). In this case, the execution of the distributed adaptation operation is finished and the adaptation manager responsible for the target node adaptation emits a positive Report message to all peers. If state information is intended to be transferred from the source to the target node, the source node's adaptation manager sends a StateTransfer

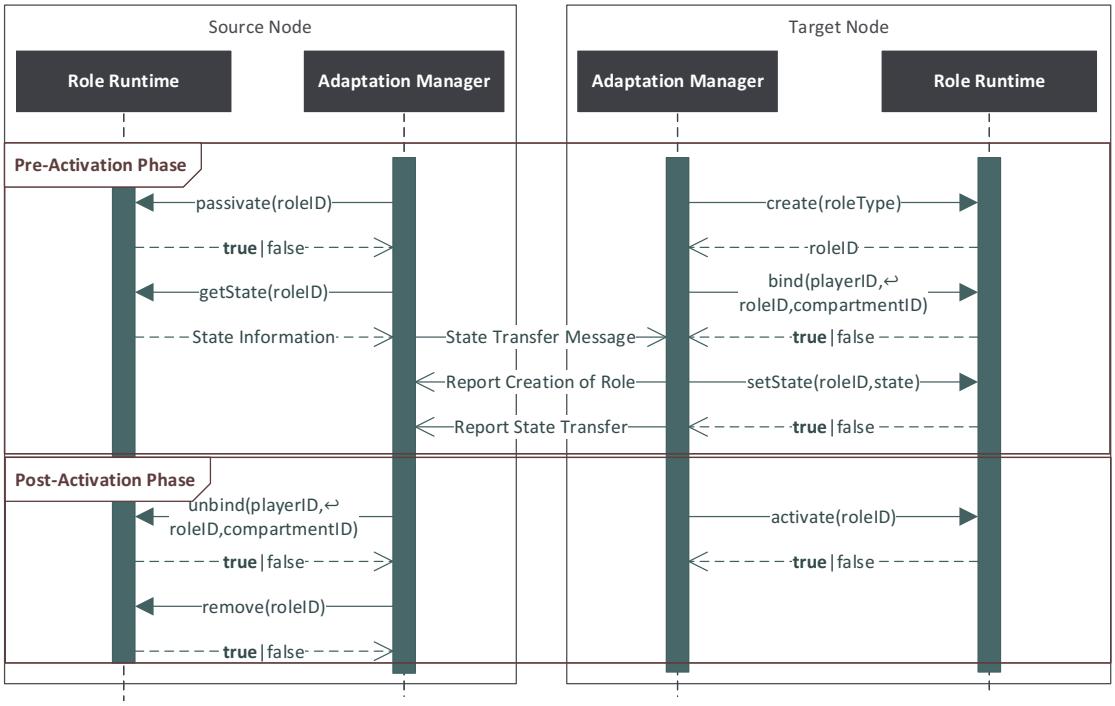
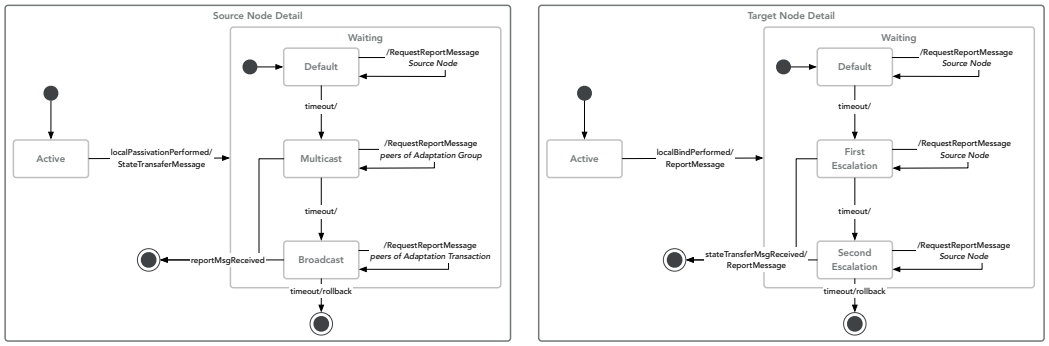


Figure 4.7.: Distributed coordination of a TS-Operation including local communication of the adaptation managers with their co-located Role Runtimes.

message to the target node containing the internal state information of the role instead of indicating the successful execution of the pre-activation phase using a Report message. The target node will still create and bind the new role instance and transmit a Report message to the source node to report the successful execution of the pre-activation phase. Additionally in this case of state transfer, the target node injects the received state information into the locally created instance of the role and sends another Report message to the source node indicating the state information was received and successfully processed. In summary, the source node's adaptation manager sends one StateTransfer message to the target node's adaptation manager, which in turn, sends two Report messages to the source node's adaptation manager if state information is to be transferred from source to target during the execution of the distributed adaptation operation. The overall procedure for a distributed adaptation operation with state transfer from source to target is depicted in Figure 4.7. The post-activation phase, which is also displayed in Figure 4.7 will be entered after the adaptation transaction was determined being executed successfully according to the previously discussed protocol mechanisms and not immediately after the pre-activation phase finishes as it might be understood from Figure 4.7.

Since the execution of distributed adaptation operations requires two adaptation man-



(a) Execution state chart for the source node of a distributed operation.

(b) Execution state chart for the target node of a distributed operation.

Figure 4.8.: Execution state charts for the source and target node of a distributed operation to cope with link failures.

agers to collaborate and exchange coordination messages in order to perform the adaptation operation, protocol behavior needs to be specified addressing the issue of link failures. In Figure 4.8, the state charts executed by the adaptation managers responsible for the source and target node's adaptations are depicted. In case of lost Report or StateTransfer messages, the respective adaptation manager utilizes the RequestReport message in order to obtain the missing progress information of the adaptation operation. The source node's behavior, which is depicted in Figure 4.8a, differs from the target node's behavior in regard of how not received Report messages are handled. Since the target node emits a Report message to all peers if the operation was performed successfully, the source node tries to obtain the required Report message in a three-step scheme, which is hierarchically structured because we assume the loss of coordination messages to occur infrequently and the protocol aims to reduce the amount of RequestReport messages and respective responses as a reaction to lost coordination messages. After a first timeout in which no Report messages were received, the source node's adaptation manager sends requests for the missing Report messages only to the target node's adaptation manager. As specified, the target node emits two Report messages to the source node: one Report message for the successful creation and binding of the role on the target side and one Report message for the successful injection of the received state information, which is also broadcast to all peers. Receiving a RequestReport message from the source node, though, the target node cannot distinguish which of those two reports is actually requested for which reason the target node will encapsulate two positive reports within a Report message in response if the execution of the adaptation operation has already been finished. Otherwise only one report is encapsulated in the

Report message, which indirectly indicates a lost StateTransfer message and the source node will immediately retransmit the local state information of the role if the Report message was received. If the Report message containing both reports was received, the source node simply considers the adaptation operation to be finished successfully. After another timeout, the source node sends a multicast request to all peers of the adaptation group. Ultimately, a broadcast request is sent to all adaptation managers involved in the execution of the adaptation transaction after another timeout. Received Report messages for the requested adaptation operation from one of the peers reached through the multicast or broadcast also indicates the adaptation operation to have finished successfully from the source node's adaptation manager's perspective. Otherwise peers would not be able to serve a Report message if the target node would have not broadcast a Report message as previously described. From the perspective of the target node's adaptation manager, only the Report message (no state information is supposed to be transferred) or the StateTransfer message can be lost on the channel. Consequently, the adaptation manager can only request the missing information from the source node's adaptation manager directly since no other adaptation manager involved in the adaptation transaction's execution can serve the desired progress information. The resulting state chart is depicted in Figure 4.8b. An escalation of four timeouts is chosen for the retrieval of missing state information to achieve the same maximum waiting duration as the source if all three escalation steps for lost report messages have been executed. From both the source and target nodes perspective, a TransactionRollback message will be issued if the respective Report or StateTransfer messages could not be obtained after a last timeout after issuing the broadcast request.

It can happen that the target node is able to obtain the required StateTransfer message from the source node's adaptation manager, but the other way around, the required Report messages cannot be obtained. Following the protocol, the source node's adaptation manager would eventually emit a TransactionRollback message after all timeout periods passed without being able to obtain the desired Report messages from the target side or one of the peers. The emitted TransactionRollback message is ignored by the peer adaptation managers if the respective adaptation operation has already been reported being executed successfully by the target node's adaptation manager. Preventing this contradicting behavior of the adaptation managers requires the TransactionRollback message to not only contain the identifier of the adaptation transaction but also the identifier of the concrete adaptation operation that caused the transmission of the message. The adaptation manager that send the TransactionRollback message in this case, awaits another timeout before local changes are actually reverted by entering the rollback state.

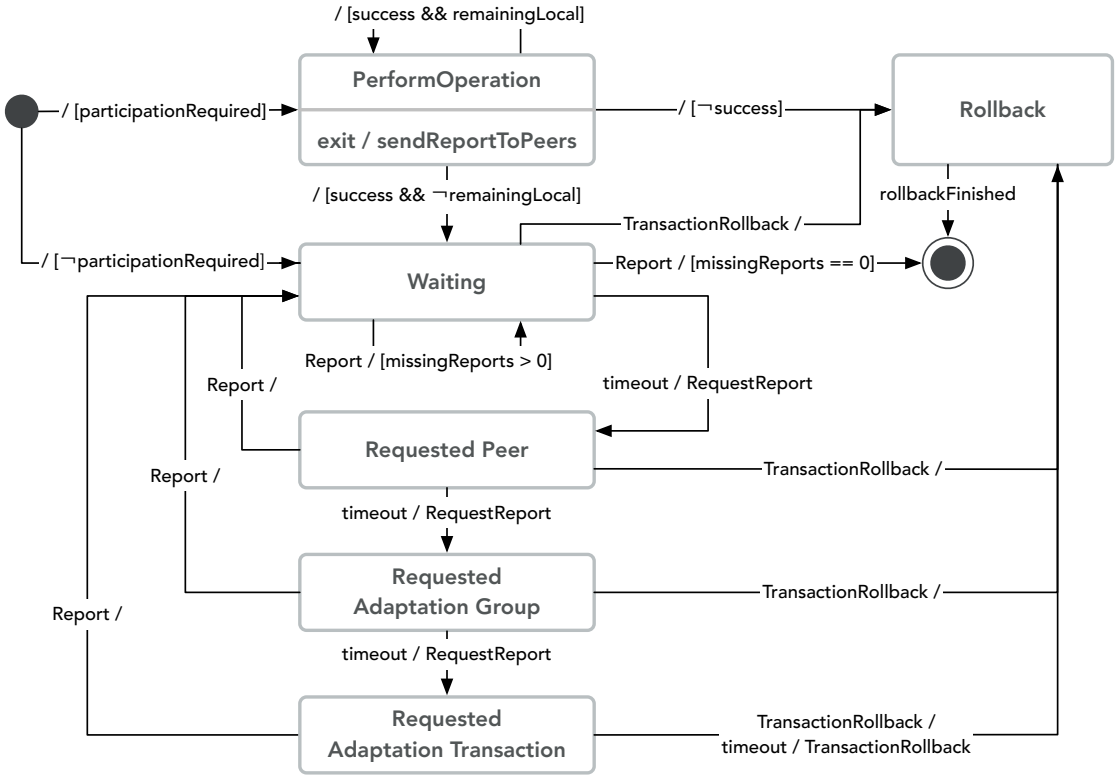


Figure 4.9.: State chart for the execution of an adaptation group by an adaptation manager.

Further details on the progress of the overall adaptation transaction will be given in the following, as the execution of adaptation groups is discussed.

Coordinating the Execution of an Adaptation Group

So far, the device-local execution of adaptation operations has been discussed as well as the coordination among two adaptation managers during the execution of a single distributed adaptation operation. The execution of adaptation transactions has been discussed as well as a sequential execution of adaptation groups until the adaptation transaction is eventually performed successfully or had to be rolled back. The discussion of the execution of adaptation groups in the following closes the apparent remaining gap between the execution of adaptation transactions and operations. As previously defined, an adaptation group contains adaptation operations configured with the same `Order` parameter and the execution order of adaptation operations within such a specific adaptation group is unrestricted.

Entering a new adaptation group for execution, the adaptation manager first determines whether any participation in the execution of the adaptation group is required, i.e.,

if the adaptation operation's source or target node's address property points to the node hosting the current adaptation manager. If no participation is required, the adaptation manager immediately enters the *Waiting* state. In this state a timeout is started in which Report messages from peers are expected to arrive. The number of Report messages expected to be received is determined by the adaptation manager based on the number and type of adaptation operations contained within the adaptation group independent of a requirement for the adaptation manager to collaborate. If the adaptation manager is required to participate in the execution of the adaptation group, it will enter the *PerformOperation* state and re-enters this state until no further local modifications of the local role-based application are required. A Report message will be sent to all peers after the successful execution of an adaptation operation that follows the previously defined execution schemes for local and distributed operations. Otherwise, a TransactionRollback message will be broadcast to all other adaptation managers, and the *Rollback* state will be entered, which matches the *Rollback* state for the execution of the adaptation transaction previously discussed.

The protocol specification to handle lost coordination messages during the execution of adaptation groups is displayed in Figure 4.9 together with the previously discussed execution states and resembles the approach used to cope with link failures for the execution of adaptation operations. After a first timeout, the adaptation manager responsible for the execution of the adaptation operation of which the progress information is missing is requested to retransmit the Report message. If this fails, i.e., if another timeout occurs, all adaptation managers involved in the execution of the current adaptation group are requested to transmit the missing progress information on the respective adaptation operation. If the adaptation manager that has received such a request has already received the progress information in question, an appropriate Report message will be sent to the requester. In a last attempt, all adaptation managers participating in the adaptation transaction are asked to transmit the missing progress information before the adaptation transaction will eventually fail. If the adaptation manager was not able to obtain the missing progress information, a TransactionRollback message will be broadcast to all peers resulting in a termination of the adaptation transaction and a rollback of all preliminary adaptations performed until that moment. A three-step approach to obtain lost coordination messages was chosen for the execution of adaptation groups as well in order to minimize the number of retransmission requests and actual retransmissions of the progress information assuming message loss to occur infrequently.

Each adaptation manager executes the protocol specification discussed above independently of other adaptation managers and solely relies on the exchanged messages to

make local decisions on how and when to proceed with the execution of the protocol specification. Consequently, the current local progress of the protocol execution may differ between adaptation managers, especially in the presence of randomly occurring link failures. The specification of the coordination protocol allows adaptation managers to proceed with the execution of a subsequent adaptation group as soon as all expected Report messages have been received. Evidently, this allows adaptation managers to proceed while others might still be waiting for lost Report messages. Adaptation managers that suffered from lost coordination messages might hence receive Report messages from subsequent adaptation operations contained in subsequent adaptation groups while they are still in the *Waiting* state. The affiliation of an adaptation operation's Report message to a given adaptation group can be easily determined with the adaptation operation's identifier contained in the Report message by the adaptation manager because this identifier is unique within the adaptation transaction and thus also within the set of adaptation groups. The adaptation manager still waiting for lost coordination messages will immediately proceed with the execution of the subsequent adaptation group upon receiving such a Report message because this message indicates that at least one peer could establish knowledge on the current adaptation group to be finished successfully rendering all other Report messages for the current adaptation group obsolete. A similar behavior is shown by an adaptation manager receiving a RequestReport message for an adaptation group it locally already considered successfully finished. In such a case, all Report messages of the locally executed adaptation group are transmitted in addition to the reports requested in the received RequestReport message. Similarly, if the aforementioned adaptation manager that is still waiting to receive Report messages for its current adaptation group, receives a RequestReport message for an adaptation operation in a subsequent adaptation group it has to execute an adaptation operation in, the current adaptation group will also be immediately considered successfully finished and the execute of the following group will be started.

A specific corner case still has to be considered with respect to an adaptation manager that became disconnected while trying to obtain progress information, but became reconnected when the TransactionRollback message is about to be delivered to the peers. Assuming the node of which the progress information could not be obtained was able to distribute the progress information to all other adaptation managers, the incoming TransactionRollback message will be ignored by the adaptation managers (remember, the adaptation operation's identifier is given within the rollback). In turn, the receiving adaptation managers transmit the latest Report messages to the sender of the TransactionRollback message or the TransactionActivation message if the adaptation transaction

was already finished successfully. Obviously, a TransactionRollback message will be sent by the other peers if the adaptation transaction has failed already for other reasons by the adaptation managers. Depending on the kind of received message in response to the TransactionRollback message, the rollback-initiating adaptation manager will either disregard its local decision and resume or finish the execution of the adaptation transaction or continue with the rollback of the temporary local changes. As previously mentioned, the adaptation manager that issued the TransactionRollback message awaits a timeout in order to decide how to proceed with the rollback. If the respective node is disconnected from the network for an amount of time much longer than the timeout period, the behavior of the remaining adaptation managers is described in Chapter 4.4.4.

During the execution of local parts of adaptation operations, the adaptation manager is able to detect adaptation failures based on the returned status information of the role runtime's provided interface operations (cf. Chapter 3.2.2 and Chapter 4.3). In such a case, the adaptation manager immediately terminates the execution of the adaptation transaction and broadcasts a TransactionRollback message to all adaptation managers involved in the execution of the adaptation transaction. Local operations that have been performed until that moment are reverted by every adaptation manager within the respective local role runtime in response. For example, if the execution of the `bind` operation failed, the previously performed `create` operation is undone by issuing a `remove` operation to the local role runtime. Similarly, already passivated roles will be reactivated. Thus, the role runtime is kept in a consistent configuration, which is the source configuration upon which all preliminary modifications have been performed.

If adaptation operations require a time longer than the timeout for their execution, i.e., if the passivation is too time consuming, incoming RequestReport messages will be answered with a Report message that has the respective progress flag set to indicate this adaptation operation is still going on, which will reset the escalation state of the requesting peers to *Waiting*.

4.4.4. A Note on Stopping Failures

We defined stopping failures as failures in which the adaptation managers shut down ungracefully while participating in the decentralized execution of an adaptation transaction. Addressing this issue in the coordination protocol is a cross-cutting concern affecting several parts of the protocol, which is the reason we focus on the issue separately in contrast to adaptation and link failures, which were incorporated in all previous discussions.

An adaptation manager that went down suddenly during the adaptation process appears as an unreachable adaptation manager for its peers. Other adaptation managers

will thus follow the previously discussed protocol in order to deal with the situation. In order to recover from a stopping failure, every adaptation manager maintains a log on the current progress of the execution state of the adaptation transaction, i.e., on received report messages for the current adaptation group as well as the state of local adaptations. The maintained log information enables the adaptation manager to replay the unfinished adaptation transaction if required. In any case, the restarted adaptation manager broadcasts a RequestReport message to all peers in order to obtain the execution state of the adaptation transaction. If the adaptation transaction was finished successfully, the peers will respond with the transmission of a TransactionActivation message or a TransactionRollback message, otherwise. In the former case, the adaptation manager replays all local adaptations using stored state information of remote roles if necessary. In the latter case, the adaptation manager just marks the adaptation transaction log as finished unsuccessfully and keeps the current configuration of the managed application unmodified.

Please note that the adaptation transaction can only finish successfully albeit an adaptation manager went down during the execution if all local participations of the respective adaptation manager could be finished ahead of the ungraceful shutdown of the respective adaptation manager. Otherwise, the remaining group of adaptation managers will always decide to terminate the execution of the adaptation transaction unsuccessfully and maintain the source configuration of the managed application.

Furthermore, restarting the adaptation manager or performing other self-healing mechanisms on the system in response to stopping failures or crashes of the managed application are out of the scope of this thesis. The mechanisms presented here solely serve the purpose to keep the managed application's configuration consistent during the adaptation process.

4.5. Update Execution of the Role-based Managed Application

The role runtime and the developed role life cycle support the introduction of previously unknown application behavior, i.e., roles, while the system is running without the need for the system to completely stop and restart. In [49], we described a mechanism to perform updates on the system using application scenario three described in the introduction of this thesis. This approach will be presented more generally in the following.

We reuse the idea of an adaptation operation including the set of required parameters, but refer to them as *Update Operations*. The number of operation types for update

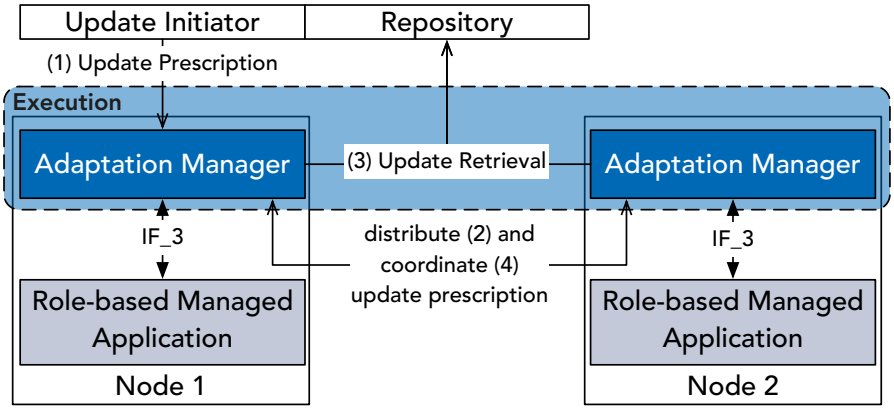


Figure 4.10.: Architecture to enable dynamic software updates in an exemplary IoT infrastructure.

operations is limited. It is possible to *Add*, *Remove* and *Exchange* behavior of the managed role-based application as a consequence of a software update. Those three operation types are sufficient to cover the basic scenarios in order to (1) add previously unknown behavior to the system, (2) remove behavior that is no longer required and (3) update existing behavior in order to, for example, fix a bug, respectively.

In order to perform updates of the role-based managed application, we expect an *Update Initiator* to be present in the system as well as a *Repository* from which new or updated role types can be downloaded. The system's architecture remains otherwise unchanged as it is illustrated by Figure 4.10. The update manager is responsible for generating an adaptation transaction, which contains the appropriate update operations, but may also contain additional adaptation operations in order to support and complement the system update. Thus, the update initiator is comparable to the planning component of the feedback loop, but different since human participation may be required in order to trigger the update process in contrast to the context-dependent adaptation triggered by the planning component, which is considered to be performed completely autonomously.

The adaptation transaction issued by the update initiator is executed in accordance with the coordination protocol presented in this chapter, but differs in some aspects. After the adaptation transaction has been distributed in the system (cf. Step 2 in Figure 4.10), role type information is retrieved from the repository by the adaptation managers affected by the update (Step 3). The interface between adaptation manager and role-based application is subsequently used to install and load the new or updated type information to the role runtime. In the following, the execution of the update operation, e.g., the addition, removal or exchange of roles, follows the execution scheme developed

Table 4.3.: Summary of the posed requirements and their fulfillment by the approach presented in this thesis.

Requirement	**Approach**	**Fulfillment**
Decentralized Execution	Coordination Protocol	✓
Stable Application State	Extension Run-time Model Role	✓
Link Failures	Dedicated Message in the Coordination Protocol	✓
Adaptation Failures	Coordination Protocol & Local Adaptation Interface	✓

in the previous sections for regular adaptation operations (Step 4).

Using the notion of *Roles* is not only a suitable abstraction for context-dependent and collaborative application behavior that allows a dynamic behavioral adaptation of the software system, but also enables the reusability of adaptation mechanisms to perform software updates at run time due to the inherent variability of the system.

4.6. Summary

The goals of a self-adaptive software system that are of concern within this thesis are the consistent adaptation of the managed application in response to changes in the computational environment of the system preventing inconsistent configurations and loss of application data. These goals are backed up by a set of requirements imposed on the envisioned adaptation execution in such a system, which are summarized in Table 4.3 including each requirement's degree of fulfillment. In this chapter, we developed a protocol that allows a distributed execution phase, which is represented by a set of *Adaptation Managers* located on every node of the system, of the feedback loop in order to perform adaptations without a central coordination component. Rather, the distributed adaptation managers are forced to collaborate using the devised protocol specification in order to perform the adaptation process. The developed role life cycle and related mechanisms of the proposed protocol allow the system to reach a stable state before any adaptations are performed, thus, fulfilling the second requirement posed to the self-adaptive software system. The protocol explicitly provides means to cope with random losses of coordination messages addressing the third requirement. The possibility to cope with adaptation and stopping failures represented by the fourth requirement are addressed in a similar manner by the protocol. Related to the goal of maintaining a consistent system configuration, the notion of an *Adaptation Transaction* was introduced in order to define an atomic and consistent execution of all adaptation operations specified within

this adaptation transaction. Being able to detect adaptation and stopping failures, the adaptation managers can roll back preliminary modifications, which keeps the system in the last known valid configuration, which is the source configuration. In summary, all requirements posed on the execution phase for a self-adaptive software system coined by a highly distributed and interconnected managed application and the necessity to perform several adaptations as one logical adaptation could be fulfilled by the approach presented within the last two chapters of this thesis.

In the following, we will outline how the research questions raised in the beginning were addressed within the presented approach.

1. *How can a stable application state in a distributed application be reached in order to allow for multiple adaptations being performed on multiple computing devices simultaneously or in an otherwise coordinated manner?*
   In this work, the *quiescence* criterion was used to describe a stable application state for the role-based managed application. The criterion itself is applied to each role instance, since a role represents the dynamic part of the system that is subject to run-time adaptations and must therefore reach a state in which no further computations are performed. Since roles may collaborate with other roles as well in order to provide a certain functionality, the decentralized set of adaptation managers is used to coordinate the process of reaching a quiescent state for a role across device borders. In order to support the notion of quiescence and a stable state in general, a life cycle for roles as run-time entities was developed to represent the current execution state of a role instance. Without the knowledge about a role's momentary execution state and collaborations, a consistent adaptation of the distributed role-based software system would be infeasible. With respect to the proposed adaptation models, the current approach can be classified as a *guided adaptation* because the application behavior is purposefully restricted by forcing roles into a quiescent state. Other and less disruptive criteria might be employed though to ensure a stable application state before the actual execution of an adaptation process. We are convinced that an *overlap adaptation* can be achieved with the notion of a *Role* as abstraction for dynamic system artifacts, but leave a detailed investigation to future work.

2. *How can a decentralized management and coordination of an adaptation process composed of multiple interdependent adaptations be realized that*
   a) copes with the loss of protocol messages and network partitioning, and
   b) copes with node failures or adaptation errors during the adaptation process?

With respect to the first requirement, namely a decentralized execution of adaptations at run time in order to avoid the issues of a performance bottleneck and of a single point of failure, a protocol was developed in response to the second research question. The proposed decentralized coordination protocol is composed of a behavioral description for the distributed adaptation managers and of a set of well-defined protocol messages. The former allows the management of an adaptation transaction's execution without a central management instance whereas the set of well-defined protocol messages is used to disseminate progress information among the adaptation managers facilitating local decision-making processes in order to deduce how and when to proceed with the execution. Hence, the protocol messages furthermore control the state transitions of the behavioral protocol of the adaptation managers. The issue of unreliable communication channels and adaptation errors is addressed by specific protocol messages and behavior whereas the stopping failures are addressed by a maintained log of the adaptation transaction's execution state complementing the local execution of the coordination protocol by each adaptation manager.

3. *What is a suitable abstraction to describe the adaptations supposed to be performed on the managed application platform-independently and is that description able to support the execution of changes?*
   We use the *Role* concept as an abstraction to describe the context-dependent and collaborative parts of the managed application that expose a specific behavior to the system. Modifying the set of played roles or the interactions among roles causes behavioral adaptations in the system. The utilization of roles covers fundamental operations such as the addition and removal of system behavior as well as the modification of collaborations between system entities. In addition to these operations, which are already used by existing approaches, it was possible to easily integrate semantically more powerful operations to the self-adaptive software system, e.g., the exchange, migration or cloning of roles including their internal state information. The set of adaptation operations proposed within this thesis and based on the proposed *Role* features covers all existing operations providing structural self-adaptations and allows for more semantically expressive operations, too. Furthermore, the presented coordination protocol also relies on the notion of *Roles* to perform changes at run time, i.e., the life cycle to determine a stable application state supporting the modification of the managed application. In summary, *Roles* are used to describe adaptations context-dependently, but also to implement context-dependent and collaborative behavior of the managed application at run

time, which eases the adaptation process of the coordination protocol and aids in maintaining a consistent configuration during the adaptation process without relying on a specific implementation-platform, which is beneficial for the utilization of the proposed approach in heterogeneous system environments. Regarding the last research question, *Roles* can be considered a suitable abstraction for the desired purpose in self-adaptive software systems.

5. Implementation & Evaluation

Having discussed and summarized the developed concepts addressing the posed requirements and research questions, this chapter gives insight into the performance of the proposed solution. The evaluation of the developed concepts comprises a formal validation of the presented coordination protocol using means of model-checking as well as a performance evaluation of a prototypical implementation of the coordination protocol in a virtualized and emulated environment. First of all, some details on the prototypical implementation, which is used to conduct the performance measurements of the coordination protocol in the emulated setup, will be given. Since this approach uses the notion of *Roles* as general abstraction in order to describe and conduct adaptations, implementation details on the managed application will be given first in Chapter 5.1. Subsequently, the overall architecture of the prototypical implementation, which differs to a certain degree from the proposed conceptual architecture, will be discussed in Chapter 5.2. In this section, only the most essential parts of the implementation of the coordination protocol within the *adaptation managers* will be outlined. The emulation of the coordination protocol in a virtualized environment will be discussed in Chapter 5.3. First, the emulation setup will be presented and the approach taken in order to obtain measurement data to evaluate the protocol's performance will be outlined. In a second step, the emulated scenarios in which the coordination protocol is executed will be presented and discussed. Third, the obtained results will be interpreted using a three-step approach: First, the properties of the coordination protocol that are supposed to be evaluated will be presented and expected results based on the chosen scenarios and the protocol design will be discussed. Second, the obtained results will be interpreted and discussed with respect to the expectations and requirements as well as the research questions. Finally, the presented emulation and the obtained insights will be briefly summarized. In Chapter 5.4, the approach taken to formally validate the developed coordination protocol will be presented before the findings will be summarized in Chapter 5.5.

5.1. The Role-based Managed Application

The emulation of the coordination protocol in a virtualized setup requires the presence of a managed application. Since the approach relies on the notion of *Roles* to describe and execute adaptations performed upon the managed application, the usage of a role-based programming language or framework as foundation for the implementation of the managed application is preferred. In the following, approaches suitable to serve as foundation for the managed application will be discussed.

In Table 3.2 on Page 50, the list of features compiled by Kühn et al. [28], a role-based software systems may support, is depicted. The *Role* features, the approach presented in thesis relies on, were also indicated in Table 3.2. Some of those features, e.g., Features #1 or #3, have to be supported directly by the role runtime whereas other features, e.g., Feature #6, can also be ensured by the adaptation management supervising local adaptations of the role-based application. Any programming language level approach, though, that meets the stated requirements conceptually and at run time can be used as implementation platform for the managed application of the self-adaptive software system.

Besides the identified mandatory role features, the following requirements posed on a role-based runtime were derived from the previous description of the adaptation interface between adaptation management and managed application including the notion of a life cycle for role instances (cf. Chapter 3.2). This life cycle, introduced in Chapter 3.2.1 on Page 38, is the first requirement, denoted as *RRQ 1*, a role runtime needs to support. Such a life cycle is required in order to determine a stable state for any given role instance that is subject to context-dependent adaptations. The support of the adaptation interface between adaptation management and managed application discussed in Chapter 3.2.2 on Page 41 can be considered a second requirement (*RRQ 2*). More specifically, access to operations that allow the adaptation management to create, delete, bind and unbind roles and players is required since the adaptation management has to be able to modify the relations between roles and players dynamically at run time. Additionally, internal state information must be obtainable, which is required in order to fully support the introduced distributed adaptation operations. The support of the role runtime to seamlessly allow the collaboration with remote roles is a third, but not a mandatory requirement (*RRQ 3*). Since our approach not only covers the behavioral and context-dependent natures of roles, but also their collaborative nature, a distributed runtime supporting such role collaborations across device borders will be a mandatory requirement in the future.

The role-based programming languages and runtimes discussed in [28] have been in-

vestigated as possible candidates to serve as foundations for the managed application in our approach, but neither approach allows the required flexibility in terms of the modification of the binding between roles and players to allow for behavioral adaptation, which is the main concern for the envisioned approach in this thesis, at run time. Only two approaches, namely SCROLL [30] and LyRT [42], fully support the degree of variability between roles and players, required by the approach proposed in this thesis. Both approaches are not part of the initial list presented in [28], but were developed later. Subsequently, both approaches were measured against the required and optional role features posed by our solution. In Table 5.1 the supported role features of both LyRT and SCROLL are measured against each other. In order to discern, which of the both runtimes fits the needs of our approach of a role-based runtime as foundation of a managed application better, the role features supported by our approach are also highlighted in Table 5.1. Role features previously marked as not considered, thus not relevant for our approach, have been omitted in Table 5.1 for a better overview. Evidently, both LyRT and SCROLL support the role features required by the adaptation management to a high degree. Feature #2 is not supported by both LyRT and SCROLL, which is also reflected in the missing fulfillment of both approaches for *RRQ 3*. Since all other required role features are supported by both approaches, LyRT was chosen as execution environment for the implementation of the managed application, since the existing Java implementation was the easiest to extend with the sole additional requirement to implement a communication interface that allows the adaptation of a LyRT-based application by the adaptation management. In the remainder of this section, we focus on the extensions made to LyRT with respect to the missing compatibility to previously posed RRQ 1 through RRQ 3.

Regarding RRQ 2, LyRT already offers all necessary methods to manipulate the role-player relation dynamically at run time through a class called `RegistryManager`, which is responsible for the management of these relations and the method dispatch between roles and players. Extending the base class `Role` within LyRT in order to write and read the internal state information of the given role instance was the only extension that had to be implemented for RRQ 2 to be fulfilled. Each subclass implemented as a role that is supposed to be adapted context-dependently therefore has to overwrite these two methods, which is a reasonable requirement for adaptation developers since the transmission is application-specific and might also only comprise a subset of the actual internal state of a role instance.

As it was previously outlined in Chapter 3.2.1 and Chapter 4.3, a run-time life cycle for *Roles* was developed to ensure the consistent adaptation through roles reaching a stable

Table 5.1.: Reduced list of role features by Steimann [40] and Kühn [28] comparing features of role-based runtimes with our approach.

#	**Feature**	**Our Approach**	**SCROLL [30]**	**LyRT [42]**
1.	Roles have properties and behaviors	✓	✓	✓
2.	Roles depend on relationships	✓	✗	✗
3.	Objects may play different roles simultaneously	✓	✓	✓
4.	Objects may play the same role (type) several times	✓	✓	✓
5.	Objects may acquire and abandon roles dynamically	✓	✓	✓
6.	The sequence of role acquisition and removal may be restricted	✓	✗	✓
7.	Unrelated objects can play the same role	✓	✓	✓
8.	Roles can play roles	✗	✓	✓
9.	Roles can be transferred between objects	✓	✓	✓
11.	Features of an object can be role-specific	✓	✓	✓
12.	Roles restrict access	✗	◆	✗
14.	An object and its roles share identity	✗	✓	✓
15.	An object and its roles have different identities	✓	✓	✓
17.	There may be constraints between relationships	✗	✗	✗
19.	Roles depend on compartments	✓	✗	✓
20.	Compartments have properties and behaviors	✗	✓	✓
21.	A role can be part of several compartments	❊	◆	✓
22.	Compartments may play roles like objects	✗	✓	✓
23.	Compartments may play roles which are part of themselves	✗	✓	✓
26.	Compartments have their own identity	✓	✓	✓
RRQ 1	Life cycle support for role instances	fav	✗	✗
RRQ 2	Provision of methods required by IF_3	man	◆	◆
RRQ 3	Retrievable and modifiable role collaborations	fav	✗	✗

Supported: ✓ | Partially Supported: ◆ | Not Supported: ✗ | fav: favorable | man: mandatory

application state prior to any ongoing adaptations from a conceptual perspective. During the adaptation process, *Passivate* messages are used between adaptation managers to facilitate reaching a stable application state among the remote instances of the managed application. More specifically, *Passivate* messages are used to gracefully stop ongoing collaborations of the role instance involved in the execution of the current adaptation operations, and the remote roles involved in the collaboration. The exchanged *Passivate* messages are locally translated by the adaptation manager to *Passivate* methods, which trigger the transition of the role to the *Passive* state in its life cycle and which are provided by the previously discussed interface of the role-based runtime. While all protocol-related prototypical implementations follow the developed state chart and make use of the specified interfaces, no actual means were implemented in LyRT to prevent method dispatches from happening when a role is supposed to be passive. Since the adaptation-specific parts of the coordination protocol, i.e., the coordination of the adaptation steps and the detection and handling of the assumed error models (cf. Chapter 4.1), are subject to the evaluation presented in this chapter, the assumption that roles at run time behave according to the proposed life cycle is sufficient. Future work, however, that also considers reaching a stable application state prior to the actual adaptation process will have to make dedicated extensions to LyRT in order to support RRQ 1.

Following this rationale, no additional implementations were made within LyRT to establish support for collaborating roles located on mutually remote devices. Consequently, RRQ 3 was also not necessary to be implemented for the conducted evaluation.

5.2. The Decentralized Adaptation Management

Conceptually, the adaptation management is responsible for the external adaptation of role-player relations within the local role-based runtime. Since LyRT only supports intrinsic adaptations, i.e., role bindings are established and released through code statements, the runtime had to be extended to support external adaptations triggered and coordinated by the adaptation manager co-located on the respective node. As a consequence, the conceptual adaptation manager (cf. Figure 4.1 on Page 54) was split into two components. The *adaptation manager* implements the coordination protocol and manages the decentralized execution of adaptation transactions. The *management endpoint* serves as extension of the adaptation management and is incorporated into the role runtime providing a communication interface to the adaptation manager. This results in two executable software artifacts that follow the external control approach, which was already used conceptually to describe the system architecture of the self-adaptive soft-

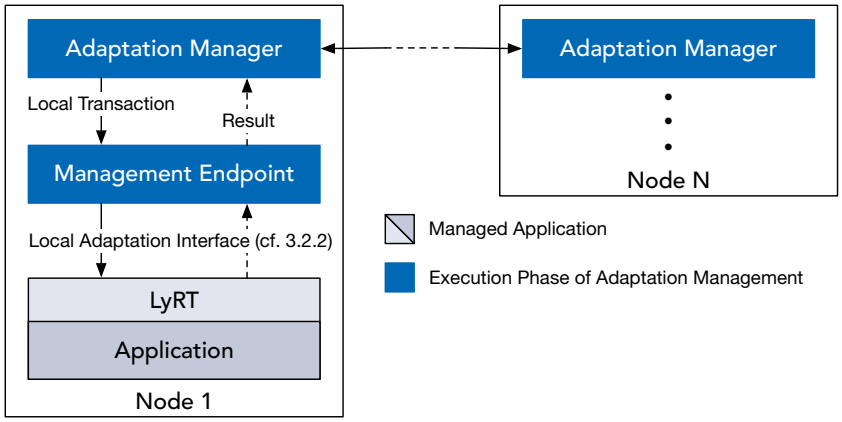


Figure 5.1.: Overview of system artifacts representing the prototypical implementation of the approach.

ware system. The adaptation manager, which is the first software artifact, implements the proposed coordination protocol and thus represents the distributed and decentralized execution phase of the feedback loop. The managed application composed of an arbitrary application implemented using the LyRT runtime and the management endpoint denotes the second software artifact. An overview of the components' part of the implementation including their communication relations is depicted in Figure 5.1.

The *management endpoint* does not only provide a communication endpoint to the adaptation management, but also implements the translation of adaptation operations presented earlier in this thesis to the interface operations provided by the role runtime, thus providing the implementation for *IF_3*, and reports the execution results to the adaptation manager. This execution of adaptation operations affecting role-player relations on the local device is implemented as *adaptation transaction* through the communication interface between adaptation manager and management endpoint. The execution of an adaptation group by an arbitrary adaptation manager exemplifies the ongoing communication between the two artifacts. In a first step, adaptation operations within this group are determined that affect role-player relations of the local role runtime. These adaptation operations are wrapped in a Transaction message, which is sent to the management endpoint. Within the management endpoint, the adaptation operations encapsulated in the received adaptation transaction are executed in accordance with the mappings of adaptation operations to local interface operations provided by LyRT. This mapping was summarized in Tables 4.1 and 4.2 on Pages 68 and 69, respectively. A Report message is sent from the management endpoint to the adaptation manager in order to report the successful execution of the respective adaptation transaction that represents

the currently executed local adaptations. The adaptation manager component maintains a mapping of the adaptation transactions representing the locally executed adaptation operations and the globally executed adaptation transaction. The management endpoint also determines the type of the adaptation operations and executes them appropriately, e.g., if it is the source node of a `migrate` operation, the role will be passivated and state information will be retrieved. The adaptation manager, however, keeps track of the overall progress of the distributed adaptation operation since the management endpoint is only aware of its local knowledge on the operations' execution progresses. In this regard, the adaptation manager requests the retransmission of Report messages from the migration's target node in the case of lost coordination messages or issues the activation of changes or the rollback in either case in accordance to the specifications of the coordination protocol to the management endpoint.

All executable software artifacts are Java implementations. Since LyRT is a Java-based role runtime, implementing both adaptation manager and management endpoint as Java applications, too, allowed for a high degree of code reuse through shared libraries for the exchange of messages between adaptation managers, but also between adaptation manager and management endpoint. The communication among all executable software artifacts is based on the standard UDP implementation of Java 1.8. Adaptation managers use two distinct ports to exchange transaction and execution control messages among each other, which is a decision made to ease the automated execution of adaptation transactions during the protocol emulation. We will outline and discuss the details of that implementation decision in the following section.

A configurable part of the coordination protocol, which may also be application-specific, is the timeout to determine lost coordination messages exchanged between adaptation managers. In the implementation used for the emulation, this timeout was implemented with 2.5 seconds for each escalation step.

When adaptation managers report the progress information of locally performed adaptations, more messages are sent within a few milliseconds than are processable in time by the prototypical implementation. The limitation becomes increasingly obvious with a growing number of nodes participating in the execution of adaptation transactions and results in a behavior that adaptation managers interpret as message losses and act accordingly, which negatively affects the emulation of the coordination protocol. Details on the observed behavior within the emulation process itself will be given in Chapter 5.3.4. To partially reduce the observed execution behavior for a majority of test cases, at the end of the execution of every adaptation group, i.e., when an adaptation manager detects its currently executed adaptation group to be finished, an additional Report message will

be sent to all peers. These additional messages sent to all peers report the progress information of all locally executed adaptation operations within the just-finished adaptation group. These additionally sent messages, however, are not part of the original protocol specification.

In order to evaluate the protocol, which will be discussed in detail in the following section, the execution flow of the protocol needs to be logged to be available for the subsequent evaluation. This requires each adaptation manager involved in the execution of adaptation transactions to log information about received messages and when the execution of an adaptation group was started and finished. Consequently, each adaptation manager maintains a log file in which the received messages as well as the execution process were persistently stored including timestamps when a significant event such as a received coordination message or the beginning of the execution of a new adaptation group occurred. Since this logging constitutes a writing operation on the hard-drive of the system, typical delays and inaccuracies within the range of a few milliseconds at most are to be expected.

5.3. Emulation of the Coordination Protocol

This part of the evaluation is concerned with the scalability and performance of the coordination protocol. The significant properties of the protocol that are supposed to be evaluated by the emulation are (1) the overall execution time of an adaptation transaction, also referred to as *adaptation duration*, (2) the rate of successfully executed adaptation transactions for varying degrees of link failure rates; and (3) the time a role that is affected by a distributed adaptation operation is unavailable during the adaptation process, which is referred to as *downtime (of the role)*. The first two properties immediately address the performance of the coordination protocol, especially with respect to the handling of lost coordination messages during the protocol's execution. The latter property is significant with respect to self-adaptive software systems in general since the projected unavailability of system behavior during an adaptation process is an important measure for the applicability and performance of a self-adaptive software system. All properties will be evaluated for varying *system sizes*, which denotes the number of adaptation managers involved in the execution of an adaptation transaction.

In this section, the emulation setup will be discussed first. A brief outline of the infrastructure and tools will be part of this discussion as well as the steps taken to obtain measurement data and to prepare assessable results. In a second part, the results obtained through the emulation will be presented and critically discussed. At this point,

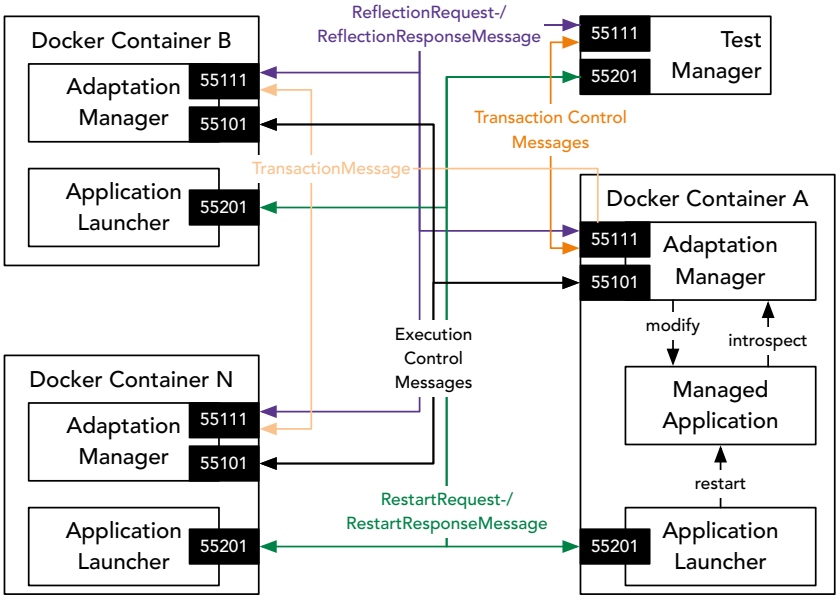


Figure 5.2.: Setup of the emulation environment depicting Docker containers, communication channels and deployed components to automatically execute adaptation transactions.

different scenarios used to asses the different properties and measures of the protocol will be outlined in preparation to the discussion of the final results. Finally, the results will be briefly summarized.

5.3.1. General Emulation Setup

In order to emulate the protocol behavior and to assess the scalability of the approach, several executions of adaptation transactions were performed in a virtualized environment. Docker[1] was used to set up the emulation environment for the conducted experiments. A Docker container in this setup represents a node of the self-adaptive software system hosting an instance of the prototypical implementations of the *adaptation manager* and the *managed application* each. Additionally, a third component was deployed in every Docker container to support the automated execution of adaptation transactions, which is referred to as *application launcher*. The overall emulation procedure for a test setup was managed and executed by a custom-made tool referred to as *test manager*, which is always deployed on the host system of the Docker containers and thus, neither deployed in a Docker container nor part of a node that is affected by an adaptation transaction. All components involved in the emulation of the protocol's execution and the communication channels between them are depicted in Figure 5.2 and will be briefly

[1] `https://www.docker.com`

explained in the following.

Adaptation Manager The adaptation manager is responsible for the execution of an adaptation transaction and implements the decentralized coordination protocol. Implementations diverting from the actual specification of the coordination protocol have been discussed in the previous section.

Managed Application The managed application is implemented using LyRT [42] and provides a random set of roles and players between which a *play* relation can be established and released dynamically at run time. A subset of roles and players was always bound immediately after the application launches to enable `Remove` and `Migrate` adaptations immediately to be performed. The application is generic enough to cover typical adaptation scenarios implied by the introductory application scenarios, but does not follow a certain scenario itself. The sole purpose of the managed application in the emulation is the provision of a role-based application upon which adaptations can be performed at run time.

Test Manager The test manager has several tasks in the emulation of the coordination protocol's execution. First of all, the test manager automatically generates adaptation transactions based on given parameters that can be set in the beginning of the emulation. These parameters include the type and number of adaptation operations that ought to be within the generated adaptation transaction as well as the number of times the generated adaptation transaction is supposed to be performed, i.e., the number of *runs*, and the *initial receiver*, which is the Docker container (adaptation manager) the adaptation transaction is sent to before it is automatically distributed from the initial receiver among all other nodes. Furthermore, all nodes part of the system need to be specified as a parameter as well to enable the test manager to communicate with the application launchers deployed to the respective containers throughout the repeated emulation process. Second, the test manager monitors the execution of an adaptation transaction using a subset of the coordination protocol's transaction control and coordination control messages. Third, the test manager collaborates with the application launchers, which are deployed on each Docker container, to restart the managed application after every execution of an adaptation transaction and starts a new run.

Application Launcher The successful execution of an adaptation transaction on the distributed managed application leaves the application in a state different from the initial source configuration on which the adaptation transaction is generated. In

order to achieve comparable results for each run of the adaptation transaction's execution, the adaptation transaction has to remain fix. The app launcher is responsible to terminate and restart the local process within the Docker container that represents the managed application (the adaptation manager is kept alive, in contrast, across several runs) and collaborates with the test manager to help coordinating the execution of subsequent runs of the same adaptation transaction.

In the following, we will refer to the term *experiment* as the repeated execution of a configured adaptation transaction for which the number of nodes that participate in the execution of the adaptation transaction remains unchanged. Different experiments vary with respect to the system size, i.e., the number of involved local adaptation managers in the adaptation transaction's execution, and the number and type of adaptation operations grouped together within an adaptation transaction.

A first step in each experiment is the generation of a random adaptation transaction, which is done automatically by the test manager based on the specified parameters. Since adaptations in the role-based application are carried out on the instance level, the test manager needs to obtain the concrete identifiers of roles, players and compartments within the distributed managed application. Solely for the purpose of obtaining the current run-time model of the role-based application, a *RequestReflection* and *ResponseReflection* message was introduced for the communication between test manager and adaptation manager. The former message is used by the test manager to request the application's run-time model and the ResponseReflection message is used by the adaptation manager to transfer the obtained information. The adaptation managers use a reflection interface provided by the managed application's management endpoint, which has not yet been discussed. Upon receiving a RequestReflection message, the management endpoint retrieves the current role-player bindings by querying the lookup table maintained by LyRT [42] and returns the obtained run-time model of the managed application to the adaptation manager using a ReflectionResponse message, which is then simply forwarded by the adaptation manager to the test manager. Since the managed application is restarted after each execution of an adaptation transaction, this step is required initially to generate a random adaptation transaction in accordance with the specified parameters and before each subsequent run of an experiment's execution since the instance identifiers generated by Java and utilized by LyRT change after every restart.

As soon as the adaptation transaction was generated, the test manager sends a Transaction message to the node that was specified as initial receiver through port 55111. Messages transmitted using this port are not subject to message loss during the execution of experiments, which eases the distribution of transaction control messages discussed in

Chapter 4.4.1. The initial receiver also uses this port to distribute the received adaptation transaction for the same reason. Since the execution of an adaptation transaction using the specified coordination protocol is supposed to be evaluated and not the distribution of an adaptation transaction, the simplification is reasonable within the scope of this evaluation. Furthermore, the distribution of adaptation transactions might not even be required if a distributed planning approach, such as DecAp [32] is used to plan out the adaptation transaction, which would then immediately be present on the respective nodes. Subsequent execution control messages, such as Report, StateTransfer or RequestReport messages, which were introduced in Chapter 4.4.1, are exchanged through port 55101, which is prone to message loss. As soon as the adaptation transaction fails or succeeds, either an TransactionRollback or TransactionActivation message, which are both transaction control messages and thus distributed through the reliable port 55111, are sent to the test manager, respectively. The current run is thus considered finished and the test manager requests the application launcher to restart the managed application using the RestartRequest message. The application launcher notifies the test manager on the successful restart of the managed application through responding with a RestartResponse message to test manager. As soon as all responses have been received from the application launchers, the test manager will continue the experiment with the next run following the described procedure. The exchanged messages used by the involved components and the respective ports those messages are exchanged through are also depicted in Figure 5.2.

Some performed runs turned out to be invalid for the intended measurements, which is the case if log information could not be obtained to calculate the adaptation duration for the respective node. Furthermore adaptation transactions that had to be rolled back because of local adaptation failures that had not been intended to happen, are also not considered for the evaluation of the protocol's performance. In both cases, the respective run was not considered for the evaluation and was repeated instead. All experiments have been repeated until a total of 100 runs was reached, which means invalid runs are not considered for the result set.

5.3.2. Data Acquisition

During the execution of an adaptation transaction, each adaptation manager logs progress information of the current execution state and writes the results on the persistent storage of the node. We utilize this log information that also contains a timestamp when the respective log message was issued to obtain information on the emulation properties, which were introduced in the beginning of this section. The important log information

we require though, are the points in time when an adaptation transaction is received by the respective adaptation manager and when the execution of the adaptation is started. Since the adaptation transaction is forwarded by the first receiving adaptation manager, each adaptation manager will evidently commence the adaptation process at a different point in time. The initial receiver in particular will start the execution of the adaptation transaction last since the adaptation process will only be started by that adaptation manager if all TransactionAcknowledge messages were received from the peers. The peers, in contrast, will start immediately with the execution of the adaptation transaction since they do not have to forward the received adaptation transaction. This is possible because it was assumed in the setup that the test manager is never deployed to a node, i.e., Docker container, affected by the adaptation process. Consequently, if an adaptation transaction is received from an IP address that is also specified in at least one adaptation operation, an adaptation manager can decide not to have to forward the received adaptation transaction. This decision, of course, is implementation specific and not prescribed by the proposed coordination protocol. Furthermore, information about the point in time when the execution of a specific adaptation group, i.e., of all operations denoted by the same `Order` parameter, is started and finished is obtained through the log information.

The last log information used to assess the performance of the coordination protocol is the point in time at which a node either detects the successful execution of the adaptation transaction locally or at which point in time a TransactionActivation message is received. Both events denote a successful execution of the adaptation transaction. Information about an unsuccessful execution is obtained in a similar manner since each adaptation manager logs the detection of a rollback based on unobtainable progress information for a given adaptation operation. Similarly, the receipt of the TransactionRollback message from a peer adaptation manager is logged.

After the experiment's execution, the logged progress information was obtained from the Docker containers and further processed using *evaluation scripts* to automatically calculate the desired values for adaptation duration, success rate and downtime of involved roles.

5.3.3. Emulated Experiments

As previously discussed, a Docker container hosts an instance of the managed application and the execution part of the adaptation management represented as adaptation manager, which implements the coordination protocol presented within this thesis. The performance assessment of the coordination protocol is aligned along three variables: the

system size, the composition of the executed adaptation transaction, and the rate of message losses occurring during the adaptation process. The *system size* denotes the number of nodes in the system that are affected by a given adaptation transaction. Evidently, a node participates in the execution of an adaptation transaction if a role is supposed to be added to or removed from a player located on the respective node. Similarly, a given node participates in an adaptation transaction if the node's adaptation manager is responsible for the execution of either the source or target node's adaptation steps of a distributed adaptation operation. The *message loss rate* describes the percentage of random drops of protocol messages exchanged between adaptation managers during the execution of the adaptation transaction. The *composition of adaptation transactions* finally describes the `Type` of adaptation operations contained within the adaptation transaction. In the following, we will focus on the different compositions of adaptation transactions and briefly discuss the rationale to choose those compositions. Each composition will be executed in combination with different system sizes and message loss rates.

The first three experiments are aligned to the introductory application scenarios, which results in three different compositions of adaptation transactions. Based on application scenario one (autonomously driving cars), the first composition contains only locally executed `Exchange` operations to simulate the exchange of the cars' driving behaviors in a coordinated manner in response to changed weather conditions. The second composition simulates the rescue-robot scenario and contains only `Migrate` operations, which have to be executed by two adaptation managers, i.e., source and target node are different. The third and last application scenario addresses the evolution of a running software system. This is simulated through adaptation transactions that solely contain `Add` and `Remove` operations to represent the removal of deprecated and the introduction of new system behavior. An adaptation manager is assumed to be responsible for the local execution of exactly one adaptation operation in all three compositions. This restricts the size of an adaptation transaction to the system size, i.e, for the first and third composition, the number of adaptation operations is equal to the number of nodes affected by the execution of the adaptation transaction. In the second composition, only half the number of adaptation operations compared to the number of nodes affected by the execution of the adaptation transaction are specified because each node either executes the target or source node's part of the distributed adaptation operation.

Since the previous set of experiments only contains homogeneous sets of adaptation operations with respect to the way these adaptation operations are executed according to the protocol definition, additional experiments were conducted containing a mixture of `Add`, `Remove` and `Migrate` adaptation operations. Since those three types cover the

entire protocol specification of the execution of adaptation operations, it is sufficient to focus on these three during the emulation of the coordination protocol. In these additional experiments, the load on the adaptation managers was also increased, which means adaptation transactions contained up to five times more adaptation operations than nodes in the system. The type of adaptation operations was evenly distributed among `Add`, `Remove` and `Migrate` operations.

For all experiments, the generated adaptation operations part of an adaptation transaction were distributed randomly across at most ten adaptation groups, i.e., the value for the `Order` parameter of each adaptation operation was within the range of $\{0..9\}$.

In contrast to the previous experiments, which were coined by a random distribution of adaptation operations across at most ten adaptation groups, a last set of four further experiments was set up in which all adaptation operations were performed in parallel, i.e., all adaptation operations were configured with an `Order` parameter of 0. The represented compositions in these experiments reflect the previously discussed ones and contain adaptation transactions composed of only `Add` and `Remove` operations, of only `Migrate` operations, and of an evenly distributed mixture of the three adaptation operation types with differing total numbers of adaptation operations within the adaptation transactions.

An overview of all previously described experiments is given in Table 5.2 including the different system sizes, the size of the adaptation transaction as well as the respective composition of the adaptation transaction. In order to estimate the coordination protocol's behavior in environments coined by unreliable communication channels, each experiment was repeated six times using different degrees of message loss rates, namely $\{0, 1, 5, 10, 25, 50\}$ percent of transmitted messages were dropped. An *iteration* of an experiment denotes its repeated execution using a specific message loss rate until 100 runs of the respective experiment have been completed.

Evidently, the chosen compositions for the adaptation transactions in combination with the number of adaptation operations executed within a given composition results in four distinguishable classes of experiments, which we will refer to as *cluster*s in the following (cf. *TS/SS* column in Table 5.2). The first cluster is coined by a *TS/SS* ratio of $1 : 2$ and comprises the experiments UC 2 and P1, which both only contained distributed migrate adaptation operations on the respective number of nodes with each node executing either the target node's or the source node's part of exactly one adaptation operation only. The second cluster is coined by a *TS/SS* ratio of $1 : 1$ and comprises experiments UC 3 and P2, which both only executed exactly one local adaptation operation (`Add` or `Remove` randomly distributed) on each node. We include experiment UC 1, coined by

Table 5.2.: Overview of the experiments used to emulate the execution of the developed coordination protocol.

#	Experiment	SS	TS/SS	Adaptation Operation Distribution			
				Add	Remove	Migrate	Exchange
1.	Use Case 1 – UC 1	4	1/1	0	0	0	4
2.		10		0	0	0	10
3.		20		0	0	0	20
4.		30		0	0	0	30
5.	Use Case 2 – UC 2	4	1/2	0	0	2	0
6.		10		0	0	5	0
7.		20		0	0	10	0
8.		30		0	0	15	0
9.	Use Case 3 – UC 3	4	1/1	2	2	0	0
10.		10		5	5	0	0
11.		20		10	10	0	0
12.		30		15	15	0	0
13.	Work Load A – WLA	4	2/1	3	3	2	0
14.		10		7	7	6	0
15.		20		14	13	13	0
16.		30		20	20	20	0
17.	Work Load B – WLB	4	5/1	7	7	6	0
18.		10		17	17	16	0
19.		20		34	33	33	0
20.		30		50	50	50	0
21.	Parallel 1 – P1	4	1/2	0	0	2	0
22.		10		0	0	5	0
23.		20		0	0	10	0
24.		30		0	0	15	0
25.	Parallel 2 – P2	4	1/1	2	2	0	0
26.		10		5	5	0	0
27.		20		10	10	0	0
28.		30		15	15	0	0
29.	Parallel 3 – P3	4	2/1	3	3	2	0
30.		10		7	7	6	0
31.		20		14	13	13	0
32.		30		20	20	20	0
33.	Parallel 4 – P4	4	5/1	7	7	6	0
34.		10		17	17	16	0
35.		20		34	33	33	0
36.		30		50	50	50	0
37.	Adaptation Failure – AF	10	4/5	2	2	2	2

SS: *System Size* denotes the number of notes affected by an adaptation transaction | TS: *Transaction Size* denotes the number of adaptation operations within an adaptation transaction.

locally executed `Exchange` adaptation operations in this cluster as well since the ratio of adaptation operations within an adaptation transaction with respect to the number of nodes involved is 1 : 1 as well, i.e., each node executes exactly one exchange operation locally. The third and fourth cluster are coined by a *TS/SS* ratio of 2 : 1 and 5 : 1, respectively, and contain both equally many `Add`, `Remove` and `Migrate` adaptation operations executed on the respective number of nodes. Evidently, in both clusters, nodes execute multiple adaptation operations during the execution of an adaptation transaction and possibly also within an adaptation group. Experiments WLA and P3 belong to the third cluster, experiments WLB and P4 belong consequently to the fourth cluster.

So far, experiment 37 has not yet been discussed. In this experiment, the behavior of the protocol in case of a local adaptation failure was emulated. Therefore, the experiment was conducted two times: the first iteration was conducted with an automatically generated adaptation transaction in the same way as the previously described experiments whereas in the second iteration, the generated adaptation transaction of the first iteration was tampered in a way to inevitably have the execution of the adaptation transaction fail. Since the instance identifiers for roles, players and compartments change for each execution of an adaptation transaction, one migrate operation's target role type was modified in a way that no instance of this type could be instantiated locally because the role type did not exist locally, thus, resulting in a local adaptation failure that was supposed to immediately have the execution of the adaptation transaction fail.

5.3.4. Results

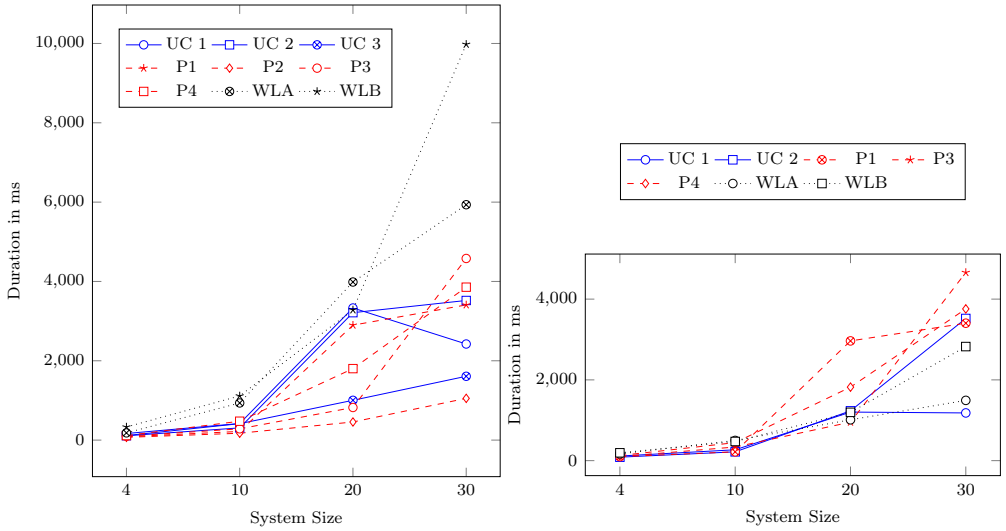
The previously described experiments cover typical application scenarios of the introductory use cases and aim to support a general evaluation of the coordination protocol's performance and scalability. Since the experiment clusters contain different compositions of adaptation transactions and are executed on different system sizes with differing message loss rates, certain expectations can be made with respect to the execution duration and success rate of the adaptation transaction. These expectations will be discussed first before the actual results will be presented and critically discussed subsequently. Finally, the results of the emulation will be briefly summarized.

Expected Results

A very first intuitive expectation is the adaptation duration to consume more time for emulations configured with a higher message loss probability on the communication chan-

nel between nodes. This expectation is evident because adaptation managers will not be able to obtain all necessary Report messages for an increasing number of lost coordination messages, which inevitably leads to timeouts that stall the execution of the adaptation transaction. Closely related to the increased adaptation duration for higher message loss rates on the communication channels is the success rate with which adaptation transactions can be finished. The success rate is expected to decrease with an increasing probability of coordination control messages to be lost on transmission during an adaptation process. For both the adaptation duration and the success rate, we expect adaptation transactions containing `Migrate` operations to consume more time to finish and to fail more often with an increasing message loss rate compared to experiments not containing any `Migrate` operations. The expectation is based on the protocol specification for the execution of distributed adaptation operations displayed in Figure 4.8 on Page 77 if the transmission of state information is required. The adaptation operation can only be finished after the target node received the state information from the source node. If this one StateTransfer message is lost, however, the entire system has to wait for the specified timeout period because the respective adaptation group will not be able to finish due to this one distributed adaptation operation. This is expected to be reflected in the adaptation duration for the emulation of the coordination protocol. Furthermore, the target node is only able to obtain the missing state information from the source node since this information is not broadcast to any other node. Consequently, three lost messages between source and target node will already be enough to fail the entire adaptation transaction, which evidently becomes more and more likely to happen with an increasingly unreliable communication channel between the respective nodes.

Another expectation is concerned with the iterations of the experiments that are free of message losses, i.e., the message loss rate is configured with 0%. If the system size grows, the number of adaptation operations within an adaptation transaction also grows. Consequently, more but at most ten adaptation groups will be randomly created for experiments 1 through 20. Adaptation transactions configured with less adaptation groups are expected to finish faster than those with more adaptation groups because adaptation groups are executed sequentially. Consequently, experiments 21 through 36 are expected to finish fastest within their respective clusters and within the given system size since all adaptation operations are performed within only one adaptation group. Similarly, experiments 13 through 20 are expected to require the most execution time within their respective clusters and within the given system size.



(a) Average adaptation duration.

(b) Average downtime of migrated role instances.

Figure 5.3.: Baselines (0% message loss) for the conducted experiments and different node sizes.

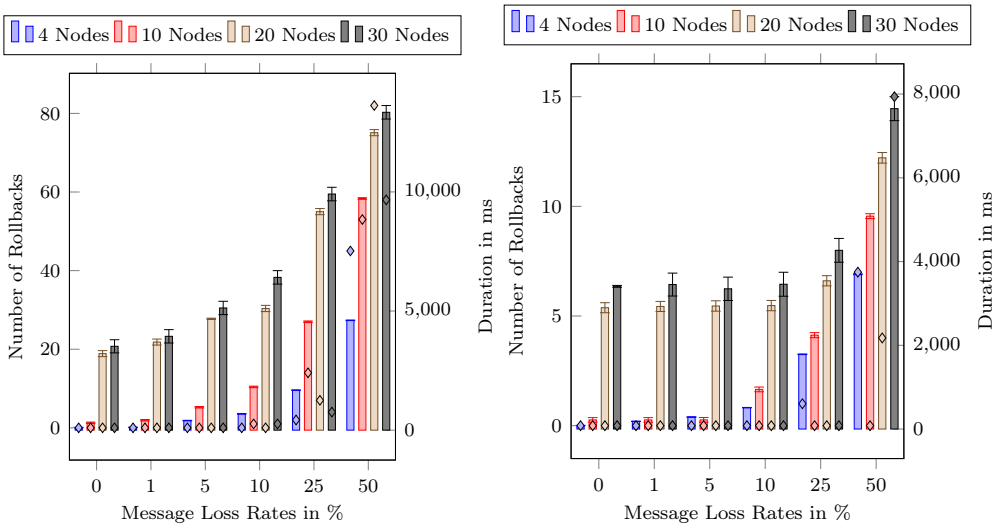
Result Presentation & Discussion

Every experiment was emulated six times. In each iteration, the communication channel between the nodes, i.e., Docker containers, was configured with a different message loss rate. Those repeated iterations of one experiment with different link failure rates belong logically together and will always be discussed in close relation to the iteration configured with 0% message loss. For the sake of simplicity and better understanding, this 0% message loss iteration will be referred to as the *baseline* of the experiment in the following.

In Figure 5.3, the baselines for all experiments with respect to the adaptation duration (cf. Figure 5.3a) and downtime of roles (cf. Figure 5.3b) are displayed. Two observations of the baselines of every conducted experiment are striking. The first result is the realization that all compositions of the adaptation transaction perform very well for the smallest system size, which contained four participating nodes. *WLB* required the most time to perform the specified adaptation transaction ($330ms$), which is an expected result since *WLB* belongs to the cluster with the largest adaptation transactions. Similarly, *P4* required the most execution time of all single-adaptation-group experiments ($114ms$) and shows an execution time comparable to *UC 1* and *UC 3*. The second striking observation is *P2* and *UC 3* performing and scaling extraordinarily well in comparison to the other results. This result is an immediate consequence of the aforementioned limitation of the prototypical implementation not being able to process all received execution control

messages if these arrive within a few milliseconds. As depicted in Figure 5.3, a rather unexpected performance decline can be observed between system sizes of 10 and 20 nodes or 20 and 30 nodes. The logged information of the execution protocol of the adaptation transaction showed unexpected message losses for the baseline scenario, i.e., the receipt of *RequestReport* messages was logged. For system sizes of 4 and 10 nodes, these lost messages occurred infrequently and have no impact on the average results of the adaptation durations or downtimes of roles. In larger system sizes, however, the phenomenon occurred frequently causing the perceivable decline in the execution times of adaptation transactions because the amount of adaptation operations processed in parallel increases (remember: we limit distribution of adaptation operations to 10 adaptation groups) and consequently does the number of exchanged *Report* messages increase appropriately. The implementation of the network communication turned out not to be fast enough to process several almost simultaneously incoming *Report* or *StateTransfer* messages, which is observable as lost coordination messages in the coordination protocol's execution log. Both *P2* and *UC 3* belong to the second cluster, which is coined by adaptation transactions containing only local adaptation operations and is hence less prone to experience this limitation. However, specific characteristics of adaptation transactions have become obvious for adaptation transactions that belong to this cluster in particular, which also explains the longer adaptation duration for *UC 1* for a system size of 20 nodes compared to a system comprised of 30 nodes. A detailed assessment of this observation will be delayed until the results of cluster 2 are discussed in general later. All other clusters contained distributed adaptation operations, too, which easily affects the adaptation duration and downtimes of roles adversely if the *StateTransfer* message is lost during the adaptation process. In summary, a limitation of the prototypical implementation caused an emulation result that was expected for message losses only to show already at the baseline of the experiments.

The limitation of the prototypical implementation is reflected in the downtime of roles depicted in average for all conducted experiments in Figure 5.3b as well. Due to adaptation transactions being atomically executed, roles passivated for a migration, exchange or cloning (not conducted throughout the evaluation) are passive until all adaptation managers agreed on the activation of the performed changes. Consequently, roles that were migrated in early adaptation groups experience a longer inactivity than such roles being migrated in later adaptation groups. The average downtime of roles across all adaptation groups is depicted in Figure 5.3b. Since all adaptation operations in *P1* through *P4* exluding *P2*, which did not execute any distributed adaptation operations such as a migration, are performed within a single adaptation group, the experienced downtimes by



(a) UC 2 (b) P1

Figure 5.4.: Cluster 1: Execution times for the experiments UC 2 and P1 for every system size with different message loss rates.

the respective roles are comparable to the actual execution times of the overall adaptation. The average downtime of roles is also significantly lower for *WLA* and *WLB* for the conducted experiments especially for the largest system size of 30 nodes, which indicates a majority of the conducted `Migrate` operations to be performed in later adaptation groups. Consequently, the results might slightly differ for other adaptation transactions that have a different distribution of the `Migrate` operations across all adaptation groups.

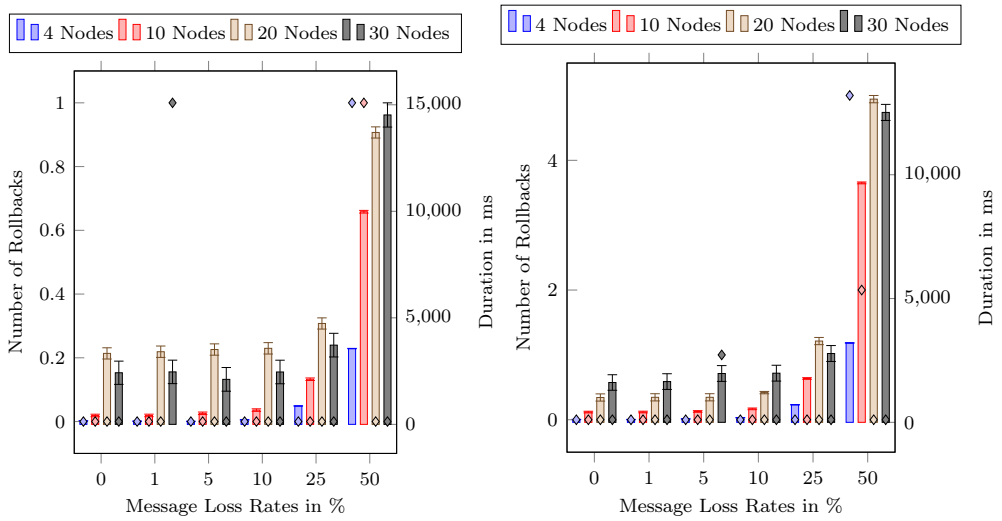
In the remainder, the results of the previously discussed clusters will be discussed focusing on the message loss and its impact on the execution of adaptation transactions for the experiments conducted within the given cluster. Furthermore, expected and unexpected results of the conducted experiments will be part of the discussion of the presented results of the emulation. For all Figures presented and discussed in the following, the diamond character (⋄) in each figure denotes the number of unsuccessfully executed adaptation transactions (rollback case) due to message loss.

Discussion of Cluster 1 As the results for the first cluster show, the coordination protocol is able to maintain a high success rate for the execution of adaptation transactions. Experiments in this cluster are configured with `Migrate` adaptation operations only, thus, the success rate for the execution of adaptation transactions is expected to be lower than for other clusters that also contain local adaptation operations. Without having discussed other clusters yet, this expectation is partially met. Experiment *UC 2* is able to

maintain a success rate of approximately 85% to 98% for a message loss rate of up to 25%, but this success rate drops to approximately 20% to 50% for iterations run with 50% message loss rate as depicted in Figure 5.4a. By contrast, experiment *P1* is able to maintain a steady success rate between 90% and 100%. Since in *P1* all operations are executed within a single adaptation group and messages are dropped randomly with a 50% probability within the emulation, this difference is caused by the higher probability of a single *StateTransfer* message to be dropped for *UC 2* because less messages are transferred within adaptation groups. Furthermore, in *UC 2*, the adaptation transaction can also fail during the transition from one adaptation group to another if Report messages are lost.

The impact of message losses on the average adaptation duration is significant in both experiments. A noticeable increase is only shown for message loss rates higher than or equal to 25%, but increase quickly to up to 44 times more time required than the baseline (4 nodes, *P1*, baseline took 84.4ms, 50% message loss took 3696.27ms). The sudden increase of the adaptation duration recognizable for both *UC 2* and *P1* in Figure 5.4 for the baseline between a system size of 10 and 20 nodes is caused by the previously discussed limitation of the prototypical implementation. It is possible that some of these lost coordination messages contributed to the decline of the success rate of *UC 2*, but a visible impact on the execution of *P1* cannot be proven.

Discussion of Cluster 2 Cluster 2 is composed of experiments configured with local adaptation operations only and the obtained results generally meet the expectations for that cluster. Experiment *P2* performs best for system sizes of 4 and 10 nodes, which is expected. Experiment *UC 3* performs best for system sizes of 20 and 30 nodes, which is unexpected. Both other experiments' adaptation durations are highly influenced by the previously discussed limitation of the prototypical implementation. The additional protocol behavior, i.e., additional Report messages are sent after an adaptation group was finished successfully, to better cope with message losses caused by unprocessed messages due to the limitations of the prototypical implementation, is responsible for the better performance of *UC 3*, especially in comparison to *P2*. Since *P2* only contains one adaptation group, this additional protocol behavior to better cope with the limitation of the implementation does not benefit this experiment. The randomly generated adaptation transactions for *UC 3* and *UC 1* do not differ significantly enough to explain the different impact of the prototypical implementation on the results of the experiments. Apparently, during the iterations of *UC 3* a larger number of crucial Report messages was not processed. A crucial message in this respect is a report that causes a timeout



(a) UC 1

(b) UC 3

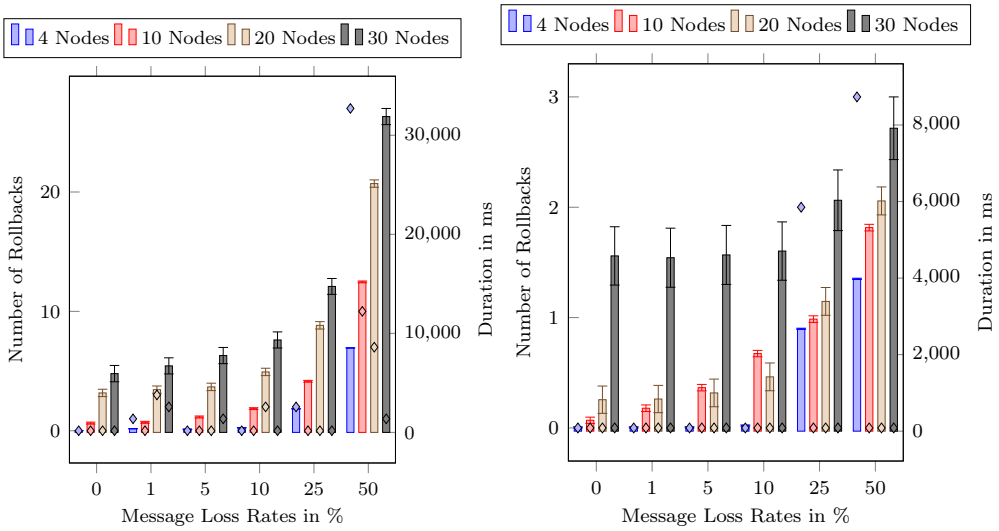
(c) P2

Figure 5.5.: Cluster 2: Execution times for the experiments UC 1, UC 3 and P2 for every system size with different message loss rates.

immediately if it was not received. Both experiments contained adaptation groups that only contain one adaptation operation. In such a situation, every Report message from the previous group is crucial because the isolated adaptation operation will not be executed before at least one timeout period if the progress information was not received. By contrast, in adaptation groups that contain multiple adaptation operations, a Report message can be lost without having an impact on the adaptation duration if both operations are located on different nodes and thus another adaptation manager has the chance to establish knowledge on the successful execution of the previous adaptation group. This other adaptation manager will continue with the execution of its own adaptation operations and the transmitted Report messages to its peers will cause them to continue with the execution of the adaptation transaction without waiting a complete timeout period. In line with this rationale, the striking observation of *UC 1* requiring more time to execute for a system size of 30 than 20 nodes can be explained.

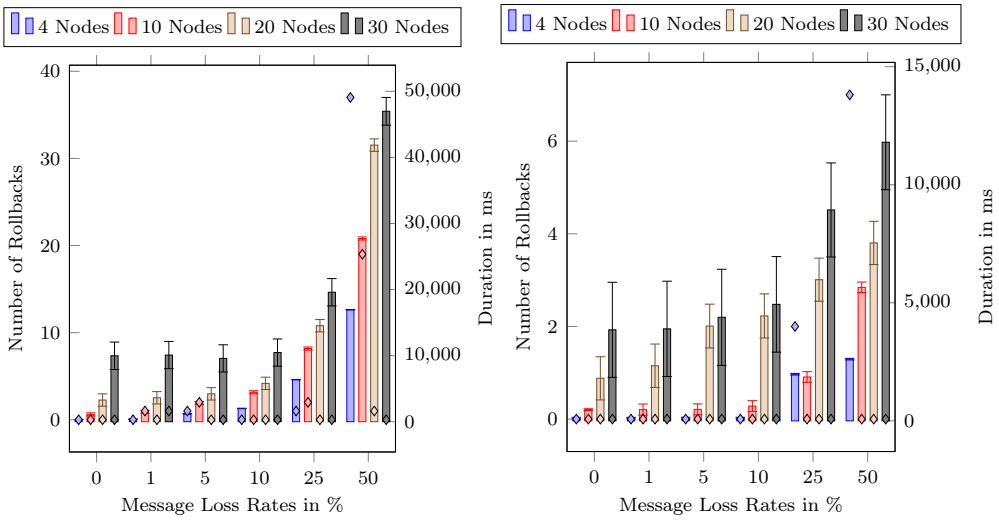
The overall results for the adaptation durations of the experiments in cluster 2 are depicted in Figure 5.5. All experiments are able to maintain a high success rate throughout all iterations of the conducted experiments for the respective system size. This cluster was expected to perform better in this regard than cluster 1 because no state information of roles was required to be transferred and crucial progress information was therefore always obtainable after two timeouts from other adaptation managers. The average adaptation duration, though, also increases by a great margin compared to the baseline iterations and exceeds even 10 seconds for the most unreliable communication channel configuration. Experiment *P2* performs best in this scenario compared to the other two experiments which is evident due to the execution of all adaptation operations in a single adaptation group and the absence of state transfers. After at most 7.5 seconds and additional execution time, an adaptation transaction is expected to fail or succeed for experiment *P2*.

Joint Discussion of Clusters 3 & 4 The results of clusters 3 and 4, which contained both local and distributed adaptation operations within the executed adaptation transaction, will be discussed together because the results in general are comparable except for different adaptation durations for both clusters which is a consequence of the different transaction sizes of both clusters. The result for cluster 3, configured with twice as many adaptation operations than nodes participating in the execution of the adaptation transaction, is depicted in Figure 5.6 for the contained experiments *WLA* and *P3*. Likewise, the results for the experiments *WLB* and *P4* are displayed in Figure 5.7. With respect to the tolerance for message losses during the execution of adaptation transactions, all



(a) WLA (b) P3

Figure 5.6.: Cluster 3: Execution times for the experiments WLA and P3 for every system size with different message loss rates.



(a) WLB (b) P4

Figure 5.7.: Cluster 4: Execution times for the experiments WLB and P4 for every system size with different message loss rates.

experiments within both clusters behave similarly. Out of both clusters, the experiments containing only a single adaptation group, i.e., *P3* and *P4* are able to maintain a success rate of 93% to 100% for all iterations within the respective system size. In contrast, *WLA* and *WLB* perform equally well, but only for system sizes comprising 20 and 30 nodes and worst for a system size of 4 nodes. The reason for this observation can be seen in the contained `Migrate` operations within the adaptation transactions with the same explanation previously given for the results obtained for cluster 1 and the difference of cluster 1's success rate compared to cluster 2. Due to the high number of adaptation operations that are contained within the respective adaptation transactions, the duration of the execution times increases for the execution of the adaptation transaction. Experiment *WLB*, for instance, requires almost 41 seconds on average for the execution of adaptation transactions comprising 30 nodes for 50% message loss on the communication channel, which is an increase of 12 times the average adaptation duration compared to the baseline of the experiment. A striking observation is *WLB* to finish the adaptation transaction faster than *WLA* for a system size of 20 nodes in average, which is a result that is better depicted in Figure 5.3a. The standard deviation for *WLB*'s baseline for this system size amounts to almost 1s across all nodes whereas the standard deviation of *WLA* amounts to approximately 370ms. Both iterations were affected by the limitations of the prototypical implementation and the range of repetitions of the baseline iterations for *WLB* copes slightly better with message losses since more adaptations are conducted locally which means an adaptation manager is likely to require less remote progress information to continue with the execution of the next adaptation group. Consequently, *WLB* performs a little better in average, but considering the standard deviation, both experiments perform equally well for this system size considering the different number of adaptation operations that are executed within the adaptation transaction.

A general remark on the Roles' Downtimes The previous discussion was focused on the adaptation duration and the success rate of adaptation transactions during the emulation of the coordination protocol in the virtualized environment. The downtime of the roles increases in a comparable manner as the adaptation duration increases for higher message loss rates. The increase of the downtime, however, is not distributed evenly across all adaptation groups within the transaction. Due to the protocol's requirement to enable a consistent transition of the system from the source to the target configuration, a role remains in a *Bound/Passive* state as long as the adaptation transaction is going on and enters this state if a collaborating role or the role itself is part of an adaptation operation. Roles are therefore passivated gradually. Consequently, roles that are part

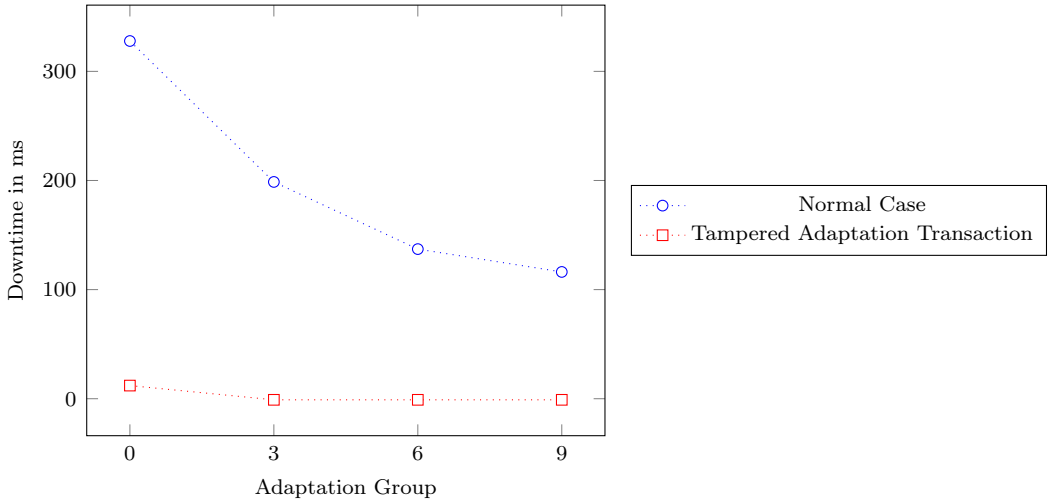


Figure 5.8.: Downtime of roles in adaptation groups for experiment 37 – Adaptation Failure.

of an adaptation operation in an early state, i.e., if the respective adaptation operation is part of an adaptation group configured with a low value for the `Order` parameter, will suffer more from a prolongation of the adaptation duration caused by message loss than roles that are subject to adaptations in later adaptation groups. Average values are therefore biased and coined by a high standard deviation.

Experiment 37 – Adaptation Failure The generated adaptation transaction for experiment 37, which was intended to emulate the behavior of the coordination protocol in the case of a local adaptation failure, contained four adaptation groups executed consecutively according to the protocol specification. In the first adaptation group (group 0) the first `Migrate` operation was executed. This adaptation operation was tampered in order to have the adaptation transaction fail as described previously. The results of the performed experiment for both executions, i.e., normal and tampered, are displayed in Figure 5.8. Apparently, the execution for the normal case yielded results that were expected. The downtime for adaptation groups in the normal case is decreasing for adaptation groups with a higher `Order` parameter since roles in adaptation groups executed early remain in a passive state longer than later executed adaptation groups resulting in the downtimes depicted in Figure 5.8. Consequently, the displayed duration cannot be misinterpreted as execution time of the respective adaptation group.

The execution of the same adaptation transaction with a tampered adaptation operation resulted in a rollback rate of 100%, which means no adaptation transaction was successfully executed – as expected. Since the `Migrate` operation executed in the first

adaptation group, i.e., the adaptation group with the `Order` parameter *0*, was tampered, the downtime for subsequent adaptation groups was measured with -1, which is a value set by the previously mentioned evaluation scripts written to process the obtained measurements results indicating that the adaptation group has not even began the execution of contained adaptation operations because an *TransactionRollback* message was issued and received beforehand. The downtime of the first adaptation group is also considerably shorter because the execution fails immediately for the migration because the type of the role to be instantiated does not exist on the target node.

This experiment was also repeated 100 times for the normal as well as the failure case and the data displayed in Figure 5.8 shows the average values for the measured downtimes.

5.3.5. Summary

The coordination protocol's performance was assessed with respect to the execution time of the adaptation process, the resulting unavailability of system behavior for different degrees of losses of coordination messages, and the influence of such message losses on the previous two parameters in a completely virtualized environment. In the emulation setup, the coordination protocol was implemented and executed by the *adaptation managers* distributed to each node co-located with a local instance of the LyRT runtime representing the *managed application*. Each node was represented by a Docker container configured with a pair of adaptation manager and managed application.

The experiments were clustered with respect to the composition scheme of the executed adaptation transactions, i.e., the number of contained adaptation operations, their type and distribution across adaptation groups. In general, the conducted experiments met the previously discussed expectations for the adaptation duration, success rate and downtime of roles. Although the prototypical implementation caused a perceivable message loss already for experiment baseline iterations, the success rate could be maintained as high as 90% to 100% for almost all experiments and message loss rates of 50%. Only adaptation transactions configured solely with distributed adaptation operations, i.e., experiments *UC 2* and *P1* of cluster 1, suffered from greater failure rates due to the necessary transfer of the roles' internal state information from the source to the target node. This cluster was expected to perform worse than other clusters in this regard, though, and *P1* also reached almost 100% success rates for all message loss rates, which is a favorable result that was not expected in this degree.

Even with the outlined limitation of the prototypical implementation, the average adaptation duration of all experiments is below 1.5 seconds for the baseline iterations

for system sizes of 4 and 10 nodes. For a system size of 4 nodes, this result is even maintained for up to 5% message loss. If 10 nodes participate in the adaptation process, only *WLB* requires more than 1.5 seconds for the execution of adaptation durations in average. The results for larger system sizes including system sizes of 20 to 30 nodes for some experiments indicate a performance along this dimension which gives a hint on the protocol's performance with an improved implementation. Especially *UC 3* provides a glimpse on the best performance of the protocols execution for adaptation transaction containing only locally executable adaptation operations.

In general, the coordination protocol is able to maintain a high success rate for the execution of adaptation transactions even in large system sizes coined by a high degree of message loss on the communication channel used for the coordination process. Experiments of clusters 3 and 4 also showed the protocol to be able to handle a high number of adaptation operations within a limited number of nodes. Up to 5 times the adaptation operations than nodes in the system perform well with respect to the success rate, but require significantly more time to execute, which gets worse with an increasing message loss rate. Message loss rates up to 10% showed to have generally little impact on the average of the adaptation duration.

5.4. A Formal Validation of the Coordination Protocol

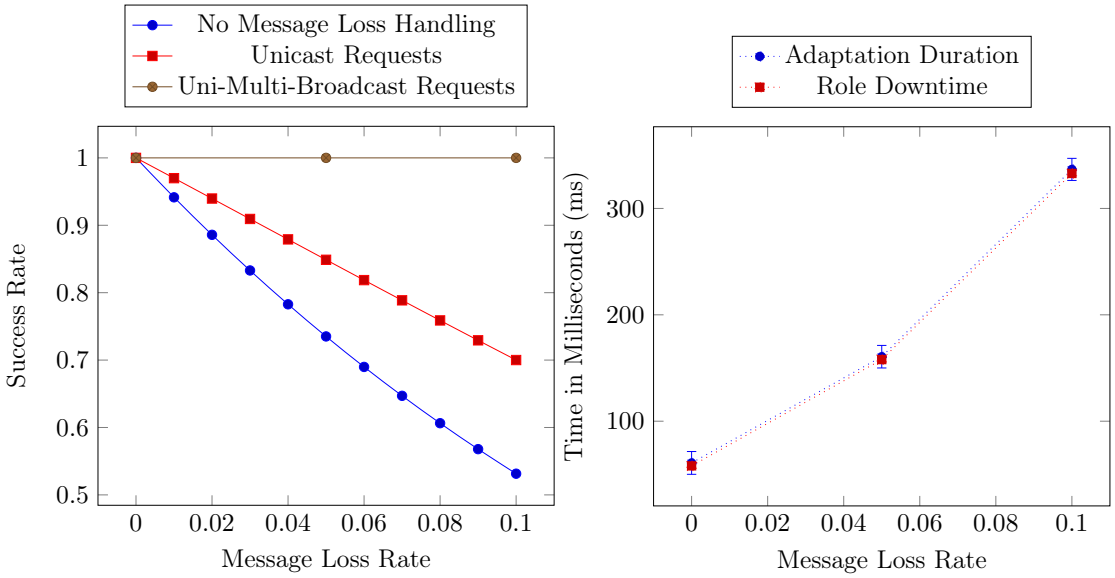
We created a formal model (cf. Appendix A), which was originally presented in [50], of the adaptation protocol and established it to be deadlock-free by means of *model checking* [2]. Model checking is a formal verification technique that checks whether a given model of the system under consideration satisfies a formal specification. A *model checker* systematically explores all possible states of the system to verify whether the system satisfies the formal specification. We modeled the protocol in the ProFeat[2] [8] modeling language. ProFeat extends the input language of the probabilistic model checker PRISM[3] [20] by feature-based concepts, and follows a translational approach, i.e., ProFeat models are translated into standard PRISM models. Subsequently, the analysis of the model is carried out using PRISM.

In the following, we give a short overview of the characteristics of the developed model. The model represents one or more nodes with their respective adaptation managers. The model addresses the behavior of the adaptation managers as it was described in Chapter 4.4. Since the execution of an adaptation group and a single adaptation operation

[2] `https://wwwtcs.inf.tu-dresden.de/ALGI/PUB/ProFeat/`
[3] `http://www.prismmodelchecker.org/`

are the significant parts of the protocol to perform adaptation processes, the developed formal model is focused on these two parts. This results in the formal model being composed of a single adaptation group and three adaptation operations covering local and distributed adaptation operations. With respect to the execution of an adaptation group, only the `Perform Operation`, `Waiting` and `Requested Peer` states were modeled. The `Rollback` state and any subsequent state reached by occurred timeouts resulted in the model reaching the final state and indicated an unsuccessful execution of the adaptation transaction. This limitations of the model are valid, as the additional escalation steps to obtain lost coordination messages do not introduce new behavior to the already modeled features of the coordination protocol and can thus be omitted with the aim to keep the model small. The managed application and the behavior of the roles are not modeled since the adaptation mechanisms are strictly separated from the application. The nodes are running concurrently and may exchange protocol messages asynchronously. The network is modeled as a finitely sized buffer that stores messages until they are received by their respective nodes. Messages may get lost and can be reordered. The model implements the `Add`, `Remove` and `Migrate` adaptation operations, which are executed in accordance to the adaptation transaction describing the roles affected by the change and the adaptation managers responsible for the adaptation transaction's execution. The model is parametrized over the number of nodes, the network buffer size and the probability for message loss. Furthermore, the protocol extension for handling message loss may be deactivated. Thus, the model can be easily adapted to check and analyze different scenarios. A complete representation of the model written in ProFeat can be found in Appendix A.

The model instance was analyzed with 3 nodes, 2 roles and an adaptation transaction consisting of one role transfer and one local operation, e.g., the addition or removal of a role. In order to keep the model small, it only describes a single adaptation transaction with exactly one adaptation group. However, since adaptation groups are executed sequentially the analysis results also apply to adaptation transactions with more than one adaptation group. We could establish the knowledge that the protocol never runs into a deadlock. Furthermore, the analysis showed that the adaptation is always successful in case of no message losses. In addition to the success rate of in the ideal case without message loss, the protocol was executed in the model checker with a set message loss rate decreasing by 1% until a message loss rate of 10% was reached. The resulting success rate is based on the simplified error handling incorporating only the first escalation step of one unicast message in response to missed Report or StateTransfer messages. The used adaptation transaction was furthermore executed with the prototypical imple-



(a) Success rate of the coordination protocol for the validated model and of the prototypical implementation.

(b) Adaptation duration and downtime of the migrated role for the performed validation.

Figure 5.9.: Results of the performed adaptation transaction for the simulation in the model checker and the emulated execution of the coordination protocol.

mentation of the coordination protocol for 0%, 5% and 10% message loss (denoted as *Uni-Multi-Broadcast Requests* inf Figure 5.9a). In Figure 5.9a, the average success rate of the experiment over 100 runs is depicted in comparison with the statistical analysis of the model checking for the just described protocol specifications. For the emulation of the adaptation transaction, the measures for the role downtime and the overall duration of the adaptation have been captured and are displayed in Figure 5.9b. The small downtime in this small application scenario is an expected result compared to the average adaptation duration and downtime of the previously conducted experiments coined by a system size of only four nodes.

5.5. Summary

The coordination protocol to perform distributed adaptations in a self-adaptive software system at run time without a central coordinator was evaluated in this chapter using a formal model checking approach and by executing a prototypical implementation of the protocol in a virtualized environment in order to obtain results how well the developed protocol is able to coordinate the execution of adaptations, especially with respect to the

Table 5.3.: Overview of the posed requirements and the chosen approach to evaluate the respective requirement.

Requirement	**Evaluation Approach**
Decentralized Execution	Validation, Emulation
Stable Application State	partially evaluated
Link Failures	Validation, Emulation
Adaptation Failures	Emulation

assumed failure models, i.e., link and adaptation failures. The evaluation of the approach also partially covers the requirements posed to the approach developed within this thesis. Table 5.3 gives an overview of the requirements and the chosen approach to evaluate the respective requirement. In the following, the results of this chapter will be summarized considering the posed requirements and the evaluation's results.

Using formal model checking, the coordination protocol could be validated to be free of deadlocks, i.e., the execution of an arbitrary adaptation transaction always reaches one of the defined final states, which means, the protocol ensures the managed application to reach the desired target configuration or remains in the source configuration in case of link or stopping failures. Furthermore, the results of this formal model checking hold for arbitrary application domains in which the proposed coordination protocol is employed. Ensuring the deadlock freeness of the coordination protocol in arbitrary application domains is an important result as the consistent transition of the managed application from its source to its target configuration, which is a major goal of a self-adaptive software system and which cannot be achieved without a deadlock-free coordination protocol. The formal validation of the coordination protocol includes the response of each adaptation manager to lost coordination messages and thus, addresses both the *Decentralized Execution* and the *Link Failure* requirement. The coordination protocol's execution was emulated to evaluate the *Decentralized Execution*, *Link Failure* and *Adaptation Failure* requirements. The emulation shows for the first two mentioned requirements that the protocol is able to adapt the managed application without a central coordination entity and that the presented approach is able to cope with lost coordination messages. Executing the protocol in environments coined by unreliable communication channels configured with different message loss rates, an assessment is made possible how well the protocol performs in ideal and unstable environments. The last requirement was evaluated using tampered adaptation transactions as input for the adaptation process. It could be practically shown that the adaptation managers indeed immediately abort the adaptation transaction as soon as the execution of one adaptation operation fails locally.

Since collaborating roles have not been part of the evaluation, the *Stable Application State* requirement was only partially evaluated. During the execution of the adaptation transactions, local roles were passivated in accordance to the protocol specification presented in Chapter 4 and in accordance to the proposed role life cycle (cf. Chapter 3.2.1), thus ensuring the single role instances to be in a stable application state before the respective adaptation operations affecting those role instances were executed. As collaborations in the emulation setup are missing, a full evaluation of this requirement with respect to ongoing transactions as described by Kramer & Magee [26, 25] was not performed and is left for future work.

Especially in response to research question two, which was concerned with the decentralized execution of multiple interdependent adaptation operations, the coordination protocol was developed within this thesis. In the previous chapter, we argued the coordination protocol to be able to cope with message losses and node failures as well as adaptation failures during the execution of an adaptation transaction. The emulation of the coordination protocol also showed practically that the loss of coordination messages during the execution of an adaptation transaction is handled well with the help of the proposed approach. A high success rate could be maintained for the majority of experiments conducted in the emulation and for different system sizes in combination with message loss rates up to and including 50%. Especially for high message loss rates such as 50%, the adaptation duration increases significantly, which immediately affects the downtime of roles being migrated, exchanged or cloned as well. For message loss rates of 10% in particular only little overhead is caused for the overall adaptation duration in average for the execution of adaptation durations, which is a favorable result.

Maintaining a consistent adaptation in a timely fashion, though, becomes increasingly difficult for larger system sizes and increasing message loss rates. Applying further means of structuring the self-adaptive software system on levels above the execution phase to minimize the scope of adaptation operations and affected nodes that need to perform an adaptation transaction in response to context changes is a valuable insight that can be gained from the conducted emulation of our approach. Patterns to organize the feedback loop within the managed application in order to structure adaptations hierarchically or to introduce regions of the system that have to be adapted consistently [51] could be a possible application within the feedback loop to fully utilize the benefits of a consistent adaptation execution as it was presented in this thesis.

Within the emulation, the timeout period between RequestReport messages used to obtain lost progress information was kept static. Since increasing node sizes and increasing numbers of adaptation operations have shown an increase in the local execution time

of adaptation operations, a dynamically configurable timeout or a short first timeout period that becomes longer for the multicast and broadcast requests as it was outlined in Chapter 4.4.3, appears to be a valid approach to improve the performance of the protocol. Especially in real-world environments that also introduce a delay on the communication channel, which was neglectable in the emulation due to the full virtualization of the setup, such a dynamic or hierarchically structured timeout handling appears feasible and favorable.

Similarly to the scope of the formal validation of the coordination protocol, the emulated results hold for arbitrary application domains as no specific application setup was used. Nonetheless, as described above, results may differ in real-world settings as network latency is introduced, resulting in possibly longer adaptation durations and downtimes of roles especially if the devices are scattered across a large spatial area. Following this thought, executions of the adaptation protocol in larger setups may perform better than projected in the emulation as the previously outlined drawbacks of the prototypical implementation may be mitigated by the introduced network latency. These considerations, however, need to be further investigated in future work.

6. Conclusion

In this thesis, a coordination protocol was presented that allows for the consistent adaptation of distributed self-adaptive software systems without a central control or management unit. The protocol is able to maintain a consistent application state throughout the adaptation process even in unstable environments coined by unreliable communication channels used to exchange control and management information or if adaptations locally fail during the execution of the adaptation. The approach followed the idea of external control [38] that strictly separates the adaptation management and the managed application of a self-adaptive software system into two distinct subsystems. Since this thesis focuses on run-time aspects during the execution of adaptations, the adaptation management in this thesis – represented by the *Adaptation Manager*s distributed across all devices part of the system – is only concerned with the execution phase of the MAPE feedback loop [23]. Maliciously behaving adaptation managers are not considered in this thesis and respective measures remain for future work. In the remainder of this chapter, the presented approaches and solutions of this thesis are briefly summarized and placed into relation with the requirements and research questions introduced in the beginning. Subsequently, future research directions related to the presented work will be discussed.

6.1. Summary of Requirements and Research Questions

The requirements posed in this thesis addressed the two main goals that are supposed to be achieved by the execution phase of a self-adaptive software system, which are to prevent the managed application from losing application data during the adaptation process and to prevent the managed application from reaching an inconsistent configuration during the adaptation process. Additionally, the envisioned solution was supposed to be able to perform adaptations on the managed application without a central coordinator that has complete knowledge about the adaptation process and which is considered a possible single point of failure that coincides with the goal to prevent an inconsistent system configuration.

Both the local interface to the role-based runtime system, presented in Chapter 3,

and the coordination protocol, presented in Chapter 4, contain concrete mechanisms addressing *Research Question 1*, which is concerned with reaching a stable application state before the adaptation process can actually be commenced. The coordination protocol provides a set of dedicated messages in order to reach a stable application state for collaborating roles as well as context-dependent roles. Both furthermore rely on the presented life cycle for roles in order to determine such a stable state. The presented work, however, is preliminary and further interesting and challenging research directions remain to be tackled, which we will elaborate on in the following section.

The developed coordination protocol allows the decentralized adaptation of the managed application and ensures the previously managed system goals to be achieved during an adaptation process. Another major feature of the coordination protocol is the ability to perform multiple correlated adaptations that require the modification of parts of the managed application located on distributed devices, which we also referred to as nodes. In Chapter 4, a coordination protocol was developed in response to *Research Question 2* that is the core of the decentralized adaptation process presented within this thesis. The coordination was designed to cope with message losses as well as with node and adaptation failures that occur during the execution of an adaptation process. In Chapter 5.4, the coordination protocol was formally validated with respect to being free of any deadlocks for different sizes of the system under adaptation, especially in the case of exchanged coordination messages getting lost during transmission.

In order to differentiate parts of the application that do not have to be adapted in response to changes in the operational environment of the system from those that are dynamic, we used the *Role* concept to separate static system behavior from context-dependent system behavior, which can be added, removed, exchanged etc. in response to changes in the self-adaptive software system's context. This notion is used as an abstraction layer within this thesis in order to enable a platform-independent protocol to coordinate the decentralized execution of adaptations. Using the *Role* concept therefore addresses *Research Question 3*. In Chapter 3, we made a first contribution to behavioral, context-dependent and collaborative roles in distributed, self-adaptive software systems establishing not only a life cycle for roles, which supports a decentralized adaptation process, but also describing an interface to a local runtime that can be used to abstract from platform-specific implementations of managed applications, e.g., component-based or service-oriented systems, in order to reuse the proposed coordination protocol.

In conclusion, all posed requirements could be addressed within the concepts presented in this thesis and all research questions were answered sufficiently. In Chapter 5, the general feasibility and applicability of the proposed concepts was shown. Additionally, a

qualitative assessment of the coordination protocol in real-world environments through an emulation was conducted giving an insight to researchers focusing on other concerns of the adaptation process, i.e., the planning of changes or what might be feasible amounts of adaptation operations to perform a consistent update of the system in contrast to when adaptation should be structured into smaller, better manageable units.

6.2. Future Work

The specification of the protocol presented in this thesis focuses on the execution of adaptation transactions in self-adaptive software systems. An adaptation transaction describes role-based modifications that are supposed to be enacted upon the managed application in response to changes in the computational environment, i.e., context, of the system. In this regard, two major research objectives can be identified that have not been covered in this thesis, but are an important part of the execution of adaptations and resolve around the stable application state that is required to be reached before adaptations take place. Within this thesis, such a state was assumed to be somehow reachable as a prerequisite for the adaptation process, which was closely investigated.

The seminal works of Kramer & Magee for a quiescent [26] or of Vandewoude et al. for a tranquil [45] application state were assumed as formal foundations for a stable application state. The discussed software reconfiguration patterns [15] and adaptation patterns [16] are a starting point in order to determine a quiescent application state without the need for a central control unit. Using the *Role* concept as abstraction to design and implement dynamic, context-dependent entities, a first step in the direction of achieving a stable application state for role-based software systems has been taken with the design of a role-based life cycle model to support quiescence and tranquility. The life cycle, however, applies to role instances local to a specific device. It remains an open question how the life cycles of different collaborating roles can be synchronized across device borders prior to the execution of an adaptation transaction in order to fully ensure a consistent transition of the application from the source to the target configuration without message loss. Reusing the proposed reconfiguration and adaptation patterns for role-based applications is an obvious starting point for future research, but requires a more detailed investigation. Especially with respect to the scope of collaborations several approaches to reach a stable application state involving multiple roles come to mind. A first intuitive solution would result in reusing the notion of a *transaction* introduced by Kramer & Magee [26] to safeguard ongoing communications within a collaboration entirely. LyRT [43] already implements such an approach on the level of a local role

runtime. An extension of this idea incorporating existing works previously mentioned constitutes this first solution. A more fine-grained second solution would only safeguard ongoing invocations of remote roles instead of entire collaborations, which will likely result in faster and more timely adaptations at the cost of a possibly higher coordination overhead. Both approaches and their mutual trade-offs are an interesting direction for future research on the decentralized execution of role-based adaptation transactions.

Similarly, the activation of the new system behavior after the adaptation transaction was executed remains an interesting research objective. The current protocol specification moves roles in a passive, i.e., stable state, as soon as they or any collaborating roles are subject to adaptations and reactivates them after all adaptations have been performed. Evidently, roles located in early executed adaptation groups remain passive for a much longer amount of time than those in later executed adaptation groups, which is an observation that is also supported by the obtained emulation results. A first research objective results in a relaxed consistency constraint for the execution of adaptation transactions, which means, performed adaptation operations can be activated gradually during the adaptation process in order to reduce the unavailability of system behavior affected by the adaptation process. As a consequence, more sophisticated compensation and rollback mechanisms would have to be developed in order to support reverting already performed and activated adaptations in the case of adaptation failures in later adaptation groups. Similarly, additional specifications may be required from the planning phase of the feedback loop or at design time of the application to indicate parts that should be kept consistent. An adaptation group, for example, is unsuitable to be considered such a consistent unit by default, if collaborations are to be released and reestablished after a role migration, which would result in three adaptation groups executed sequentially but imposing a scope of consistency during the adaptation process. A second research objective is the coordinated agreement of the activation itself. The current protocol specification detects adaptation and node failures preventing a consistent activation of the performed adaptations, but only until the point in time all adaptation managers agreed upon activating the target configuration. Node failures that happen after that agreement and during the activation or adaptation failures preventing a successful local switch from the source to the target configuration after this agreement and in the moment the activation takes place is a corner case that is currently not supported by the protocol specification. Having the execution of local activations agreed upon using Report messages is a simple but naive way of solving this issue, which would result in three phases required to perform the adaptation process (cf. Chapter 4.3).

A. Formal Model

In the following, the developed formal model of the approach is depicted as described in Chapter 5.4 on Page 117.

```
// Adaptation Protocol Model (asynchronous) {{
// vim: foldmethod=marker foldmarker={{,}} :

mdp

// }}
// Model Parameters {{

family {
    MSG_LOSS_PERCENT : [0..10];
}

const bool WITH_RETRY = true;
const BUFFER_SIZE = 2;
// const double P_MSG_LOSS = 0.01;
formula P_MSG_LOSS = MSG_LOSS_PERCENT / 100;

// }}
// Constants {{

const NO_ROLE = -1;

// }}
// Feature Model {{

root feature {
    all of Node[NUM_NODES];

    modules Network(BUFFER_SIZE), Scenario;
}

```

```
feature Node {
    modules Node;
}

// }}
// Scenario {{

const NUM_NODES = 3;
const NUM_ROLES = 2;

const NODE_REMOVE = -1;
const NODE_ADD = 2;
const NODE_ADD_ROLE_ID = 1;
const NODE_TRANSFER_FROM = 0;
const NODE_TRANSFER_TO = 1;

const bool NODE_INVOLVED = { false, true, true };
const INITIAL_ROLE_ID = { 0, NO_ROLE, NO_ROLE };

const STEP_INIT     = 0;
const STEP_ADAPT    = 1;
const STEP_REMOVE   = 2;
const STEP_ADD      = 3;
const STEP_TRANSFER = 4;
const STEP_TIMEOUT  = 5;
const STEP_DONE     = 6;

label "goal" = step = STEP_DONE & for node in [0 .. NUM_NODES - 1]
    { running(node) & ... };
formula running(i) = Node[i].state = 0 & !Node[i].
   missing_role_state;

module Scenario {
    // nodes (actively) involved in the adaptation
    involved : array [0 .. NUM_NODES - 1] of bool init false;

    step : [STEP_INIT .. STEP_DONE] init STEP_INIT;

    [] step = STEP_INIT -> (step' = STEP_ADAPT) & for i in [0 ..
       NUM_NODES - 1] { (involved[i]' = NODE_INVOLVED[i]) };
```

```
    [adapt] step = STEP_ADAPT -> (step' = STEP_REMOVE);

    for node in [0 .. NUM_NODES - 1] {
        [remove[node]] NODE_REMOVE = node & step = STEP_REMOVE ->
           (step' = STEP_ADD);

        for role in [0 .. NUM_ROLES - 1] {
            [add[roleto(role, node)]] NODE_ADD = node &
               NODE_ADD_ROLE_ID = role & step = STEP_ADD -> (step'
                = STEP_TRANSFER);
        }

        for target in [0 .. NUM_NODES - 1] {
            [transfer[fromto(node, target)]] NODE_TRANSFER_FROM =
               node & NODE_TRANSFER_TO = target & step =
               STEP_TRANSFER -> (step' = STEP_TIMEOUT);
        }

        [sentreports[node]] true -> (involved[node]' = false);
    }

    [timeout] step = STEP_TIMEOUT & for i in [0 .. NUM_NODES - 1]
       { !involved[i] & ... } -> (step' = STEP_DONE);

    [] NODE_REMOVE = -1 & step = STEP_REMOVE -> (step' = step + 1)
       ;
    [] NODE_ADD = -1 & step = STEP_ADD -> (step' = step + 1);
    [] ( NODE_TRANSFER_FROM = -1 | NODE_TRANSFER_TO = -1) & step =
        STEP_TRANSFER -> (step' = step + 1);
}

// }}
// Module Network {{

const EMPTY = -1;

module Network(buf_size) {
    cell : array [0 .. buf_size - 1] of [EMPTY .. NUM_MSGS - 1]
       init EMPTY;

    // send: enqueue
```

```
        for msg_id in [0 .. NUM_MSGS - 1] {
            // enqueue in first free slot
            for c in [0 .. buf_size - 1] {
                [send[msg_id]] for i in [0 .. c - 1] { cell[i] !=
                    EMPTY & ... } & cell[c] = EMPTY ->
                     (1 - P_MSG_LOSS): (cell[c]' = msg_id) +
                     P_MSG_LOSS: true;
            }
        }

        // recv: dequeue
        for msg_id in [0 .. NUM_MSGS - 1] {
            for c in [0 .. buf_size - 1] {
                [recv[msg_id]] cell[c] = msg_id -> (cell[c]' = EMPTY);
            }
        }
    }

    // }}
    // Module Node {{

    const S_ACTIVE   = 0;
    const S_PASSIVE  = 1; // awaiting commands or report messages
    const S_SEND     = 2; // sending report messages
    const S_TRANSFER = 3; // send state transfer message
    const S_RESEND   = 4; // resending a report message after request
    const S_REQUEST  = 5; // sending a request for all pending reports
    const S_MAX      = 5;

    module Node {
        // Local State {{

        state : [0 .. S_MAX] init S_ACTIVE;
        prev_state : [0 .. S_MAX] init 0;

        // used to iterate over nodes
        index : [0 .. NUM_NODES] init 0;

        // role residing on this node
        role_id : [NO_ROLE .. NUM_ROLES-1] init INITIAL_ROLE_ID[id];
```

```
    // nodes with pending report message
    missing : array [0 .. NUM_NODES - 1] of bool init false;

    // state transfer from other node is pending
146 missing_role_state : bool init false;

    // pending request from this node
    request_from : [0 .. NUM_NODES - 1] init 0;

151 // }}
    // Adaptation Messages {{

    // initiate adaptation
    [adapt] state = S_ACTIVE -> (state' = S_PASSIVE) &
156     for i in [0 .. NUM_NODES - 1] { (missing[i]' = involved[i
            ]) };

    // ADD
    for r in [0 .. NUM_ROLES - 1] {
        [add[roleto(r, id)]] state = S_PASSIVE -> (role_id' = r) &
            (state' = S_SEND) & (missing[id]' = false);
161 }

    // REMOVE
    [remove[id]] state = S_PASSIVE -> (role_id' = NO_ROLE) & (
        state' = S_SEND) & (missing[id]' = false);

166 // TRANSFER
    for node in [0 .. id - 1] {
        [transfer[fromto(id, node)]] state = S_PASSIVE -> (role_id
            ' = NO_ROLE) & (state' = S_TRANSFER) & (request_from' =
            node);
        [transfer[fromto(node, id)]] state = S_PASSIVE ->
            (role_id' = Node[node].role_id) & (state' = S_SEND) &
                (missing[id]' = false) & (missing[node]' = true) &
                (missing_role_state' = true);
171 }
    for node in [id + 1 .. NUM_NODES - 1] {
        [transfer[fromto(id, node)]] state = S_PASSIVE -> (role_id
            ' = NO_ROLE) & (state' = S_TRANSFER) & (request_from' =
            node);
```

```
        [transfer[fromto(node, id)]] state = S_PASSIVE ->
            (role_id' = Node[node].role_id) & (state' = S_SEND) &
               (missing[id]' = false) & (missing[node]' = true) &
               (missing_role_state' = true);
176 }

    // }}
    // State Transfer {{

181 for node in [0 .. NUM_NODES - 1] {
        [send[transfer(id, node)]] request_from = node & state =
           S_TRANSFER -> (state' = S_RESEND) & (prev_state' =
           S_PASSIVE); // reuse resend state to send report to
           target node
        [recv[transfer(node, id)]] id != node -> (
           missing_role_state' = false);
    }

186 // }}
    // Report Messages {{

    // receive REPORT messages
    for i in [0 .. NUM_NODES - 1] {
191     [recv[report(i, id)]] missing[i] -> (missing[i]' = false);
    }
    [] state = S_PASSIVE & !has_missing -> (state' = S_ACTIVE);

    // send REPORT messages
196 for i in [0 .. id - 1] {
        [send[report(id, i)]] state = S_SEND & index = i -> (index
           ' = index + 1);
    }
    [] state = S_SEND & index = id -> (index' = index + 1);
    for i in [id + 1 .. NUM_NODES - 1] {
201     [send[report(id, i)]] state = S_SEND & index = i -> (index
           ' = index + 1);
    }
    [sentreports[id]] state = S_SEND & index = NUM_NODES -> (index
       ' = 0) & (state' = S_PASSIVE);
    [send[report(id, id)]] false -> true; // block REPORT messages
        with same sender and reveiver
```

```
206     // resend REPORT message
        for i in [0 .. NUM_NODES - 1] {
            [send[report(id, i)]] request_from = i & state = S_RESEND
               -> (state' = prev_state) & (prev_state' = 0) & (
               request_from' = 0);
        }

211     // }}
        // Request Messages {{

        // handle message loss
        [timeout] WITH_RETRY & state = S_PASSIVE & has_missing -> (
           state' = S_REQUEST);
216     [timeout] WITH_RETRY & state = S_ACTIVE | state = S_PASSIVE &
           !has_missing -> true;
        [timeout] !WITH_RETRY -> true;

        // receive REQUEST messages
        for i in [0 .. NUM_NODES - 1] {
221         [recv[request(i, id)]] true -> (prev_state' = state) & (
               state' = S_RESEND) & (request_from' = i);
        }

        // send REQUEST messages
        for i in [0 .. NUM_NODES - 1] {
226         [send[request(id, i)]] state = S_REQUEST & index = i &
               missing[i] -> (index' = index + 1);
            [] state = S_REQUEST & index = i & !missing[i] -> (index'
               = index + 1);
        }
        [] state = S_REQUEST & index = NUM_NODES -> (index' = 0) & (
           state' = S_PASSIVE);

231     // }}
    }

    formula has_missing = for i in [0 .. NUM_NODES - 1] { missing[i] |
        ... };

236 // }}
```

```
// Messages {{

const REPORT  = 0;
const REQUEST = REPORT + NUM_NODES * NUM_NODES;
const TRANSFER = REQUEST + NUM_NODES * NUM_NODES;
const NUM_MSGS = TRANSFER + NUM_NODES * NUM_NODES;

formula report(from, to) = REPORT + from * NUM_NODES + to;
formula request(from, to) = REQUEST + from * NUM_NODES + to;
formula transfer(from, to) = TRANSFER + from * NUM_NODES + to;
formula roleto(role_id, node_id) = node_id * NUM_ROLES + role_id;
formula fromto(source, target) = source * NUM_NODES + target;

// }}
```

B. Protocol Messages

B.1. Transaction Control Messages

In the following, a graphical overview on the message types used to distribute adaptation transactions as well as to distribute information about the successful or unsuccessful execution of adaptation transactions is given. A detailed description of the *transaction control messages* was given in Chapter 4.4.1 on Page 71.

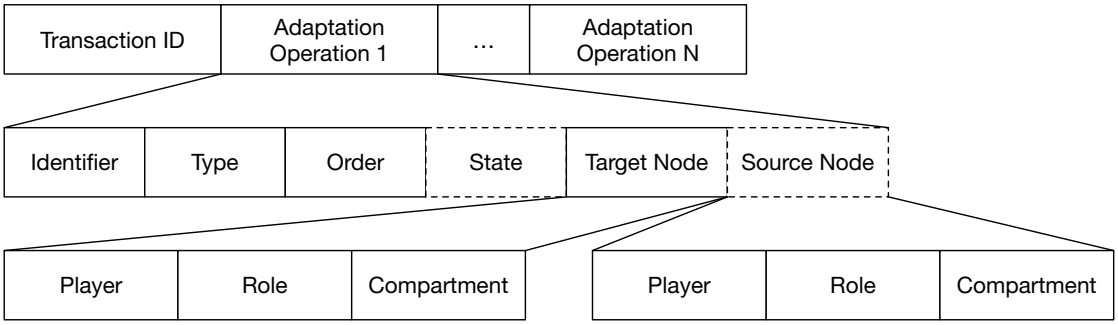


Figure B.1.: Transaction Message

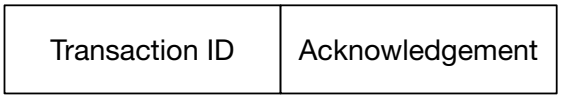


Figure B.2.: TransactionAcknowledgement Message

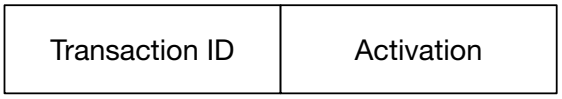


Figure B.3.: TransactionActivation Message

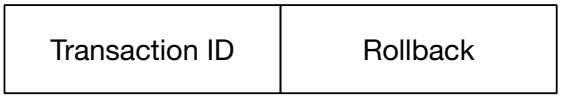


Figure B.4.: TransactionRollback Message

B.2. Execution Control Messages

In the following, a graphical overview on the message types used to coordinate the execution of adaptation transactions, groups and operations is given. A detailed description of the *execution control messages* was given in Chapter 4.4.1 on Page 4.4.1.

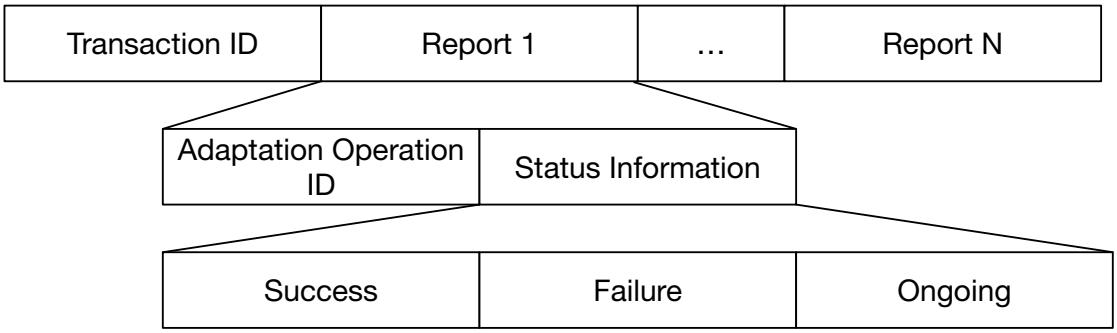


Figure B.5.: Report Message

Figure B.6.: StateTransfer Message

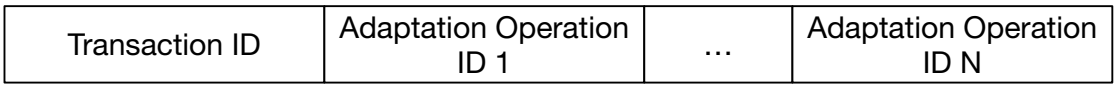


Figure B.7.: RequestReport Message

Bibliography

[1] Charles W Bachman and Manilal Daya. "The role concept in data models." In: *Proceedings of the third international conference on Very large data bases* (Oct. 1977), pp. 464–476.

[2] C Baier and J P Katoen. *Principles of model checking*. The MIT Press, 2008.

[3] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. "BRAIN - A Framework for Flexible Role-Based Interactions in Multiagent Systems." In: *CoopIS/DOA/ODBASE* 2888.Chapter 11 (2003), pp. 145–161.

[4] Giacomo Cabri, Letizia Leonardi, and Franco Zambonelli. "Separation of Concerns in Agent Applications by Roles." In: *ICDCS Workshops* (2002), pp. 430–435.

[5] Jacek Cała. "Migration in CORBA Component Model." In: *Distributed Applications and Interoperable Systems* 4531.Chapter 11 (2007), pp. 139–152.

[6] Robert N Charette. "This car runs on code." In: *IEEE spectrum* 46.3 (2009), p. 3.

[7] Betty H C Cheng and Ji Zhang. "Specifying adaptation semantics." In: *ACM SIGSOFT Software Engineering Notes* 30.4 (May 2005), pp. 1–7.

[8] Philipp Chrszon et al. "Family-Based Modeling and Analysis for Probabilistic Systems - Featuring ProFeat." In: *FASE 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. 2016, pp. 287–304.

[9] Alan W Colman and Jun Han. "Roles, players and adaptable organizations." In: *Applied Ontology* (2007).

[10] Mahdi Derakhshanmanesh et al. "GRAF: Graph-based Runtime Adaptation Framework." In: *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. New York, NY, USA: ACM, 2011, pp. 128–137.

[11] Simon Dobson et al. "A survey of autonomic communications." In: *ACM Trans. Auton. Adapt. Syst.* 1.2 (Dec. 2006), pp. 223–259.

[12] J Ferber, F Michel, and J Báez. "AGRE: Integrating Environments with Organizations." In: *Environments for Multi-Agent Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, July 2004, pp. 48–56.

[13] D Garlan et al. "Rainbow: architecture-based self-adaptation with reusable infrastructure." In: *Computer* 37.10 (Oct. 2004), pp. 46–54.

[14] Ioannis Georgiadis, Jeff Magee, and Jeff Kramer. "Self-organising software architectures for distributed systems." In: *Proceedings of the first workshop on Self-healing systems*. New York, New York, USA: ACM, Nov. 2002, pp. 33–38.

[15] Hassan Gomaa and M Hussein. "Software reconfiguration patterns for dynamic evolution of software architectures." In: *Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA 2004)*. IEEE, 2004, pp. 79–88.

[16] Hassan Gomaa et al. "Software adaptation patterns for service-oriented architectures." In: *SAC* (2010), pp. 462–469.

[17] Robrecht Haesevoets. "Macodo: Architecture-Centric Support for Dynamic Service Collaborations." PhD thesis. Jan. 2012.

[18] Svein O Hallsteinsen, Jacqueline Floch, and Erlend Stav. "A Middleware Centric Approach to Building Self-Adapting Systems." In: *Gschwind T., Mascolo C. (eds) Software Engineering and Middleware. SEM 2004. Lecture Notes in Computer Science* 3437.Chapter 9 (2004), pp. 107–122.

[19] Rolf Hennicker and Annabelle Klarl. "Foundations for Ensemble Modeling - The Helena Approach - Handling Massively Distributed Systems with ELaborate ENsemble Architectures." In: *Specification, Algebra, and Software* 8373.Chapter 18 (2014), pp. 359–381.

[20] A Hinton et al. "PRISM: A Tool for Automatic Verification of Probabilistic Systems." In: *TACAS'06*. 2006, pp. 441–444.

[21] Tobias Jäkel et al. "Position Paper - Runtime Model for Role-Based Software Systems." In: *ICAC* (2016).

[22] Tobias Jäkel et al. "RSQL - a query language for dynamic data types." In: *IDEAS* (2014), pp. 185–194.

[23] Jeffrey O Kephart and David M Chess. "The Vision of Autonomic Computing." In: *Computer* 36.1 (Jan. 2003), pp. 41–50.

[24] Annabelle Klarl. "Engineering Self-Adaptive Systems with the Role-Based Architecture of Helena." In: *WETICE Workshops* (2015), pp. 3–8.

[25] Jeff Kramer and J Magee. “Self-Managed Systems: an Architectural Challenge.” In: *Future of Software Engineering, 2007. FOSE ’07.* Minneapolis: IEEE, May 2007, pp. 259–268.

[26] Jeff Kramer and J Magee. “The evolving philosophers problem: Dynamic change management.” In: *Software Engineering, IEEE Transactions on* 16.11 (1990), pp. 1293–1306.

[27] Christian Krupitzer et al. “A survey on engineering approaches for self-adaptive systems.” In: *Pervasive and Mobile Computing* 17 (2014), pp. 184–206.

[28] Thomas Kühn et al. “A Metamodel Family for Role-Based Modeling and Programming Languages.” In: *Software Language Engineering.* Cham: Springer International Publishing, Sept. 2014, pp. 141–160.

[29] Rogério de Lemos et al. “Software Engineering for Self-Adaptive Systems: A Second Research Roadmap.” In: *Software Engineering for Self-Adaptive Systems* 7475.Chapter 1 (2010), pp. 1–32.

[30] Max Leuthäuser and Uwe Aßmann. “Enabling View-based Programming with SCROLL: Using roles and dynamic dispatch for etablishing view-based programming.” In: *MORSE/VAO ’15: Proceedings of the 2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering.* ACM Request Permissions, July 2015.

[31] Nancy A Lynch. *Distributed algorithms.* [Nachdr.] The Morgan Kaufmann series in data management systems. San Francisco, Calif. : Morgan Kaufmann, 2006.

[32] Sam Malek, Marija Mikic-Rakic, and Nenad Medvidovic. “A Decentralized Redeployment Algorithm for Improving the Availability of Distributed Systems.” In: *Component Deployment.* Berlin, Heidelberg: Springer Berlin Heidelberg, Nov. 2005, pp. 99–114.

[33] D Menasce et al. “SASSY: A Framework for Self-Architecting Service-Oriented Systems.” In: *Software, IEEE* 28.6 (Nov. 2011), pp. 78–85.

[34] Henry Muccini, Mohammad Sharaf, and Danny Weyns. “Self-adaptation for cyber-physical systems: a systematic literature review.” In: *SEAMS ’16: Proceedings of the 11th International Workshop on Software Engineering for Adaptive and Self-Managing Systems.* New York, New York, USA: ACM, May 2016, pp. 75–81.

[35] P Oreizy et al. “An architecture-based approach to self-adaptive software.” In: *IEEE Intelligent Systems* 14.3 (May 1999), pp. 54–62.

[36] Christian Piechnick et al. “Using role-based composition to support unanticipated, dynamic adaptation-smart application grids.” In: *Proceedings of . . .* 2012.

[37] Romain Rouvoy et al. “MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments.” In: *Software Engineering for Self-Adaptive Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 164–182.

[38] Mazeiar Salehie and Ladan Tahvildari. “Self-adaptive Software: Landscape and Research Challenges.” In: *ACM Trans. Auton. Adapt. Syst.* 4.2 (May 2009), 14:1–14:42.

[39] Alexander Schill and Thomas Springer. *Verteilte Systeme :Grundlagen und Basistechnologien /*. eXamen.press. Berlin ;, Heidelberg [u.a.] : Springer, 2007.

[40] Friedrich Steimann. “On the representation of roles in object-oriented and conceptual modelling.” In: *Data Knowl. Eng. ()* 35.1 (2000), pp. 83–106.

[41] Daniel Sykes, Jeff Magee, and Jeff Kramer. “FlashMob: distributed adaptive self-assembly.” In: *Proceeding of the 6th international symposium*. New York, New York, USA: ACM, May 2011, pp. 100–109.

[42] Nguonly Taing et al. “A dynamic instance binding mechanism supporting run-time variability of role-based software systems.” In: *Companion the 15th International Conference*. New York, New York, USA: ACM, Mar. 2016, pp. 137–142.

[43] Nguonly Taing et al. “Consistent Unanticipated Adaptation for Context-Dependent Applications.” In: *the 8th International Workshop*. New York, New York, USA: ACM, July 2016, pp. 33–38.

[44] Tetsuo Tamai and Supasit Monpratarnchai. “A Context-Role Based Modeling Framework for Engineering Adaptive Software Systems.” In: *APSEC* 1 (2014), pp. 103–110.

[45] Yves Vandewoude et al. “Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates.” In: *IEEE Transactions on Software Engineering* 33.12 (Dec. 2007), pp. 856–868.

[46] Pieter Vromant, Danny Weyns, and Sam Malek. “On interacting control loops in self-adaptive systems.” In: *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. Catholic University of Leuven. ACM, 2011.

[47] Martin Weissbach. “Adaptation Mechanisms for Role-Based Software Systems.” In: *OTM Workshops* 9416.Chapter 1 (2015), pp. 3–4.

[48] Martin Weissbach and Thomas Springer. “Coordinated Execution of Adaptation Operations in Distributed Role-based Software Systems.” In: *SAC 2017: Symposium on Applied Computing Proceedings*. New York, NY, USA: ACM, 2017, pp. 45–50.

[49] Martin Weissbach et al. “Decentralized coordination of dynamic software updates in the Internet of Things.” In: *WF-IoT* (2016).

[50] Martin Weissbach et al. “Decentrally Coordinated Execution of Adaptations in Distributed Self-Adaptive Software Systems.” In: *2017 IEEE 11th International Conference on Self-Adaptive and Self-Organizing Systems (SASO*. IEEE, 2017, pp. 111–120.

[51] Danny Weyns et al. “On Patterns for Decentralized Control in Self-Adaptive Systems.” In: *Software Engineering for Self-Adaptive Systems II*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 76–107.

[52] Markus Wutzler, Martin Weissbach, and Thomas Springer. “Role-Based Models for Building Adaptable Collaborative Smart Service Systems.” In: *2017 IEEE International Conference on Smart Computing (SMARTCOMP*. IEEE, 2017, pp. 1–6.

[53] Ji Zhang and Betty H C Cheng. “Model-based development of dynamically adaptive software.” In: *Proceedings of the 28th international conference on Software engineering*. 2006.