Martin Weißbach, Philipp Chrszon, Thomas Springer, Alexander Schill

# Decentrally Coordinated Execution of Adaptations in Distributed Self-Adaptive Software Systems

**Erstveröffentlichung in / First published in:**

*International Conference on Self-Adaptive and Self-Organizing Systems (SASO).* Tucson, 18. – 22.09.2017. IEEE Xplore, S. 111 – 120. ISBN 978-1-5090-6555-4.

DOI: https://doi.org/10.1109/SASO.2017.20

Final edited form was published in "International Conference on Self-Adaptive and Self-Organizing Systems (SASO). Tucson 2017", S. 111 - 120.
ISBN: 978-1-5090-6555-4
https://doi.org/10.1109/SASO.2017.20

# Decentrally Coordinated Execution of Adaptations in Distributed Self-Adaptive Software Systems

Martin Weißbach*, Philipp Chrszon†, Thomas Springer*, and Alexander Schill*
*Computer Networks Group, Technische Universität Dresden, Dresden, Germany
†Chair of Algebraic and Logical Foundations, Technische Universität Dresden, Dresden, Germany
{martin.weissbach1,philipp.chrszon,thomas.springer,alexander.schill}@tu-dresden.de

***Abstract*—Software systems in domains like Smart Cities, the Internet of Things or autonomous cars are coined by a high degree of distribution across several independent computing devices and the requirement to be able to adjust themselves to varying situations in their operational environment. Self-adaptive software systems are a natural choice to implement such context-dependent software systems. A multitude of approaches already implement self-adaptive systems and some consider even distribution aspects. Yet, none of the existing solutions supports the coordination of adaptation operations spanning multiple independent nodes, which is necessary to ensure a consistent adaptation even in presence of network errors or node failures.**

**In this paper, we tackle this challenge to execute adaptations in distributed self-adaptive software systems in a coordinated manner. We present a protocol that enables the self-adaptive software system to execute correlated adaptations on multiple nodes in a transactional manner ensuring an atomic and consistent transition of the distributed system from its source to the desired target configuration.**

**The protocol is validated to be free of deadlocks for any given adaptation at any point in time using a model-checking approach. The performance of our approach is investigated in experiments that emulate the protocol's execution on real devices for different sizes of distributed applications and adaptation scenarios.**

## I. Introduction

Self-adaptive software systems are an established approach to model and implement software systems that have to modify their own computational behavior or structure in response to changes in their operational environment. Applications that are expected to modify their own behavior, e.g., from domains like Smart Cities, the Internet of Things or autonomous cars, are coined by a high degree of both logical and spatial distribution. Such applications are furthermore highly interconnected, i.e., distributed parts of the system exchange messages and information to provide their intended functionality. In response to changes in the operational environment (i.e., context) the self-adaptive software system is likely to be required to adapt several application parts located on different nodes. Being able to control the adaptation process without a central control unit in an otherwise distributed environment is an essential feature for self-adaptive software systems [1], [2] to overcome performance bottleneck and single-point of failure issues every centralized approach is prone to.

In this regard, we focus our work on the distributed and decentralized execution of adaptations at run time, which means that the components of the self-adaptive software system responsible for the execution of run-time adaptations, can only adapt local parts of the distributed system and need to collaborate in order to perform the adaptation [2]. Our research so far was concerned with the decentralized execution of adaptations in self-adaptive software systems [3], i.e., components responsible for the execution of adaptations are only able to perform adaptations locally, but have to collaborate with each other to control the execution process.

Software systems able to modify their own behavior in response to changes in their operational environment are usually coined by *static system behavior* that is fix and does not depend on external properties and dependent *dynamic system behavior* that may be dynamically added or removed to or from the system. Such context-dependent adaptations are usually summarized as *structural adaptations* in contrast to *parameter adaptations*, which only allow to set parameters of an algorithm or a component, for example, to modify the respective system behavior [4]. The approach presented in this paper is concerned with *structural adaptations* of distributed self-adaptive software systems. In order to capture dynamic and static system behavior, we utilize the abstraction of *Roles* and *Players*, which offers a wide set of features to model and implement context-dependent applications [5]. A *Role* in our approach has its own state and behavior and captures context-dependent functionality of the application whereas a *Player* implements static, context-independent functionality of the application. By playing different roles, the perceivable behavior of the player can be modified context-dependently. A structural adaptation of the self-adaptive software system resulting in a behavioral change is thus achieved by modifying the *plays* relation between players and roles. Using the notion of roles to abstract from the concrete implementation of the adaptable application parts is not a limiting factor to our approach with respect to the applicability to the existing work. In [6], the notion of roles is applied to component-based software systems whereas [7] translates roles to services and [8] uses the abstraction on the level of Java objects to design and implement self-adaptive software systems.

In this paper, we describe our approach to coordinate the execution of adaptations in distributed self-adaptive software systems. Our approach includes a set of role-based adaptation operations that describe atomic adaptation steps. We introduced the term *Adaptation Transaction* [3] to encapsulate dependent adaptation operations in order to ensure a consistent transition of the self-adaptive software system from a source

Final edited form was published in "International Conference on Self-Adaptive and Self-Organizing Systems (SASO). Tucson 2017", S. 111 - 120.
ISBN: 978-1-5090-6555-4
https://doi.org/10.1109/SASO.2017.20

to a target configuration (Section II). The execution of such an adaptation transaction is controlled using our proposed *Coordination Protocol* to allow for an adaptation without central coordinator. In this paper, we extend the coordination protocol to ensure a consistent and decentralized execution of adaptation transactions in unstable environments coined by unreliable communication channels used to coordinate the execution of adaptations. We address the issue of lost coordination messages during the adaptation process to advance our previously presented ideas in this matter and elaborate on the protocol's behavior in the presence of random adaptation errors occurring at run time to ensure a consistent application configuration (Section III). We validate our protocol using a formal model checking approach to prove the proposed protocol to be deadlock free even in failure scenarios and perform emulated performance measurements of our protocol executing adaptation transactions of different size in different system sizes (Section IV). Subsequently, we focus on related work and critically discuss our approach (Sections V and VI, respectively) before we conclude this paper (Section VII).

## II. Foundations

Our previous research [9], [3] serves as foundation we rely on in this paper and will be briefly summarized in the following. First, we will outline the system architecture and the assumed system model for our research. Subsequently, the terms *Adaptation Operation* and *Adaptation Transaction* will be discussed, which describe the concrete adaptations to be executed. Lastly, the interface to locally modify parts of the distributed software system will be briefly summarized.

### A. System Model and Architecture

We assume the self-adaptive software system to follow an *external adaptation* [4] approach, i.e., application specific concerns are strictly separated from adaptation concerns, e.g., adaptation mechanisms. The architecture of our proposed solution, which is depicted in Figure 1, reflects this consideration. A *Node* is a computational unit that hosts the distributed parts of the self-adaptive software system, which are the *Adaptation Manager* and the *Managed Application*. The *Managed Application* provides all the static and dynamic behavior of the application and is being adapted at run time in response to changes in the operational environment of the system. We assume the managed application to use the notion of roles and players as well to implement dynamic and static parts of the system, respectively. The managed application itself is distributed across several *Nodes*. Different parts of the application may perform different tasks at run time, i.e., the set of application behavior distributed across all nodes of the system is not assumed to be homogeneous. Each managed application maintains a run-time model of its local configuration including the set of players and the roles each player is playing as well as the local and remote collaborations of each role. Each local part of the managed application is accompanied by an instance of the *Adaptation Manager*, which can adapt the local subsystem of the managed application

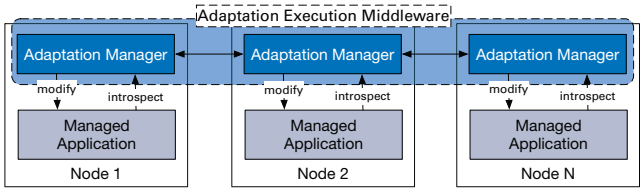



Figure 1. System Overview

but has to collaborate with other adaptation managers on remote nodes in order to perform the overall adaptation of the managed application as a whole.

With respect to Kephart and Chess [10], a node can be seen as an *autonomic element* in which the managed application represents the managed element. Since our approach focuses on the consistent execution of adaptations at run time, the adaptation manager can be mapped to the execution component of an autonomic manager implementing the MAPE-K feedback loop. Our approach, in contrast, is only concerned with decentralizing the execution phase and is thus a subsystem of the autonomic manager.

In terms of the underlying communication infrastructure of the system, we assume that each node in the system is able to reach any other node. The communication channel between nodes, however, is not expected to be reliable, i.e., messages exchanged by the adaptation managers to perform the adaptation, might be lost during transmission.

### B. Adaptation Operation and Transaction

We describe a *consistent application configuration* as a configuration of the managed application in which all static and dynamic behavioral parts adhere to a specific run-time model for the currently active context. In response to context changes, the adaptation management's *Planning* phase derives a system configuration that is consistent with the new context and issues an adaptation plan to the *Execution* phase of the feedback loop. We define the term *consistent adaptation* as the transition of the system from a given source configuration to a target configuration. Such a transition is described by the derived adaptation plan of the adaptation management's planning phase. Our work is focused on the *consistent adaptation* of the distributed self-adaptive software system, hence we expect the change plans to be provided as input to our approach.

We proposed the notion of an *Adaptation Transaction* [3] to describe the necessary adaptations to transition a distributed software system from a source to a target configuration. An adaptation transaction is composed of multiple *Adaptation Operations* that describe a single adaptation step, e.g., adding or removing context-dependent behavior, i.e., roles, to or from the running system [3]. In order to ensure a consistent adaptation, an adaptation transaction is executed atomically, i.e., the execution cannot be interrupted, and either all adaptation operations are executed successfully or none of the adaptation operations take effect. Using this atomic execution approach, an adaptation transaction ensures a consistent application con-

Final edited form was published in "International Conference on Self-Adaptive and Self-Organizing Systems (SASO). Tucson 2017", S. 111 - 120.
ISBN: 978-1-5090-6555-4
https://doi.org/10.1109/SASO.2017.20

figuration. Evidently, rollback mechanisms to revert temporary changes are supported by the execution middleware as well. Frequent context changes possibly require adaptation transactions to be serialized before execution or to be performed otherwise in isolation to maintain the consistency of the managed application. Currently, however, we do not support the *isolation* of adaptation transactions.

### C. The Local Adaptation Interface

We follow an external adaptation approach [4], which allows for a clear separation of adaptation and application concerns. The notion of roles to distinguish context-dependent system behavior from context-independent behavior, i.e., players, is reflected in the interface between adaptation manager and managed application as well. The interface allows the adaptation manager to monitor the internal configuration of the managed application, i.e., information about players and the roles currently played by them can be obtained as well as internal state information of any given role instance, which is required for certain types of adaptation operations, e.g., the relocation of a role instance between devices [3]. Furthermore the interface allows the adaptation manager to locally adapt the managed application, i.e., creating new role instances and establishing a *plays* relation to a given player or removing role instances deemed discardable by the adaptation management from a player. Especially the last step includes mechanisms to transition the role to a quiescent state [11] or to otherwise ensure the role to be in a state in which no computational tasks are performed. Since roles encapsulate application behavior that can be executed at any time, the interface needs to enable the adaptation management to observe and affect the execution state of roles to allow for a consistent adaptation at run time. We will elaborate on the protocol support to reach such a state for the overall adaptation process in the following section.

## III. Decentralized Coordination Protocol

The previously discussed adaptation operations and transactions describe the modifications the decentralized execution middleware is expected to perform on the managed application. In this section, we focus on the *Decentralized Coordination Protocol* that enables the execution middleware to perform adaptations without a central control system. We first outline the protocol's support to reach a stable application state before the actual adaptation process commences. Subsequently, we discuss the protocol and its coordination messages to ensure a consistent adaptation covering local adaptation failures. Lastly, the extended protocol behavior to cope with link failures, e.g., lost coordination messages during the execution of adaptation transactions, will be presented.

### A. Stable Application State Support

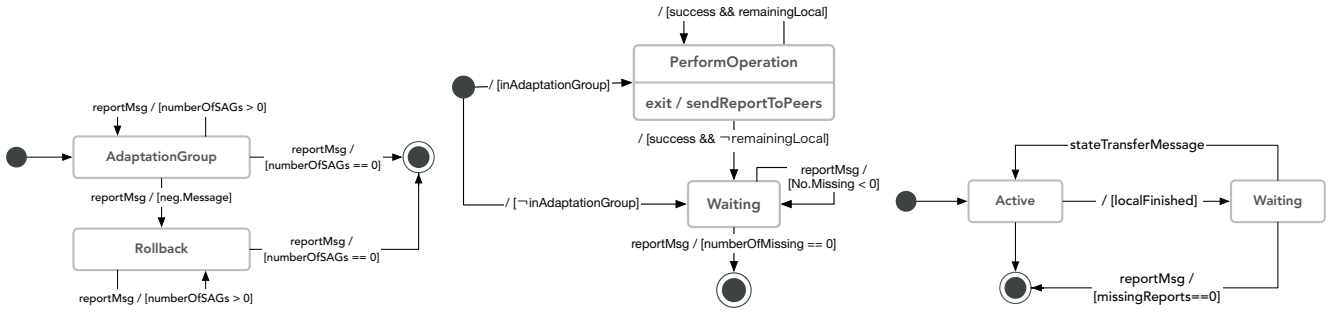
The adaptation of a running program requires the program to reach a *stable application state* [12] first. In such a state no parts of the program that are subject to the adaptation perform computational tasks and all data has been saved. The concept of *quiescence* [11] proposes a set of criteria that specify when a given adaptable entity, e.g., a component, has reached a stable state. Consequently, any coordination protocol needs to ensure all context-dependent parts of the managed application, i.e., roles, affected by an adaptation to reach a quiescent state. In terms of our approach, a role instance that is part of an adaptation operation and thus an adaptation transaction, has to reach such a quiescent state before it can be removed or relocated, for example. Players are able to play multiple roles simultaneously. Roles that are not subject to the ongoing adaptation process can be continued to be actively played by the system's player resulting in only fractions of the system affected by the consequences of a quiescent state.

In a distributed application, roles also collaborate with other roles located on remote nodes in order to provide the application's intended functionality. In [7], this relational nature of roles is explored in detail from an engineering perspective for collaborative software systems. Since an *Adaptation Transaction* only specifies the concrete adaptation operations supposed to be performed by the adaptation management, the *Adaptation Managers* have to determine a quiescent application state for the managed application. In order to ensure a consistent adaptation of the application, the adaptation managers need to obtain information on which locally managed roles collaborate with roles on remote nodes. When the information has been obtained, the coordination protocol provides the *Passivate* message, which is sent to all adaptation managers managing roles supposed to be passivated locally. The successful passivation of a role locally is indicated using the *Report* message, which is used to disseminate status information among adaptation managers. As soon as a role was successfully passivated, the adaptation of that specific role can commence.

### B. The Coordination Protocol

The coordination protocol provides the following messages, which were partially presented in [3], to coordinate the progress of an adaptation transaction: *Report* messages are used to disseminate status information on the execution of local parts of individual adaptation operations as well as on the adaptation transaction. A *Report* message contains the unique identifier of the adaptation operation as well as a status flag to indicate, whether the adaptation step was performed successfully, not successfully or whether the execution is still ongoing. Unsuccessful report messages will cause the adaptation transaction to be terminated and thus the managed application to remain in its source configuration. *StateTransfer* messages are used to transfer internal state information of a role from the source to the target node. It contains the identifier of the adaptation operation which requires the state transfer and the state information itself. *TransactionActivation* messages are issued when all adaptation operations of the adaptation transaction could be performed successfully. This message triggers the activation of the managed application's behavior of the target configuration. *TransactionRollback* messages are used in response to a negative report message and will cause all adaptation managers to rollback temporary

Final edited form was published in "International Conference on Self-Adaptive and Self-Organizing Systems (SASO). Tucson 2017", S. 111 - 120.
ISBN: 978-1-5090-6555-4
https://doi.org/10.1109/SASO.2017.20




(a) Transaction State Chart (b) Adaptation Group State Chart (c) Local Operation

Figure 2. The Coordination Protocol State Chart of an Adaption Manager

modifications locally made. Finally, the source configuration's behavior of the managed application will be reactivated.

The coordination protocol for an adaptation transaction is depicted in Figure 2(a). The received adaptation transaction is structured into *Adaptation Groups*, which contain adaptation operations that are allowed to be executed in parallel. The `Order` parameter of the adaptation operation (cf. [3]) indicates if adaptation operations can be executed in parallel, i.e., adaptation operations of the same order belong to the same adaptation group. The adaptation groups are executed sequentially and the next group is started as soon as all report messages the respective adaptation manager expected for the current adaptation group have been received indicating a successful execution. Otherwise, the adaptation transaction will be terminated unsuccessfully and the *Rollback* state will be entered in which all temporary adaptations are reverted.

If an adaptation group does not contain any operations that require the adaptation manager to coordinate local adaptations, the adaptation manager immediately enters the *Waiting* state in which it remains idle until all expected report messages were received. If the adaptation manager is required to perform local adaptations based on the specified adaptation operations within the adaptation group, the adaptation manager enters the *PerformOperation* state. In this state, adaptation operations are executed according to the state chart depicted in Figure 2(c). The concrete execution behavior and the interplay of the adaptation manager with the managed application through the shared interface, which we briefly discussed earlier, was described in [3]. Local modifications are made in the *Active* state and the state chart is immediately exited for local adaptation operations such as the addition or removal of context-dependent behavior. If the collaboration of a remote adaptation manager is required, the *Waiting* state of the local operation state chart will be entered, in which the adaptation manager waits for progress reports from the remote adaptation manager. When the local operation was performed, a report message is sent to all peers of the adaptation transaction. If further adaptation operations are supposed to be performed locally (indicated by the *remainingLocal* condition in Figure 2(b)), the respective adaptation manager remains in the *Perform Operation* state. The adaptation manager enters the *Waiting* state otherwise until all *Report* messages have been received. In that case, the adaptation manager will continue with the execution of the next adaptation group if there is another adaptation group to be executed or terminate the adaptation transaction successfully otherwise cf. Figure 2(a).

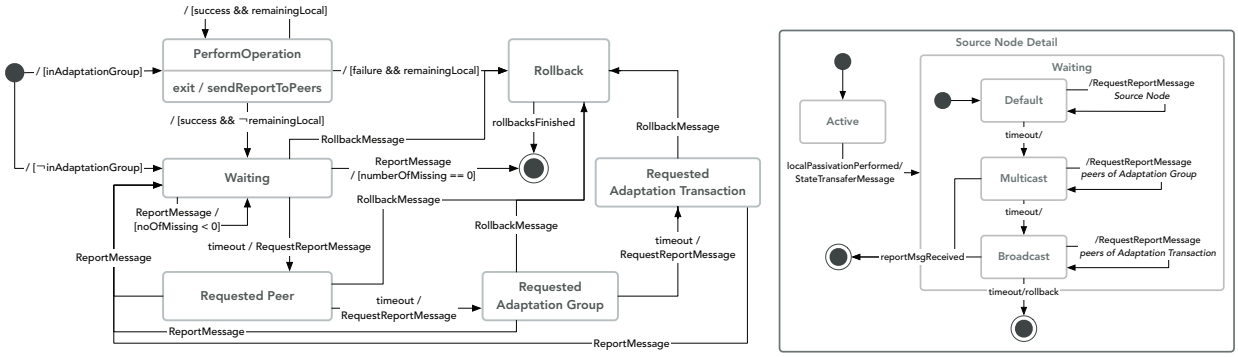
In case of adaptation failures at run time, e.g., a specific role cannot be bound successfully, the adaptation manager will react with the transmission of a *TransactionRollback* message to all peers, which terminates the adaptation transaction and all adaptation managers revert temporary changes resulting in the managed application remaining in the source configuration.

### C. Handling Link Failures

The coordination protocol presented so far is able to maintain a consistent configuration of the managed application in the presence of randomly occurring adaptation failures at run time using a transactional execution of adaptations with a rollback mechanism. The protocol ensures the application to either reach the desired target configuration or to remain unmodified in the source configuration.

Without an extended protocol behavior, the execution of an adaptation transaction would run into a deadlock while waiting for report messages if those messages would be lost during transmission. The part of the protocol that coordinates the execution of an adaptation transaction depicted in Figure 2(a), does not require any extension to cope with link failure since the main responsibility is the coordination of the execution of adaptation groups and the failure case behavior if an adaptation transaction was terminated unsuccessfully. The current execution state of an adaptation transaction, however, is determined within the adaptation group. Furthermore, no additional information except the *Report* messages are required to continue with the next adaptation group or to terminate the transaction. Thus, the part of the protocol to control the execution process of the overall adaptation transaction does not require any additional mechanisms to handle link failures. The other parts of the protocol coordinating the execution of adaptation groups and individual adaptation operations, in contrast, require additional mechanisms to detect and handle lost coordination messages.

Final edited form was published in "International Conference on Self-Adaptive and Self-Organizing Systems (SASO). Tucson 2017", S. 111 - 120.
ISBN: 978-1-5090-6555-4
https://doi.org/10.1109/SASO.2017.20




(a) Adaptation Group State Chart (b) Distributed Operation – Source Node

Figure 3. The Coordination Protocol State Chart of an Adaption Manager with extensions to handle Link Failures

We first consider the extended protocol behavior for the execution of adaptation groups depicted in Figure 3(a). Assuming all local adaptation operations of the adaptation group were performed successfully, the adaptation manager is in the *Waiting* state to collect report messages of other adaptation managers for all remaining operations that are part of the adaptation group. If no *Report* messages from peers were received after a timeout during the *Waiting* state, the adaptation manager sends a *RequestReport* message for every missing report on an adaptation operation to the adaptation manager that handles the execution of the respective operation. A *RequestReport* contains the identifier of the adaptation operation of which the Report is missing. The receiving adaptation manager will respond accordingly with the retransmission of the requested *Report* message. In case a report message was received, the protocol continues as described in the previous section unless further reports are missing. After the first timeout, another two escalations are performed by the adaptation manager. In a first step, a *RequestReport* message is issued to all adaptation managers within the same adaptation group and in a second step, after another timeout, all adaptation managers within the adaptation transaction are requested for the report. Assuming link failures to occur only sporadically, the different levels of escalation serve the purpose to reduce the number of messages transmitted to retrieve missing progress information from remote adaptation managers. Since report messages are always issued to all adaptation managers within a transaction, any adaptation manager is able to provide status information on any other adaptation operation if requested.

In the *Waiting* state or any other escalation state to obtain progress information on the execution of remote adaptation operations, the adaptation manager may receive a *Report* message for an adaptation operation with a higher value for the `Order` parameter, i.e., the adaptation operation belongs to an adaptation group that is supposed to be subsequently executed. Evidently, at least one remote adaptation manager was able to establish knowledge on the successful execution of the adaptation group still in execution by the local adaptation manager and continued with the execution of the next adaptation group. If such a report message is received, the local adaptation manager will consider its currently executing adaptation group as finished successfully and continues with the execution of the following one. Consequently, not every lost coordination message immediately results in a timeout that delays the execution of the adaptation transaction.

The same protocol behavior is implemented for the coordination of distributed adaptation operations between two adaptation managers performing the adaptations of the source and target node respectively (cf. [3]). The behavior differs for the target and source node of the distributed adaptation operation, though. After transitioning the role into a passive state the source node sends the internal state information to the target node and enters the *Waiting* state. The adaptation manager of the target node, in contrast, creates a new instance of the role, reports the success to the peer and awaits the transmission of the source node's role's state information. The adaptation manager of the source node follows a similar behavior as described for the execution of adaptation groups when waiting for *Report* messages from the target node's adaptation manager (cf. Figures 3(b)). Since the target node depends on the receipt of the state information from the source node's adaptation manager, the behavior differs. The protocol was designed to perform three requests as well, but all those requests are directed to the source node to obtain the missing state information. If either source or target node failed to obtain the required number of report messages for the execution of the adaptation operation, a *TransactionRollback* message will be issued to all peers of the adaptation transaction. No protocol extensions for the rollback of adaptation transactions are required in the case of local adaptation failures.

If the source node's adaptation manager happens to be temporarily isolated after the transmission of the role's internal state information and the successful receipt by the target node's adaptation manager, the source node might send a *TransactionRollback* message to the other adaptation managers. If the adaptation operation has already been reported successfully by the target node's adaptation manager, all peers that received the rollback instruction will send a report

Final edited form was published in "International Conference on Self-Adaptive and Self-Organizing Systems (SASO). Tucson 2017", S. 111 - 120.
ISBN: 978-1-5090-6555-4
https://doi.org/10.1109/SASO.2017.20

message for the respective operation to the source node's adaptation manager and ignore the rollback, thus, continuing the execution of the adaptation operation. If no information from the target node could be received by the other adaptation managers, the operation is considered to have failed and they send *TransactionRollback* messages to their peers as well and the adaptation transaction is considered terminated.

The worst case scenario for link failures is a successful transmission of *RequestReport* message and a complete loss of all responses. Assume $n$ denotes the number of adaptation managers involved in the execution of the adaptation transaction and $m$ denotes the number of adaptation managers collaborating to execute the currently active adaptation group. In the worst case are then $2n+2m+2$ messages transmitted per executed operation within an adaptation group if the adaptation manager is not involved in the execution of the adaptation operation or if the adaptation manager handles the source node adaptation. From a target node's perspective of a distributed operation, 6 additional messages are transmitted in total.

## IV. Evaluation

Our approach to evaluate the Coordination Protocol will be presented within this section. First, we formally verify the protocol to be free of deadlocks during its execution, which means every executed adaptation transaction either finishes successfully or the coordination protocol terminates the execution and the managed application remains in its source configuration. Subsequently, we focus on the evaluation of the performance of our proposed protocol in scenarios that differ with respect to the number of nodes involved in the adaptation transaction, the number of adaptation operations within an adaptation transaction and the probability of lost communication messages.

### A. Formal Verification

We created a formal model of the adaptation protocol and established its deadlock freedom by means of *model checking* [13]. 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[1] [14] modeling language. ProFeat extends the input language of the probabilistic model checker PRISM[2] [15] 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 our model[3]. The model represents one or more nodes with their respective adaptation managers. The behavior of the adaptation managers follows the state charts depicted in Figure 2. 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 role, remove role and migrate role adaptation operation (cf. [3]), which are executed according to a specified scenario, i.e., the adaptation transaction describing the roles affected by the change and the adaptation managers responsible for the adaptation transaction's execution. In order to handle message loss, the modeled adaptation managers also implement the first protocol extension, where a *RequestReport* message is sent to peers in case of missing *Report* messages. The two additional escalations (where a request is sent to all members of the adaptation group and the transaction) are not modeled. 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.

We have analyzed a model instance with 3 nodes, 2 roles and an adaptation consisting of one role transfer and one local operation (add or remove role). In order to keep the model small, it only describes a single transaction with exactly one adaptation group. However, since adaptation groups are executed sequentially the analysis results also apply to multiple transactions with possibly more than one adaptation group. We could establish that the protocol never runs into a deadlock. Furthermore, the analysis showed that the adaptation is always successful in case of no message loss.

### B. Performance Evaluation

Having established the protocol to be free of deadlocks while performing an adaptation transaction, we subsequently evaluate the performance of our approach using an emulated setup to analyze the duration of adaptation operations and the unavailability of roles that are subject to the adaptation. First, aspects of the protocol's prototypical implementation important for the execution of the emulation will be stated. Subsequently, the setup of the experiments will be described and the obtained results will be presented and discussed.

*1) Implementation:* The adaptation managers were implemented on the Java Virtual Machine (JVM) 1.8 using the JVM's standard UDP implementation for the message-based communication to manage the adaptation process. A time of 2.5 seconds was used as timeout of the coordination protocol to decide if messages were lost during the execution of adaptation transactions. The distributed managed application was implemented using LyRT [16], which is a role-based runtime based on Java allowing the dynamic binding of roles, i.e., context-dependent behavior, to players, i.e., static system behavior. We relied on interprocess communication between each adaptation manager and its respective managed application to execute local adaptations. We extended the LyRT

[1] https://wwwtcs.inf.tu-dresden.de/ALGI/PUB/ProFeat/
[2] http://www.prismmodelchecker.org/
[3] https://www.rn.inf.tu-dresden.de/martin/saso-17/

Final edited form was published in "International Conference on Self-Adaptive and Self-Organizing Systems (SASO). Tucson 2017", S. 111 - 120.
ISBN: 978-1-5090-6555-4
https://doi.org/10.1109/SASO.2017.20

runtime to support the communication interface we discussed earlier for that purpose.

*2) Experiment Setup:* The experiment was conducted on one of our lab's computers equipped with 16GB RAM and 4 CPU cores running Debian 8. To simulate different sizes of the distributed system in terms of devices (nodes), we utilized Docker[4] to easily scale the size of the experiments with respect to the number of nodes involved in the experiments. Previous experiments were conducted with a very limited system and transaction size [3], which we wished to extend. In the experimental setup, each Docker container served as a node hosting an instance of the adaptation manager and the managed application each. All Docker containers shared a common network, which allowed them to directly communicate with each other. As a consequence, the network latency can be neglected for the measured results due to this virtualized network of Docker containers in which the distributed self-adaptive software system is emulated. The experiment itself was controlled from a tool hosted on the Docker host, communicating with each container using a predefined port to obtain information about the progress of the experiments' execution.

Proposed solutions to generate adaptation plans without a central management instance suggested system sizes comprising up to 100 nodes [17], [18]. We argue that not all nodes involved in the decision-making process will actually have to perform structural adaptations in response to a context change and chose node sizes below that threshold for our experiments. Following this consideration, we decided to run the coordination protocol for adaptation transactions including 10 and 20 nodes, to which we will refer to as *system size*.

For each system size, we generated different sizes of adaptation transactions with respect to the number of adaptation operations to be executed as depicted in Table I (*transaction size*). The composition of the adaptation transactions, representing different workloads or adaptation scenarios of real world applications, was set up as follows: The first application scenario, named *Workload 1* (WL 1), contains solely distributed adaptation operations, e.g., context-dependent behavior is migrated between nodes (cf. experiments #1, 5, 9, 13). The second application scenario (WL 2) contains only adaptations that can be executed locally on one device (cf. experiments #2, 6, 10, 14). The third and last application scenario we investigated contains both local and distributed adaptation operations. A composed adaptation transaction in that class of scenario contains three types of adaptation operations, namely add, remove and migrate behavior, all equally represented within the adaptation transaction. These scenarios were applied to both system sizes with different transaction sizes resulting in adaptation transactions containing twice as much (WL 3a) and five times as much (WL 3b) operations as nodes within the respective system. The adaptation transactions were generated automatically and adaptation operations within a generated adaptation transaction were distributed over at most 10 adaptation groups, i.e., the value of an adaptation operation's `Order` parameter ranged from $\{0..9\}$. Consequently, adaptation transactions with more adaptation operations may perform better since the ratio of adaptation operations that can be executed in parallel is higher for more adaptation operations and a fixed number of adaptation groups.

Table I
TRANSACTION AND SYSTEM SIZES USED FOR EXPERIMENTS AND THE DEGREE OF MESSAGE LOSSES ON THE CHANNEL.

| # | Transaction Size | System Size | Message Loss |
|---|---|---|---|
| 1 | 5 (*WL 1*) | 10 | 0 |
| 2 | 10 (*WL 2*) | | |
| 3 | 20 (*WL 3a*) | | |
| 4 | 50 (*WL 3b*) | | |
| 5 | 5 (*WL 1*) | | 10 |
| 6 | 10 (*WL 2*) | | |
| 7 | 20 (*WL 3a*) | | |
| 8 | 50 (*WL 3b*) | | |
| 9 | 10 (*WL 1*) | 20 | 0 |
| 10 | 20 (*WL 2*) | | |
| 11 | 40 (*WL 3a*) | | |
| 12 | 100 (*WL 3b*) | | |
| 13 | 10 (*WL 1*) | | 10 |
| 14 | 20 (*WL 2*) | | |
| 15 | 40 (*WL 3a*) | | |
| 16 | 100 (*WL 3b*) | | |

The experiments were conducted for each system size with (10%) and without message loss of coordination messages. *Report* and *StateTransfer* message types were exposed to the random message loss since both are essential for the execution of the adaptation transaction. The same rules were applied to the *RequestReport* message type, which is issued if reports of peer adaptation managers could not be received. The message loss itself was configured by setting an `iptables`[5] rule in each Docker container for the inbound communication channel. *Report* messages are small enough in size to fit into a single UDP packet, which makes the usage of `iptables` a viable solution to simulate random message losses on the channel. As a consequence, the information transferred in a *StateTransfer* message had to be kept small enough to also fit into one UDP packet, because these messages would more often fail to be successfully transmitted than other messages, which would distort the results.

*3) Results:* Each experiment (i.e, #1 through #16 in Table I) was repeated 100 times and the duration of the adaptation process, which is the time between the receipt of the adaptation transaction and of all *Report* messages, was measured for every adaptation manager. For the evaluation of the protocol, which is focused on the execution of adaptation operations, we considered only context-dependent roles, i.e., roles did not collaborate with each other but solely modified their players' behavior context-dependently. Thus, roles are passivated as soon as their operation group is ready to execute.

In Figure 4, the results of experiments #1 through #8 are displayed for the execution with and without message loss on the communication channels between nodes. The average execution for WL 2 and WL 3a took approximately

[4]https://www.docker.com

[5]https://linux.die.net/man/8/iptables

Final edited form was published in "International Conference on Self-Adaptive and Self-Organizing Systems (SASO). Tucson 2017", S. 111 - 120.
ISBN: 978-1-5090-6555-4
https://doi.org/10.1109/SASO.2017.20

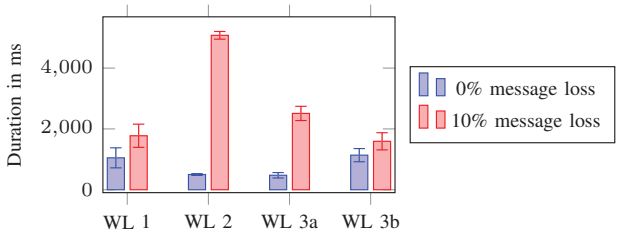



Figure 4. Average adaptation duration and standard deviation for a system size of 10 nodes per workload (WL).

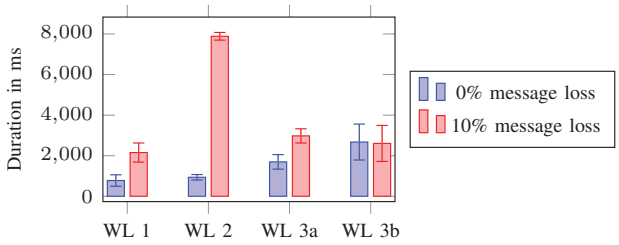



Figure 5. Average adaptation duration and standard deviation for a system size of 20 nodes per workload (WL).

501ms and 476ms, respectively. The slightly larger standard deviation for WL 3a in fact indicates both workloads not to differ significantly from each other. The additional effort required to coordinate the execution of distributed adaptation operations in WL 3a in contrast to WL 2 is compensated by the higher degree of parallel execution of adaptation operations. WL 3b contained five times the operations of WL 2, but only required approximately twice as long (1136ms) to perform an adaptation transaction in average, which is also a consequence of the higher degree of parallel execution per adaptation group.

In Figure 5, the results of experiments #9 through #16 are displayed for the execution with and without message loss on the communication channels between nodes. The observation of the execution time for workloads 3 can also be made for adaptation transactions involving the collaboration of 20 adaptation managers. WL 3b finished faster in average than WL 3a, but is coined by a greater standard deviation over all 100 iterations. Overall, the difference of the execution duration of workloads 3 is within 200ms considering the average of WL 3a and the upper bound of WL 3's standard deviation. It is also interesting to see the execution time for 10% message loss of WL 3b to be 68ms shorter than without message loss, which is can be explained with differing load on the nodes, e.g., caused by the Garbage Collection of Java. In general the results indicate an increasing execution time for adaptation transactions for a larger number of adaptation operations.

The results for workloads 1 and 2 are comparable for all performed tests, i.e., the execution took longer for a system size of 10 nodes and was roughly 150ms faster in average for a system size of 20 nodes. Since both workloads had exactly one adaptation manager perform exactly one local part of a distributed operation (WL 1) or one local adaptation operation

Table II
AVERAGE DOWNTIME OF MIGRATED APPLICATION BEHAVIOR FOR EXPERIMENTS #12 AND #16 IN SECONDS.

| # | Adaptation Group 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 12 | 2.4 | – | 1.3 | 1.1 | 1.0 | 0.8 | 0.6 | 0.5 | 0.4 | 0.2 |
| 16 | 2.4 | – | 1.2 | 1.0 | 0.9 | 0.7 | 0.6 | 0.4 | 0.3 | 0.2 |

(WL 2), the results are as expected. WL 1 requires a higher coordination effort due to the migrate operations whereas WL 2 contains more adaptation operations, which require only the dissemination of progress information.

In Table II the average downtime of the migrating roles is depicted for the two largest experiments, i.e., #12 and #16. The automatically generated adaptation transaction did not contain any adaptation operations configured with the value 1 for the `Order` parameter, which is the reason no values could be measured for the resulting adaptation group. A first evident observation is the decline of the role's downtime with increasing adaptation groups, which is expected since the coordination protocol only passivates a given role when it is due for adaptation and not as soon as the execution of the adaptation transaction commences. The second observation is the small offset of the measured downtime between both experiments, which indicates the protocol to tolerate 10% message loss well for the largest workload in the system.

Apparently, the impact of lost coordination messages is decreasing as the number of adaptation operations increases for a given system size, which holds true in particular for workloads 3, which have similar compositions of the executed adaptation transaction. Workload 2 contains adaptation groups that only contain one adaptation operation. This results in the poor performance of the experiments with message loss, since a single lost *Report* message immediately leads to a timeout period as knowledge of the successful execution of the of the previous adaptation group cannot be established.

In the performed experiments, role collaborations were not considered, i.e., roles could be immediately passivated. Both network latency and the required time to reach a stable state for the respective role part of the adaptation process would therefore have to be added to the results.

## V. RELATED WORK

A solid research body exists that focuses on the design and architecture of self-adaptive software systems (see [19]) including the MAPE-K feedback loop [10] as an architectural means to enable self-adaptability of software systems through self- and context-awareness. With an increasing degree of distribution of adapted applications, centralized adaptation management approaches become impractical, which leads to several efforts to distribute the adaptation management subsystem. Patterns of how to distribute phases of the feedback loop [2] or the decentralization of the decision making process have been the main research focus.

FlashMob [18] and DecAp [17] are examples of approaches that share information about the system's current configuration

Final edited form was published in "International Conference on Self-Adaptive and Self-Organizing Systems (SASO). Tucson 2017", S. 111 - 120.
ISBN: 978-1-5090-6555-4
https://doi.org/10.1109/SASO.2017.20

to decide in a decentralized manner when and how to (re-)assemble the system. A different approach by Georgiadis et al. [20] discusses the distributed execution of such changes through distributed component managers. The component managers synchronize using reliable broadcasts to publish join/ leave messages of components entering or leaving the system. Locks on the system, that ensures only one component manager at a time to be able to perform local changes on a globally consistent configuration, are also acquired through broadcasting. However, a component manager can only change its local component and a discussion how several component managers would collaborate to perform multiple changes in a consistent way is missing. Our approach takes the results of decentralized decision making approaches as input to consistently execute multiple dependent changes.

Since adaptations are performed at run time, a state has to be found in which it is safe to adapt the system or an entity. In [11] this state is called the *quiescent* state of a component and means the component not to be involved in any communication and is, thus, safe for removal or update, for example. Since the concept of quiescence imposes the restriction of several system components for one component to reach such a state, the conditions to reach this state have been relaxed to achieve less system interruption sacrificing the property to ever reach such a state; the resulting concept is referred to as *tranquility* [21]. Different adaptation semantics have been proposed that describe the system behavior formally when adaptations are about to be performed. In [12], three semantics are distinguished: (1) one-point adaptation (system behavior changes from one point in time to the next), (2) guided adaptation (the program is restricted to reach a state at which it can be adapted), and (3) overlap adaptation (a program temporarily exposes both old and new behavior with the old system being restricted in its functionally until the new behavior eventually takes over completely). Our approach makes use of these foundations to determine a point in time that allows the safe modification of role bindings.

In *multi-agent-systems*, roles serve as abstraction to describe collaborations of agents within a group of agents [22], [23]. Similarly, roles abstract from the concrete capabilities an agent has to provide in order to play a role. For example, in a peer-to-peer network, super peers can be established to structure the network, but a super peer is essentially only a role played by a peer with enough computational resources. Apart from that, the collaboration between super peer and its set of child peers clearly describes communication flows in the system. Due to the collaboration-centric notion of roles, an agent's possible behaviors remain static at run time and are either executed actively or not depending on the collaborations the agent participates in, but it is not possible to dynamically add behavior or roles to an agent that was not foreseen to be played by the agent at design time.

In summary, to the best of our knowledge, no current research effort exists that investigates how to perform multiple changes of a highly distributed software system in a coordinated and consistent manner across several devices. Please note that we are not concerned with the decision making process itself as it was discussed in [17], [18], but with the execution of such calculated change prescriptions at run time.

## VI. Discussion

In this paper, we proposed a *Decentralized Coordination Protocol* to ensure a reliable execution of multiple adaptation operations within a distributed self-adaptive software system without the need for a central management instance. We used the notion of an *Adaptation Transaction* to capture correlated adaptation operations and to define a scope that describes a consistent transition of the system from a source to a target configuration through these operations. We directly address the atomicity and consistency criteria well known from transactional systems and support the durability property but leave the implementation to the managed application of the self-adaptive software system. The isolation property is considered out-of-scope in this work and left for future investigation based on the assumption that a run-time adaptation can be finished before the next context change occurs. In highly dynamic software systems in which frequent context changes require the system to modify its structure, isolation becomes crucial when adaptations overlap but contradict each other.

The a-priori specification of the execution order of adaptation operations by the adaptation management's planning component is a strong assumption we made, which can be relaxed in the future for certain scenarios. Assume a role instance that is collaborating with other roles is supposed to be exchanged, migrated or removed in response to a context change. The (dis-)connection adaptation operations, which are required to be performed before and after the actual adaptation and which are still required to be specified explicitly, can be performed by the adaptation managers implicitly if information on role collaborations can be obtained from the managed application.

A role-based software system, which incorporates either quiescence [11] or tranquility [21], requires complete knowledge about the collaborations among roles and their run-time instances. Run-time models derived from design time models that are capable of specifying role collaborations, e.g, [7], [24], are an important prerequisite to achieve consistent adaptations of context-dependent, behavioral and relational software systems using the role-abstraction. Both quiescence and tranquility may not be a suitable criterium to define a stable application state for any role-based software system. We used the notion to express a state of a role in which it is safe to perform the adaptation operation on the role without negative side effects on the managed application. A more thorough investigation on reaching such a stable state for a collaborative role therefore remains an interesting research task for future work.

The experiments have furthermore shown that the duration of an adaptation transaction's execution is influenced by the amount of comprised adaptation operations and the degree of parallel execution of adaptation operations for a given system size. For many application scenarios, such as the coordinated exchange of application behavior on multiple nodes only little ordering is required during the execution of an adaptation

Final edited form was published in "International Conference on Self-Adaptive and Self-Organizing Systems (SASO). Tucson 2017", S. 111 - 120.
ISBN: 978-1-5090-6555-4
https://doi.org/10.1109/SASO.2017.20

transaction. Hence, our approach is best suited for systems that require a consistent execution of several adaptation operations in response to change in the computational environment, but require only little hierarchical structure among the performed adaptation operations. The unavailability of roles also becomes short if less adaptations are executed sequentially due to their specified `Order` within the adaptation transaction.

The average adaptation duration for the largest experiments, i.e., WL 3 for 20 nodes, is 5 times higher than for our previous experiments (cf. [3]), which is an acceptable result compared to the fact that 10 times more nodes and 100 times more adaptation operations were performed. The average downtime of a role was 3 to roughly 6 times higher compared to the previously performed experiments, but covers the execution of multiple adaptation groups. If only the last executed adaptation group is considered, the measured average downtime of roles involved in this group are comparable to the previously conducted experiments. Compared to the greatly enlarged size of both the involved nodes and the size of the adaptation transaction, the coordination protocol scales reasonably well.

## VII. Conclusion

In this paper we presented a *Decentralized Coordination Protocol* to control and manage the adaptation of a distributed self-adaptive software system. The protocol ensures a consistent adaptation of the managed application in the presence of link failures, i.e., lost coordination messages during the adaptation process, and local adaptation failures. We rely on the notion of *Roles* to describe adaptation operations supported by the coordination protocol and as an abstraction to separate static and dynamic behavior of the managed application. The term *Adaptation Transaction* was introduced to consistently perform adaptation operations at run time.

We validated our approach using a formal model checking technique to prove the coordination protocol to be free of deadlocks in the presence of lost coordination messages. Hence, the consistent adaptation is ensured in any case since the managed application can be ensured to reach the desired target state described by the adaptation transaction or the managed application is ensured to remain in the source configuration otherwise. The coordination protocol's performance was investigated in experiments that emulated the protocol's execution on real devices for different compositions and sizes of adaptation transactions and different system sizes.

Future research can resolve around the notion of *Roles* as an abstraction for adaptable entities to investigate the issue of reaching a quiescent or tranquil state for highly interconnected applications that require the adaptation of different interconnected parts of the application consistently. A relaxed consistency constraint for the execution of adaptation transactions would be interesting as well. Such a relaxed constraint could allow parts of the system to be activated while other parts of the system are still under adaptation providing compensation mechanisms in case of adaptation failures instead of a rollback.

## Acknowledgements

This work is funded by the German Research Foundation (DFG) within the Research Training Group "Role-based Software Infrastructures for continuous-context-sensitive Systems" (GRK 1907).

## References

[1] R. de Lemos *et al.*, "Software Engineering for Self-Adaptive Systems: A Second Research Roadmap." *Software Engineering for Self-Adaptive Systems*, vol. 7475, no. Chapter 1, pp. 1–32, 2010.

[2] D. 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.

[3] M. Weißbach and T. Springer, "Coordinated Execution of Adaptation Operations in Distributed Role-based Software Systems," in *Symposium on Applied Computing*. New York, NY, USA: ACM, 2017, pp. 45–50.

[4] M. Salehie and L. Tahvildari, "Self-adaptive Software: Landscape and Research Challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, pp. 14:1–14:42, May 2009.

[5] T. Kühn *et al.*, "A Metamodel Family for Role-Based Modeling and Programming Languages," in *Software Language Engineering*. Cham: Springer International Publishing, Sep. 2014, pp. 141–160.

[6] A. Klarl, "Engineering Self-Adaptive Systems with the Role-Based Architecture of Helena." *WETICE Workshops*, pp. 3–8, 2015.

[7] R. Haesevoets *et al.*, "Architecture-centric support for adaptive service collaborations," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 1, pp. 2–40, Feb. 2014.

[8] T. Tamai and S. Monpratarnchai, "A Context-Role Based Modeling Framework for Engineering Adaptive Software Systems." *APSEC*, vol. 1, pp. 103–110, 2014.

[9] M. Weißbach *et al.*, "Decentralized coordination of dynamic software updates in the Internet of Things," in *Internet of Things (WF-IoT), 2016 IEEE 3rd World Forum on*. IEEE, 2016, pp. 171–176.

[10] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003.

[11] J. Kramer and J. Magee, "The evolving philosophers problem: Dynamic change management," *IEEE Trans. Softw. Eng.*, vol. 16, no. 11, pp. 1293–1306, 1990.

[12] B. H. C. Cheng and J. Zhang, "Specifying adaptation semantics," *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–7, May 2005.

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

[14] P. 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.

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

[16] N. Taing *et al.*, "A dynamic instance binding mechanism supporting runtime variability of role-based software systems," in *Companion Proc. of the 15th Int. Conf. on Modularity*, ser. MODULARITY Companion 2016. New York, NY, USA: ACM, 2016, pp. 137–142.

[17] S. Malek *et al.*, "A Decentralized Redeployment Algorithm for Improving the Availability of Distributed Systems," in *Component Deployment*. Berlin, Heidelberg: Springer Berlin Heidelberg, Nov. 2005, pp. 99–114.

[18] D. Sykes *et al.*, "Flashmob: Distributed adaptive self-assembly," in *Proc. of the 6th Int. Symp. on Soft. Eng. for Adapt. and Self-Managing Systems*, ser. SEAMS '11. New York, NY, USA: ACM, 2011, pp. 100–109.

[19] C. Krupitzer *et al.*, "A survey on engineering approaches for self-adaptive systems," *Pervasive and Mobile Computing*, no. 0, pp. –, 2014.

[20] I. Georgiadis *et al.*, "Self-organising software architectures for distributed systems." New York, NY, USA: ACM, Nov. 2002, pp. 33–38.

[21] Y. Vandewoude *et al.*, "Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 856–868, Dec. 2007.

[22] M. Becht *et al.*, "ROPE - Role Oriented Programming Environment for Multiagent Systems." *CoopIS*, 1999.

[23] J. Odell *et al.*, "A Metamodel for Agents, Roles, and Groups," in *Agent-Oriented Software Engineering V*. Berlin, Heidelberg: Springer Berlin Heidelberg, Jul. 2004, pp. 78–92.

[24] M. Wutzler *et al.*, "Utilizing Role-based Models for On-Demand Composition of Smart Service Systems," in *Companion Proc. of the 1st Int. Conf. Programming '17*. New York, NY, USA: ACM, 2017.