IEEE Copyright Notice

This work has been published in *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, Tucson, AZ, USA, 2017, pp. 39-44.

DOI: 10.1109/FAS-W.2017.118
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8064094&isnumber=8064070

RoleDiSCo: A Middleware Architecture and Implementation for Coordinated On-Demand Composition of Smart Service Systems in Decentralized Environments

Markus Wutzler and Thomas Springer and Alexander Schill
Technische Universität Dresden
Faculty of Computer Science, Chair of Computer Networks
01062 Dresden, Germany
Email: markus.wutzler@tu-dresden.de, thomas.springer@tu-dresden.de, alexander.schill@tu-dresden.de

***Abstract*—Future smart computing environments will heavily rely on the collaboration of services, which are dynamically interconnected to Smart Service Systems in order to develop their full potential. Such computing environments are coined by a high degree of distribution and decentralization rendering a centralized control of service composition and system adaptation infeasible. Self-organizing Software Systems tackle this issue to automatically compose service systems at run time. Due to missing holistic system specifications, they constitute a mainly linear service chain, which contradicts the collaborative nature of Smart Service Systems. Thus, we proposed a Two-Phase Development Methodology, which includes a role-based collaboration specification, for engineering Smart Service Systems. In this paper, we present RoleDiSCo, a middleware for coordinated on-demand composition of Smart Service Systems in decentralized environments in order to bridge the gap between design and run time.**

1. Introduction

Future smart computing environments, *e.g.,* smart home, smart mobility, smart city or smart health, will heavily rely on the spontaneous collaboration of independently developed, autonomously operating services in order to develop their full potential. The increasing number of smart systems inevitably leads to a huge number of systems that potentially provide such services. While only a few systems may participate in a collaboration, the number of concurrent collaborations is considerably high and individual systems might participate in multiple collaborations simultaneously. This renders central management infeasible as it becomes a bottleneck and is no viable option for such large-scale, volatile environments.

Dynamically interconnecting independently developed systems in decentralized environments at run time is a challenging task. Protocols, such as UPnP [1], are limited to a fixed number of collaborating systems. *Self-organizing Software Systems* (SOSSs) [2] deal with the composition of dependent service systems with many autonomous services. Their structure emerges without central control by matching provided and required service ports. Due to missing holistic collaboration specifications, they constitute a linear service chain, delegating requests across services until a task is completed [3], but lack support for complex service collaborations at run time, such as binding multiple instances of the same service type simultaneously. Complex service collaborations require specifications, but those usually lead to monolithic systems that follow a closed-world assumption and are only dynamic within their specified system boundary.

In Smart Service Systems, independently developed services are dynamically interconnected on-demand at run time, forming a *complex service collaboration* to combine its individual subsystems' capabilities in order to develop their full potential. Smart Service Systems face three major challenges: First, independently developed and autonomously operating services will have to be composed on-the-fly to form complex application structures providing desired functionalities. However, developing the service and its *role* essential to the incorporating collaboration should remain as distinct as possible. Second, such a service collaboration will have to adapt to changes in the users' situation or its own computational environment. Finally, due to the high degree of distribution and decentralization, such systems will have to be composed and adapted without static central control.

Previously, we proposed a *Two-Phase Development Methodology* [4], including a *Role-based Collaboration Specification* (RBCS) for Smart Service Systems to address the first challenge. We also investigated context-dependent, structural adaptation [5] to address the second challenge.

In this paper, we tackle the third challenge. Elaborating on our previous explorations [6], we present RoleDiSCo, a middleware for coordinated on-demand composition of Smart Service Systems in decentralized environments.

2. Foundations

Roles [7] are an intuitive abstraction that perfectly matches the collaborative nature of Smart Service Systems. *Collaborations* are a foundation of the role concept, in which the role is an abstract functionality whose actual performance is delegated to the role's player. Players can be restricted through requirements specified by the role (*e.g.,* provided methods or contextual properties). Since this *plays* relationship can be dynamically established and detached at run time, dynamic system behavior can be achieved. Thereby, the role concept enables serendipity [3], *i.e.,* to deal with

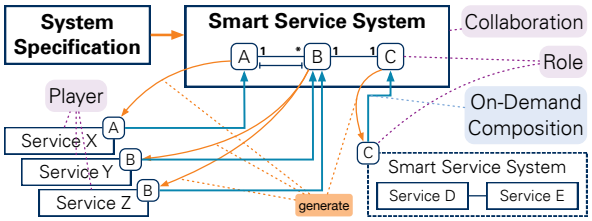

Figure 1. The RoleDiSCo Idea: Loose Coupling, Adaptivity, Serendipity.

integrating unforeseen services, already at design time. In other words, a role can be seen as a previously unknown plugin, enabling the player to join in a collaboration.

Figure 1 depicts the RoleDiSCo approach: A Smart Service System is specified as a collaboration of roles. A role encapsulates an abstract functionality which is performed by a player, *i.e.,* an independently developed service. Hence, the service can be considered the player of a role. The service, in turn, may play multiple roles in several collaborations.

2.1. Two-Phase Development Methodology

RoleDiSCo's *Two-Phase Development Methodology* [4] demarcates the service's development from that of its role required through the service's participation in various collaborations. In Phase ①, the collaboration designer specifies the Smart Service System as a self-contained collaboration using the proposed RBCS, in which roles capture the abstract functionality a service should provide to the collaboration. Thereof, a partial implementation is generated. In Phase ②, several (other) developers complement this partial implementation, extending the autonomous service, *i.e.,* the player, to provide the role's concrete performance. This preserves the independent development of the service separated from its role essential to a collaboration. Consequently, the resulting services developed following this methodology can be dynamically composed to Smart Service Systems on-demand.

2.2. Scenario

We consider an interactive, tech-enhanced classroom setting in which both the lecturer and the students have smart devices, *e.g.,* smart phones or tablets. The lecturer delivers the lecture using a smart device to present accompanying slides. The students' smart devices, by contrast, are only able to display the current slide and the annotations the lecturer adds to the slides. The students, additionally, are able to give feedback to the lecturer and also may ask questions by virtually raising their hand. The lecturer is the head of the classroom collaboration and keeps it alive, while students may join and leave during the collaboration.

Since our main focus is on structural composition and adaptation, we rely on a basic implementation of the scenario as this is sufficient in order to explain and discuss our approach hereinafter. Listing 1 shows the specification of the scenario using the RBCS. Thereof, a partial implementation (in Java) is generated. Listing 2 shows the generated implementation of the collaboration and comprises

```
module org.roledisco.samples.classroom
collaboration InteractiveClassroom {
  context LectureContext lectureContext
  coordinator role Lecturer {
    context LecturerContext lecturerContext
    player op receiveFeedback(String message)
    op setSlide(Slide slide) { // outbound method call
      slideBase64 = slide.toBase64()
      Student.setSlide(slideBase64) } }
  role Student {
    context LectureContext lectureContext
    context StudentContext studentContext
    op submitFeedback(String message) {
      Lecturer.receiveFeedback(message) }
    op setSlide(String slideBase64) {
      slide = Slide.fromBase64(slideBase64)
      player.setSlide(slide) } }
  multiplicities { Lecturer one-to-many Student }
  constraints { Lecturer >-< Student } }
```

Listing 1. Scenario's Collaboration Specification.

```
package org.rosi.roledisco.samples.classroom;
@Collaboration(
  coordinator = SimpleCoordinatorRole.class,
  roles = SimpleRole.class )
@Constraint(
  from = SimpleCoordinatorRole.class,
  to = SimpleRole.class,
  value = RoleConstraint.ROLE_PROHIBITION )
@Multiplicity(
  from = SimpleCoordinatorRole.class,
  to = SimpleRole.class, value = RoleLink.ONE_TO_MANY )
public class InteractiveClassroomCollaboration extends
    AbstractCollaboration {
  @Context LectureContext lectureContext;
  /* ... */ }
```

Listing 2. Generated Collaboration Implementation.

mostly structural information. Listing 3 shows the generated implementation of the non-coordinating *Student* role. The generated *Lecturer* role is similar to that of the *Student* and therefore has been omitted.

```
package org.rosi.roledisco.samples.classroom;
public class StudentRole extends AbstractRole {
  @Context LectureContext lectureContext;
  @Context StudentContext studentContext;
  public void submitFeedback(String feedback) {
    Dispatcher.getDispatcher().dispatchToRole(this,
        LecturerRole.class,"receiveFeedback",msg); }
  public void setSlide(String slideBase64){
    Slide slide = Slide.fromBase64(slideBase64);
    Dispatcher.getDispatcher().dispatchToPlayer(this,"
        setSlide",slide); }}
```

Listing 3. Generated Student Role Implementation.

3. Middleware Architecture

Figure 2 shows the high-level architecture of a subsystem equipped with the RoleDiSCo middleware[1], which is assumed to run on each subsystem (node) in the infrastructure. The *Application Container* contains the roles and the collaboration, as a result of Phase ① of the Two-Phase Development Methodology [4]; the player-specific parts provided by a developer of Phase ②, *i.e.,* the *Context Providers*, the *Players* or *Bindings*; as well as the independently developed service. *Collaboration Specifications* can be reconstructed from the

1. https://bitbucket.org/roledisco/roledisco

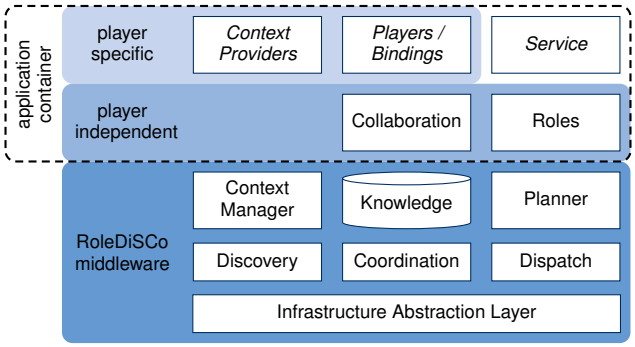

Figure 2. Architectural Overview of the RoleDiSCo Middleware.

collaboration and role classes. Since a collaboration is a self-contained bundle, an application container contains all role types even if it does not provide players for all of them.

The middleware consists of several modules with different responsibilities, discussed in detail in the remainder of this section. Figure 2 omits supportive modules: a message queue processes messages from remote subsystems and delegates them to their respective modules; an event system is used to notify components about changes, *e.g.,* updated discovery information, which allows loose coupling of the components.

3.1. Infrastructure Abstraction Layer

Subsystems communicate via the *Infrastructure Abstraction Layer*, which has been introduced as a replaceable component in order to completely abstract from concrete underlying protocols and infrastructures. We assume that every node in the infrastructure is reachable via this layer. The layer provides the following interface:

publish(message)
Broadcasts the message to *all* subsystems within the infrastructure without expecting a reply (non-blocking).

send(target[s], message)
The message is only sent to the specified target system(s).

dispatch(target[s], method name, parameters [, return type [, callback]])
Dispatches a method call to the specified target system(s). If a *return type* is specified, this method expects the remote method to provide a return value, implying a blocking method call. A *callback* may be passed, which is executed upon receiving the return value, and results in a non-blocking method call.

At this point, we assume reliable messaging as well as request/response handling in order to realize method calls with return values. The received messages are to be forwarded to the message queue, which distributes it internally to the corresponding modules. In order to send messages to specific subsystems, they require unique addresses which are generated within the middleware as the very same subsystem might be connected via multiple links. Thus, each subsystem generates a unique identifier [8] the first time it is started and keeps that during its lifetime. Our current implementation uses JGroups [9], which provides reliable, decentralized messaging as well as request/response handling.

3.2. Local Repositories & Knowledge

The middleware and thereby each subsystem has a *Knowledge* base to maintain knowledge for continuous operation. This comprises a repository for *Collaborations Specifications*, one for *Discovery Information*, one for *Dispatch Information*, the *Fills Relation*, and the *Context Cache*.

The *Collaboration Specifications* repository contains all locally available collaboration specifications, comprising the collaboration class, its role classes, and thereby also the relationships within the collaboration. The repository is managed by the *Discovery* module, as explained later, but specifications may also be added to and removed from the repository manually: Adding a collaboration specification to the repository requires the collaboration type, the coordinating role type and all non-coordinating role types. Removing a collaboration from the repository only requires the collaboration type. The *Fills Relation* complements this repository and contains potential players for role types. The »*fills relation*« usually refers to the modeling level and defines which player type plays which role type. This, however, contradicts the idea of serendipity. Thus, the *Fills Relation* is populated at run time, as a task of the *Discovery* module.

The *Discovery Information* repository contains all role types available in the infrastructure that have a corresponding player to perform that role, the respective subsystem the role type is located on, and its associated context information. This repository, however, only knows *that* the role type has a player on the respective subsystem, but neither its implementation nor class name. Moreover, the repository may contain multiple entries with the same subsystem and the same role type but different contexts, in case a role type might be applicable in several contexts. Context information, by default, includes context features that are part of both the collaboration and a non-coordinating role type. In the example given before, this would apply to the `LectureContext`. The repository is populated by the *Discovery* module, as explained in the subsequent section.

The *Dispatch Information* repository captures run-time information of a running pervasive collaboration. Once a pervasive collaboration is instantiated (*cf.* Section 4), all participating subsystems keep track of that collaboration and its role instances. This enables to dispatch method calls between roles. Although only the *Pervasive Collaboration Coordinator*, *i.e.,* the initiating subsystem, is in charge of maintaining this information, it is replicated to all subsystems participating in the collaboration. This allows to dispatch

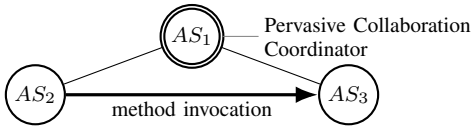

Figure 3. Direct Method Invocation, Bypassing the Coordinating Subsystem.

method calls bypassing the coordinating role's subsystem (*cf.* Figure 3) and circumvents bottlenecks. If a non-coordinating subsystem notices, for instance, the absence of a required subsystem, it is able to queue the messages until the coordinator orders that subsystem to do something different.

3.3. Discovery

The *Discovery* module is responsible for broadcasting and collecting discovery information within the infrastructure. Discovery information comprises the collaboration type and role type, both as fully qualified name derived from the collaboration specification, the subsystem those types are located on, and the respective context information. Only role types that have a complementing player implementation are broadcast. The discovery module is subdivided into local and remote discovery and therefore has to perform different tasks. Additionally, the discovery module monitors the infrastructure for subsystems which joined or left.

The *local role discovery* scans the subsystem for available collaboration and role types, which are then added to the collaboration specifications repository if they have not been added yet. Therefore, the partial implementation contains the `@Collaboration` annotation (*cf.* Listing 2:2–4), which allows to find collaboration types and to register them and their roles to the collaboration specifications repository. Subsequently, the collaboration specification can be reconstructed using the annotations shown in Listing 2 and Listing 3.

The *local player discovery* scans the subsystem for complementing player implementations for the locally available role types. We assume that the subsystem's runtime can be introspected and, especially in the case of a role-based runtime, that it is possible to request complementing player types for role types. First, the local role-based runtime is introspected and requested to provide the complementing player implementations. If that fails, the role-based runtime is asked whether it has a complementing player implementation for the role type. At this point, the middleware would only know that a player exists but it would not be aware of the number of implementations or their types. The first steps are skipped unless a role-based runtime is available. Second, configuration files containing role types and complementing player implementations are processed. Last, the (generic) runtime is searched for classes annotated with `@Player(<role type>)` [4]. Those classes' annotations also contain the role type the player class can play. Thereby, the *Fills Relation* is populated.

Knowing the local role types that have a corresponding player is a crucial prerequisite for *publishing locally available role types* as only those role types that have a player are to be published. Subsequently, the discovery information is broadcast whenever the infrastructure changes, the collaboration specifications repository changes, the context associated to the role type changes, a role type has no complementing player implementation anymore, or a complementing player for a previously not playable role type is available. For each role type to be published a *RoleAnnouncement* message containing the collaboration type, the role type, the unique identifier of the local subsystem, and the respective context features, is sent to all other subsystems. Listing 4 shows an example message for the scenario introduced earlier. The *remote role discovery* processes these messages and adds their content to the local discovery information repository if the discovered role type is available locally as well.

```
1 RoleAnnouncementMessage:
2   CollaborationType: org.r...o.samples.classroom.
        InteractiveClassroomCollaboration
3   RoleType: org.r...o.samples.classroom.StudentRole
4   Subsystem: e3858665-063d-4d2b-9a11-3b90848e90a8
5   Context: LectureContext{...}
```

Listing 4. Sample RoleAnnouncement Message.

Finally, the discovery module monitors the infrastructures for changes, *e.g.,* subsystems that left. For the sake of simplicity, we rely on mechanisms integrated in JGroups [9]. If necessary, an operating collaboration has to be adapted, causing changes to the dispatch information repository.

3.4. Context Manager

During discovery, the collaboration and its roles cannot provide context information on their own easily as they only exist as types and therefore could only provide static information. Conceptually, context should be provided by the player, which is, at first, the adaptive subsystem and therein a complementing class, core object, etc., which is provided by the runtime. Thus, the *Context Manager* is introduced to acquire context information defined in the RBCS. Therefore, Phase ② developers are asked to implement a *Context Provider*, providing the following method:

`getContext(role type [, feature])`
: This method has to return the concrete context values for the role type's features specified in the collaboration specification's context blocks, *e.g.,* an instance of `StudentContext` for feature `studentContext`. Unless a concrete feature is specified, context values for all features shall be returned.

The context manager obtains the context features to be requested from the providers using the `@Context` annotations of the respective fields inside the derived collaboration and role classes (*e.g.,* Listing 3:3–4). The context providers have to register themselves to the context manager. Providers can notify the context manager, whose interface is described below, about changes using the `update` method.

`update(provider, role type [, feature])`
: Registered context providers may push context changes. The context manager will retrieve the context values itself. This may also cause dependent modules, such as the discovery module, to perform some tasks.

`getContext(role type [, feature])`
: Returns the cached context values for a given feature and a specific player class or delegates the call to the corresponding context provider(s) if no values are cached. If the feature is omitted, a set of key-value pairs is returned. It is used by other modules, *e.g.,* discovery, as context providers are not accessed directly.

3.5. Dispatcher

The *Dispatcher* module resolves method calls to remote roles and to local roles' players. It integrates with the local runtime in order to retrieve players for role types, as described

for the *local player discovery* before. This is also a reason why the interface looks like a typical dispatch component of a role-based runtime, such as LyRT [10], which, for instance, provides a registry to look up such information.

`dispatchToPlayer(role instance, method name, parameters)`
Delegates a call from a remote role to the local player.

`dispatchToRole(calling role instance, target role type, method name, parameters[, callback])`
Dispatches a call from a local to a remote role. The calling role instance is used to determine the collaboration and, thus, the target role instances. The instances are required to determine the target addresses before the method is dispatched to the infrastructure abstraction layer. As method calls may be dispatched in a non-blocking way, an optional callback might be passed that expects the return value of the remote system as input.

`dispatchToRoles(calling role instance, target role type, method name, parameters[, callback])`
In contrast to the method before, this method dispatches a call from a local role instance to several remote role instances, which are determined as before. The callback will be invoked for every received return value.

`hasPlayer(role type[, context])`
Used within the discovery and dispatcher modules to determine whether a role type has a complementing player implementation. The default strategy works as explained within the *local player discovery*.

`getPlayer(role type[, context])`
The local runtime is asked to return an instance of a complementing player implementation for the given role type in an (optional) context. It is used for dispatching as well as binding and unbinding during composition.

`[un]bind(role instance, player instance)`
Binds or unbinds a role during composition. By default, the binding relation is stored in an internal memory. A role-based runtime might choose more sophisticated strategies.

`activate(role instance)`
Activates a role binding. After that, method calls are effectively dispatched to the player and to other roles. The default strategy is to annotate the respective entry of the relation mentioned afore with an *active* flag.

3.6. Planner

The *Planner* module calculates the composition plan of a collaboration, *i.e.,* on which subsystems which roles have to be instantiated and integrated into the collaboration. The planner module takes the collaboration specification and the associated discovery information from the respective repositories as input and returns a set of composition instructions, *i.e.,* the roles to be instantiated and their respective subsystems and contexts. As our focus is on composition and adaptation, we assume that such planners exist. For instance, OptaPlanner [11] is a lightweight, embeddable planning engine based on Java. Yet, we focused on basic support of different application structures, such as one-to-one and one-to-many with multiple target role types (*i.e.,* the coordinating role is connected to an/many instance/s of more than one other role type). We follow a greedy strategy, *i.e.,* we aim to incorporate as much roles as possible. Context matching is achieved as follows: The non-coordinating role's context features are matched against those of the collaboration. With respect to the scenario, a student's role's `lectureContext` must be equal to that of the collaboration. In general, only context features being of the same type and appearing in both a non-coordinating role and its surrounding collaboration are matched against each other.

4. Towards Coordinated Composition

Our goal was to bridge the gap between the design-time specification and the run-time execution, *i.e.,* to achieve automated discovery and composition, especially in decentralized environments. Thus, we will sketch how the proposed middleware contributes to tackle these challenges.

The derived *Partial Implementation* of the *Two-Phase Development Methodology* [4] is the prerequisite to bridge the gap between design and run time. The generated `@Collaboration` annotation (Listing 2:2–4), in conjunction with the constraints (Listing 2:7–8), enables the middleware to recreate the collaboration specification at run time. Consider a small scenario with 3 subsystems S_1 to S_3; S_1 has the *Lecturer* role type, S_2 and S_3 have the *Student* role type. All have complementing players, which were retrieved using the `@Player` annotation, as well as respective *Context Provider*s, which were registered to the *Context Management*.

S_1 starts and broadcasts a `RoleAnnouncementMessage`, comprising, i.a., the lecturer role type, the originating subsystem and the available context information. Once S_2 launches, S_1 will detect a change in the infrastructure and send the message to S_2 again. S_2 broadcasts a `RoleAnnouncementMessage` as well. The same applies to S_3.

Thus, discovery knowledge is established in a decentralized way, which is a prerequisite for composition. The composition process is triggered by instantiating the generated collaboration class (*cf.* Listing 2). The coordinating *Lecturer* role may have been instantiated prior to that point, thus, instantiating the collaboration; otherwise it will be instantiated with the collaboration and bound to its player.

Next, the *Planner* is provided with the collaboration, its context (ctx_C), and the obtained discovery information, *i.e.,* $\{(S_2, Student, ctx_2), (S_3, Student, ctx_3)\}$. It matches the contexts ctx_i against ctx_C and returns a set of composition instructions, *i.e.,* the roles to be instantiated, their respective subsystem and context. If ctx_3 does not match ctx_C, *e.g.,* if the student is interested in a different lecture than that the collaboration is used for, it could return $\{(S_2, Student, ctx_2)\}$.

Please note that both composition and adaptation are centralized on-demand and coordinated by the system the collaboration is triggered on, denoted as *Pervasive Collaboration Coordinator* [4]. The reliable exchange of messages allows to instruct other subsystems to instantiate roles and bind players. A protocol to achieve composition and

adaptation in a coordinated way is a remaining task. Before composition is completed, dispatch information, similar to composition instructions, is shared with the respective subsystems, enabling the *Dispatcher* to dispatch method calls between systems.

5. Discussion & Related Work

Advantages of the role concept have been discussed in the context of the *Two-Phase Development Methodology* [4]. At run time, we continue to benefit from the independence of a role and its player, as they can be bound and unbound, which results in varying behavior. It is, moreover, an advantage to distinguish between types and instances, as a playing entity thus can intuitively participate in several collaborations using multiple instances of the same role type.

Decentralization has been addressed two-fold: The homogenous middleware architecture does not rely on a predefined, static master node. All information that has to be shared is replicated and subsequently used solely locally. In order to avoid decentralized decision-making, *e.g.,* calculating the composition plan, which is a challenging task [12] and expected to be time-consuming, we centralize coordinated composition (and subsequent adaptation) *on-demand*, thereby avoiding a statically predefined central coordinator.

In contrast to SOSSs, such as GoPrime [3], we maintain a complex service structure within a pervasive collaboration as long as it needs to operate or until it fails because its specification's constraints are violated.

In *DEECo* [13], component communication and bindings are extracted from the implementation and implicitly specified in an ensemble specification, comparable to our RBCS. DEECo is limited to *coordinator-to-member* interactions and vice versa. It implies predefined processes and its components solely operate on local knowledge shared with other components by a common middleware layer. Recently, a notion of roles was added [14], however, they only act as a shared interface and have no behavior. Hence, there is no support for serendipity. Components are known in advance to the ensemble. DEECo defines context-based membership predicates, *i.e.,* functions that evaluate context features, in order to restrict the collaborators of the collaboration, which are more fine-grained than our basic context matching.

The *Helena Approach* [15] provides a formal foundation for modeling distributed systems by teaming up roles into ensembles. It is a component-based approach, which provides an ensemble specification similar to DEECo. An ensemble specification of Helena, however, does not result in a distributed system with automated discovery and composition [16]. Additionally, players in Helena have no intrinsic behavior, thus, exchanging a player does not result in a different performance of a role.

6. Conclusion

We presented the architectural details of RoleDiSCo, our middleware for coordinated on-demand composition of Smart Service Systems in decentralized environments. The RoleDiSCo middleware completes the overall RoleDiSCo idea (*cf.* Figure 1), in which a smart service system is specified as collaboration of roles. Existing services simply adopt these roles. The role's associated collaboration specification enables the proposed middleware to perform automated discovery, coordinated composition and subsequent adaptation, all by preserving the individual development lanes.

In the future, a generic planner solution needs to be developed. A protocol for coordinated composition and subsequent adaptation is a current work in progress.

Acknowledgments

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] Open Connectivity Foundation, Inc., "UPnP Device Architecture 2.0," Tech. Rep., 2015.

[2] G. Di Marzo Serugendo *et al.*, "Self-Organisation: Paradigms and Applications," in *Engineering Self-Organising Systems*, G. Di Marzo Serugendo *et al.*, Eds. Springer, 2004, pp. 1–19. DOI: 10.1007/978-3-540-24701-2_1.

[3] M. Caporuscio *et al.*, "GoPrime: A Fully Decentralized Middleware for Utility-Aware Service Assembly," *IEEE Trans. Softw. Eng.*, vol. 42, no. 2, pp. 136–152, 2016. DOI: 10.1109/TSE.2015.2476797.

[4] M. Wutzler *et al.*, "Utilizing Role-based Models for Distributed On-Demand Service Composition," in *Comp. Proc. of Programming'17*, ACM, 2017. DOI: 10.1145/3079368.3079390, in press.

[5] M. Wutzler *et al.*, "Role-Based Models for Building Adaptable Collaborative Smart Service Systems," in *2017 IEEE Intl. Conf. on Smart Computing (SMARTCOMP)*, IEEE, 2017, pp. 1–6. DOI: 10.1109/SMARTCOMP.2017.7947041.

[6] M. Wutzler, "Exploring On-Demand Composition of Pervasive Collaborations in Smart Computing Environments," in *OTM 2016 Workshops. OTMA 2016, Rhodes, Greece.* I. Ciuciu *et al.*, Eds. Springer, 2017, pp. 305–314. DOI: 10.1007/978-3-319-55961-2_31.

[7] G. Boella and F. Steimann, "Roles and Relationships," in *Proc. of the 2007 Conf. on Object-Oriented Technology*, Springer, 2007.

[8] P. Leach *et al.*, "A Universally Unique IDentifier (UUID) URN Namespace," IETF, RFC 4122, 2005, pp. 1–32.

[9] *JGroups*. [Online]. Available: http://www.jgroups.org.

[10] N. Taing *et al.*, "A Dynamic Instance Binding Mechanism Supporting Run-time Variability of Role-based Software Systems," in *Comp. Proc. of the 15th Intl. Conf. on Modularity*, ACM, 2016, pp. 137–142. DOI: 10.1145/2892664.2892687.

[11] *OptaPlanner*. [Online]. Available: https://www.optaplanner.org.

[12] D. Weyns *et al.*, "On Patterns for Decentralized Control in Self-Adaptive Systems," in *Softw. Eng. for Self-Adaptive Systems II*, R. de Lemos *et al.*, Eds., Springer Berlin Heidelberg, 2013, pp. 76–107.

[13] J. Keznikl *et al.*, "Towards Dependable Emergent Ensembles of Components: The DEECo Component Model," in *2012 Joint Conf. on Software Architecture and European Conf. on Software Architecture*, IEEE, 2012, pp. 249–252. DOI: 10.1109/WICSA-ECSA.212.39.

[14] T. Bureš *et al.*, "Towards Intelligent Ensembles," in *Proc. of the 2015 European Conf. on Software Architecture Workshops*, ACM, 2015. DOI: 10.1145/2797433.2797450.

[15] R. Hennicker and A. Klarl, "Foundations for Ensemble Modeling – The Helena Approach," in *Specification, Algebra, and Software*, Springer, 2014, pp. 359–381. DOI: 10.1007/978-3-642-54624-2_18.

[16] A. Klarl *et al.*, "From Helena Ensemble Specifications to Executable Code," in *Formal Aspects of Component Software*, Springer, 2014, pp. 183–190. DOI: 10.1007/978-3-319-15317-9_11.