Tobias Jäkel, Martin Weißbach, Kai Herrmann, Hannes Voigt, Max Leuthäuser

Position paper: Runtime Model for Role-based Software Systems

Erstveröffentlichung in / First published in:

International Conference on Autonomic Computing (ICAC). Würzburg, 17. – 22.07.2016. IEEE Xplore, S. 380 – 387. ISBN 978-1-5090-1654-9.

DOI: https://doi.org/10.1109/ICAC.2016.17

Final edited form was published in "International Conference on Autonomic Computing (ICAC). Würzburg 2016", S. 380 - 387. ISBN: 978-1-5090-1654-9
https://doi.org/10.1109/ICAC.2016.17

Position paper: Runtime Model for Role-based Software Systems

Tobias Jäkel*, Martin Weißbach†, Kai Herrmann*, Hannes Voigt*, and Max Leuthäuser‡
*Database Systems Group
†Computer Networks Group
‡Software Technology Group
Technische Universität Dresden, D-01062 Dresden
E-Mail*‡: firstname.lastname@tu-dresden.de
E-Mail†: martin.weissbach1@tu-dresden.de

***Abstract*—In the increasingly dynamic realities of today's software systems, it is no longer feasible to always expect human developers to react to changing environments and changing conditions immediately. Instead, software systems need to be self-aware and autonomously adapt their behavior according to their experiences gathered from their environment. Current research provides role-based modeling as a promising approach to handle the adaptivity and self-awareness within a software system. There are established role-based systems e.g., for application development, persistence, and so on. However, these are isolated approaches using the role-based model on their specific layer and mapping to existing non-role-based layers. We present a global runtime model covering the whole stack of a software system to maintain a global view of the current system state and model the interdependencies between the layers. This facilitates building holistic role-based software systems using the role concept on every single layer to exploit its full potential, particularly adaptivity and self-awareness.**

I. Introduction

Software systems find themselves in increasingly dynamic environments. This is not a new trend, so software developers found agile and efficient ways to continuously implement new features, fix bugs, or react to changing requirements while maintaining a high software quality [4]. All these changes keep the developer in the loop. However, in today's scenarios software systems need to adapt to changing environments frequently and immediately. Keeping a human developer in the loop is practically not feasible. Hence, software systems need to be self-aware and adapt to their environment autonomously. For instance a mobile phone should be able to recognize if the user is in a meeting and mute the alarm.

Software systems are usually composed of multiple heterogeneous components, which is hard to control and handle for a human. A self-aware software component recognizes itself within the environment of other components of the software system to behave and interact accordingly [15]. This adaptation of the component's state and behavior is the central challenge to achieve the expected dynamics. Though, modeling this adaptivity hits the limits of current object-oriented solutions (especially static objects), which require explicit implementation of the adaptivity rendering the software system more complex, hard to maintain, and less robust.

Current research develops the concept of *role-based* [2] software to intuitively handle the dynamics, particularly the self-awareness and adaptivity of software components [12]. Roles allow to easily adapt the behavior and states of software components at runtime. For instance a person at a university plays the student role, which adapts the state and behavior of the person accordingly. In another context, e.g., a soccer team, the same person could play the goalkeeper role resulting in completely different states and behavior. Role-based software establishes such dynamics as primary high-level concepts, so we can easily model and implement them.

Role-based software is an established concept on the application level. Current approaches map the role-based application to existing runtime concepts, e.g., Object Teams[1] is based on Java and SCROLL is based on Scala [14]. So, already on the runtime level, we loose the role-based abstraction, even though there is promising research establishing the role concept on the other layers of the software stack as well. For instance, RSQL provides a role-based database management system [11], Role Relational Mapping provides a role-based persistence layer [5], and PROtEUS provides role-based process management [18], and so on. These are isolated approaches on the single levels of the software stack.

The runtime models of the isolated layers are well defined and understood, however, a global runtime model and the interactions between these layers are not defined so far. In particular, each layer handles the states of a role individually, which provides consistent role states for the specific layer, but from a global software system perspective the combination of those states may be invalid. For example, a role is registered in the persistence layer (valid individual state) but not represented in the database (valid individual state). Globally seen, this is an invalid state, because each role that is registered in the persistence layer needs to be represented in the database as well. Additionally, a transition of a role state on one layer may be not represented adequately on all others resulting in an invalid global state of that particular role. This problem is getting worse the more layers are involved in the system or the more states can be represented on the layers.

[1] http://www.objectteams.org

Final edited form was published in "International Conference on Autonomic Computing (ICAC). Würzburg 2016", S. 380 - 387. ISBN: 978-1-5090-1654-9
https://doi.org/10.1109/ICAC.2016.17

To overcome the problem of global invalid states, we create a global runtime model to describe and track the state of roles over the software stack to ensure a correct interaction between the different layers. Hence, the behavior and states intended at one layer are adequately represented at the other layer as well. Our holistic role-based runtime model realizes the global book keeping, so all layers of the software system are based on one coherent knowledge. Precisely knowing which roles are played by whom is essential to correctly adapt the state and the behavior at any layer.

The contributions of this work are: As prerequisite we firstly identify various local role states on several layers of a software system. Secondly, we introduce a global crosscutting role sate and thirdly define a management component conceptually, which ensures a valid role state and state transitions on each layer as well as on a global software system perspective.

The remainder is structured as follows: We detail the role concept in Section II and describe our view on the layers of a software system in Section III. Section IV identifies possible local role states on each layer. These are the prerequisites to define the states of our global runtime model and the respective transitions in Section V. We discuss implementation techniques, the architecture and metamodel variants in Section VI. Finally, Section VII concludes the paper.

II. The Role Concept

The role concept is based on the idea of separations of concerns such that the core will be separated by its fluent and context-dependent parts. However, roles and the underlying primitives, like dynamic role binding at run time, can be utilized to describe and implement context-dependent behavior and structure. Thus, role-based modeling and programming enables self-aware adaption at runtime, too.

Originally, the role concept was introduced in the 1970s by Bachman [2]. Over the last decades researchers have proposed several role-modeling approaches. Surveys showed 26 features associated to roles in general. Initially, Steimann identified 15 features for mostly relational roles [19]. On top of that, Kühn et al. proposed 11 additional features to capture the context-dependent nature of roles [12]. However, the term role often causes confusion, because there is a different notion of roles in each domain (like data modeling or software engineering). In this paper we assume roles as objectified roles encapsulating context-dependent behavior and structure. Additionally, roles are used to extend a core object's behavior and structure at runtime. Thus, a player (the core object) is able to dynamically start or stop playing roles resulting in different behavior and structure of the same (core) object during runtime. For example, in traditional object-oriented languages, the behavior and structure of an object is statically defined by its class, i.e., each situation an object can be in has to be modeled in advance. In contrast, in role-based approaches, the context-dependent behavior and structure is moved from the core objects to roles. For instance, imagine a person that becomes a student at a university, thus, the core object of that particular person is extended by a new role of type student enabling new structure (i.e., a student ID) and new behavior (i.e., goToLecture()) of that specific person. In particular, we assume the Compartment Role Object Model (CROM) as metamodel for our notion of roles, which is briefly introduced below [13].

TABLE I
Ontological Foundation

Concept	*rigid*	*founded*	*identity*
Natural Types	yes	no	unique
Role Types	no	yes	derived
Compartment Types	yes	yes	unique
Relationship Types	yes	yes	composed

Generally, CROM distinguishes between four elemental meta types: (i) Natural Type, (ii) Compartment Type, (iii) Role Type, and (iv) Relationship Type. Ontologically, these types can be distinguished by the meta properties *rigidity*, *foundedness*, and *identity*. The first one specifies an instance's need to be part of this type for its entire lifetime, whereas the second one denotes the required existence of other types. The last one distinguishes between whether a type's identity is unique, composed or derived [13]. A summary of these types and their corresponding specification is shown in Table I. Natural Types build a core object that does not depend on any other types. Thus, it is rigid, non-founded and has a unique identity. For instance, a *person* is such a Natural Type, since persons can exist without any other types, each person is uniquely identifiable and an instance will always be part of this type. The opposite of Natural Types are Role Types, which are anti-rigid, founded and have a derived identity. A *student*, for instance, is a Role Type that depends on a player, in our case the person, and the compartment *university* as context. A Compartment Type builds a new core object, but also depends on participating Role Types. Thus, it is rigid and founded. Moreover, a Compartment Type can be seen as objectified context that has participating Role Types in it, but can also fill Role Types on itself. Additionally, this type has a unique identity, because core elements have to be uniquely identifiable. As mentioned before, a university is an example for a Compartment Type. The last meta type is the Relationship Type, that is applied between two Role Types only, hence, it is rigid, founded and the identity is composed by the players' identity of the participating Role Types. For example, imagine the Role Types testee and tester that are related by the Relationship Type exam.

Additionally, CROM requires several constraints on the relations between the aforementioned meta types. At first, each Role Type needs to be connected to a player, that can be a Natural Type or Compartment Type. Furthermore, a Role Type needs to participate in exactly one Compartment. Additionally, empty Compartments are prohibited, i.e., each Compartment Type requires at least one Role Type to participate. Next, Relationship Types are context-dependent, hence, participating Role Types need to be of the same Compartment Type.

Final edited form was published in "International Conference on Autonomic Computing (ICAC). Würzburg 2016", S. 380 - 387. ISBN: 978-1-5090-1654-9
https://doi.org/10.1109/ICAC.2016.17

These definitions and distinction of meta types allows a role-based context-dependent representation of dynamic entities at runtime. Nevertheless, the life-cycle of Natural Instances and Compartments is straight forward and comparable to objects. Thus, the rest of this paper is going to focus on roles[2] only.

III. Software System Layers

A software system, in general, describes various compound software components to fulfill a dedicated task. Those tasks can range from graphical user interface on the front-end side over application server to low level components like an operating system. However, for persistent software systems we can identify four layers performing special subtasks: (i) *Application Layer*, (ii) *Runtime Environment Layer*, (iii) *Persistence Layer*, and (iv) *Database Layer*.

Application Layer

The Application Layer describes all software components that are directly related to application software. This includes any sort of user interfaces, communication services, and back-end solutions. Mostly, the business logic is embedded in the Application Layer. Additionally, this layer is often considered as software from a user and programmer perspective, neglecting the fact that other layers enable this separation[3]. For example, imagine a java-written application managing exams at a university that provides a sign up process for exams while checking exam preliminaries at the same time. Concerning our proposed 4-layered architecture, the Application Layer is considered top-most, since it is visible to the end-user.

Runtime Environment Layer

The second considered layer integrates all components necessary to compile and run any kind of application. It takes care of producing executable code out of high level programming languages, memory management, and instantiation and deletion of objects. Thus, no application will be executed without services out of this layer. For instance, consider the Java Software Development Kit (JDK) including the Java Runtime Environment (JRE). Image a student who logs in to the aforementioned exams management application and registers for an exam, thus, a new testee role will be created. First, the memory is allocated and second the new instance is create by the runtime environment. Finally, the newly created role instance is bound to the person who registered and to the university compartment. Since the Application Layer requires mechanisms provided by the runtime environment, e.g., for creating and binding role instances, the Runtime Environment Layer sits below the Application Layer.

Persistence Layer

The next layer integrates all components of a software systems that provide persistence services to applications. Those are desired from an application's perspective to store objects beyond application's runtime. Additionally, the durable storage is beneficial in case of application errors or breakdowns. Since the data model of applications and databases usually differ, this layer often provides mapping services to transform runtime objects into database objects and vice versa. In case of object-oriented applications and relational database management systems this issue is known as object-relational impedance mismatch. Along with the mentioned tasks, the persistence layer also keeps track of loaded and stored objects to decide whether an object needs to be inserted or updated in the database. Exemplarily, the Java Persistence API (JPA) specifications can be seen as representative for such a layer. Consider a student signs up for an exam and that information needs to be persisted, the Persistence Layer transforms the object into relations and creates proper statements. From a software stack perspective this layer comes third seen from above, since it is not necessary to implement and communicates with the application layer and runtime environment layer.

Database Layer

Finally, we consider a Database Layer that provides at least durability and consistency characteristics. The former is important to guarantee object storage beyond runtime whereas the latter one ensures proper and application schema conform storage. Isolation and atomicity properties may not be required, especially in case a NoSQL database management system is deployed in a software system. Additionally, this layer takes care of efficient data access and storage. Imagine the student signed up for an exam and the persistence layer called a statement, that database layer inserts this information into the data storage while logging this action at the same time. In our 4-layer software stack perspective, this layer is the bottom one, since all others are built on top.

IV. Local Role States

Applying the role concept on each of these layers causes different states a role may have on each layer. In this section we elaborate the possible states on each layer. Thus, we firstly introduce those states in general and secondly apply them onto the different layers.

A. States of Roles

Within the role life-cycle roles appear in different states having different characteristics. In general, we distinguish between the following main states: *not existent* (nex), *unbound* (ub), and *bound* (b),

On the one hand, the not existent state describes the situation in which a role is unknown to a certain layer. On the other hand, unbound and bound describe states of roles if they are known to a specific layer. In the *unbound* case, the role is neither bound to a player nor a compartment, which results in the two sub-states *unbound player* and *unbound compartment*. The former is applied if the role is bound to a compartment but not to a player, hence, the latter one describes the situation that a role is not bound to compartment but to a player. In contrast to the unbound state, the bound state describes the situation a role is bound to a player and compartment. This

[2]Within this paper the term role refers to an instance of a role type.

[3]In this paper the term application refers to software related to this layer.

Final edited form was published in "International Conference on Autonomic Computing (ICAC). Würzburg 2016", S. 380 - 387. ISBN: 978-1-5090-1654-9
https://doi.org/10.1109/ICAC.2016.17

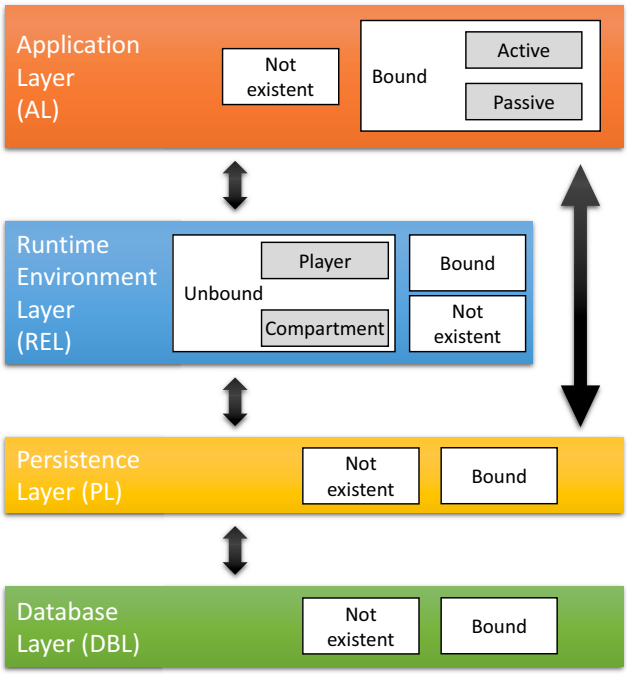

Fig. 1. Local Role States in the Software Stack

state can be separated into two sub-states, too. The first one is *bound passive* describing a role is currently unused. The second one, *bound active*, defines the opposite situation, when a role is currently used and methods are actively performed on it. Figure 1 illustrates the particular role states for each layer. Additionally, the bidirectional arrows indicate a communication interface between corresponding the layers.

B. Applying Role States to the Layers

Runtime Environment Layer: As mentioned in Section III the Runtime Environment Layer instantiates and destroys objects. In case of roles, this layer additionally binds the role to a player and adds it to a compartment. During this instantiation process, a role appears as *unbound (ub)*, *unbound player (ubp)* or *unbound compartment (ubc)*. The same states may appear during the destruction process, where roles must be unbound from their players and compartments before deleting them. Finally, if the role is related to a player and compartment, it switches to the *bound* state. Only the bound state is consistent to the role metamodel we assume, thus, only this state is exposed to other layers. Referring to our student example, in response to the enrollment of a new student at a university, the system would instantiate a new student role, bind it to the respective instance of the Natural Type *person* within the university compartment. After the first step, the new student role is in *unbound* state. Afterwards, the person is connected to this particular role, hence, the state switches to *unbound compartment*. Finally, the student role is associated to the university resulting in the *bound (b)* state.

Application Layer: The Application Layer distinguishes between *not existent (nex)* and *bound (b)* roles, in which it may be in an active or passive state. Again, *not existent* refers to an unknown role from an application perspective. Nevertheless, the role might be present in other layers. If a role is known to an application, it is in state *bound passive (bp)* or *bound active (ba)*. The former is applied in case no action is performed on the role, contrarily the latter one is applied if an action is performed on this particular role. For instance, the student role is present and signs up for an exam. While signing up, the role is in *bound active* and switches back to *bound passive* if this sign up process is completed and no other actions are running on this role.

Persistence Layer: In general, this layer communicates with the Runtime Environment and Application Layer. Usually, an application triggers a storage or loading operation to the Persistence Layer by providing role references. These references are used to retrieve the corresponding roles from the runtime environment. Afterwards, the runtime roles are transformed into the database data model and query language statements are generated. The Persistence Layer distinguishes between *not existent (nex)* and *bound (b)* states only. As mentioned before, unbound states of the Runtime Environment Layer are not exposed to any other layer, hence, they cannot be represented in this layer. Additionally, the differentiation between *bound active* and *bound passive* roles is not required, because these states only depend on application internal method calls. Furthermore, unfinished method calls can be rerun in case of application failures and crashes. Thus, all roles will be recovered in bound passive state and methods recalls set them to bound active automatically. Imagine the exam sign up application crashes while the student is currently signing up for an exam. After recovery, the student role is in bound passive state. If the student signs up again, the role will be set to bound active automatically, until this process has finished.

Database Layer: The Database Layer communicates with the Persistence Layer only, hence, it must represent any additional states. Consequently, the Database Layer distinguishes between *not existent (nex)*, if the role is unknown and *bound (b)*, if the role is known. Sub-states like unbound are conceivable in a database system, especially if the storage process requires multiple statements to create and connect the role. Nevertheless, there exist query languages, like RSQL, that capture special role semantics and enable role creation and binding within a single statement [11]. *Not existing* refers to an unknown role from the database perspective, hence, it is a transient role that is only known to the Application Layer and Runtime Environment Layer. In contrast, the bound state describes a known role to the Database Layer. Imagine, a person enrolls at the university and thus, gets a student role attached. In case this information needs to be persistent, the Database Layer directly reflects this information by processing the statements generated by the Persistence Layer accordingly.

Final edited form was published in "International Conference on Autonomic Computing (ICAC). Würzburg 2016", S. 380 - 387. ISBN: 978-1-5090-1654-9
https://doi.org/10.1109/ICAC.2016.17

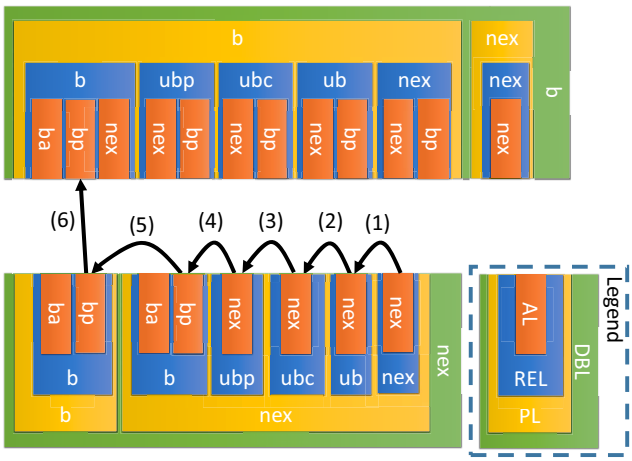

Fig. 2. Global Role State Changes for Adding a Role to a Player

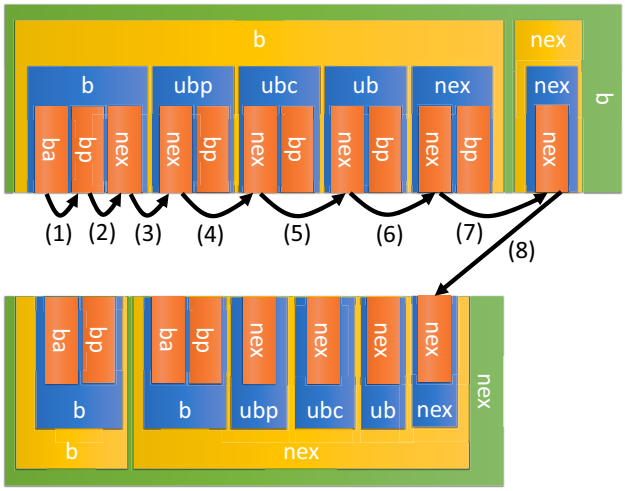

Fig. 3. Global Role State Changes for Removing a Role from a Player

V. Global Role States

The previously discussed life cycle states of roles in the respective software system layers do not exist in isolation, but relate to each other in order to ensure a consistent behavior of the overall software system. Therefore, we integrated the separately developed life cycle models of each system layer into a global model that precisely describes the software system's role-based adaptation behavior. As a result, we get a set of 4-tuples ($\{\{ba, bp, nex\} \times \{b, ubc, ubp, ub, nex\} \times \{b, nex\} \times \{b, nex\}\}$) describing the state of a role on each respective layer, e.g., the tuple $\{ba, b, nex, nex\}$ describes a role that is actively played by a player on the Application Layer (ba), is hence bound (b) on the Runtime Environment Layer and not persisted yet, thus, not existent (nex) on the Persistence and Database Layer. Evidently, not all tuples generated through mere combination of all possible states are valid, e.g., a role cannot be actively played in the application while it is unbound in the runtime environment. Figure 2 depicts the set of global states after elimination of all invalid combinations.

In the remainder of the section we will outline three concrete scenarios where the combined model prescribes life cycle transitions of roles during the run time of the application. First we address the addition of a new role to a player and its subsequent storage into the system's database. Secondly, we will discuss the removal of a role from its bound player including the deletion of the role from the Database Layer. In general, the removal of a role can be considered the inverse operation of the previously described add operation. Lastly, we will consider the restoration of a role from the database.

Imagine the basic scenario in which a player is supposed to start playing a new role. First of all, only the Application Layer and the Runtime Environment Layer are concerned with the procedure. The Runtime Environment Layer creates a new instance of the role, which is now unbound on this layer and not existent on the other layers, and binds it subsequently to the player and a compartment that determines the context in which the role is active. If the role is furthermore supposed to be persistent, it receives a save command eventually that is handled by the Runtime Environment Layer. The role is written to the Persistence Layer that writes it through to the Database Layer. Consequently, the role is now in a bound state on each layer of the software system. The entire process is displayed in Figure 2. Please note that the information whether a role is actively played, i.e., whether its compartment is active or not, is an information that is transparent for all layers below the Application Layer since it is only required for the internal method dispatch within the role-based application.

Similarly, the removal of a role from a player can be described as the sequence of global role states indicated by the numbered arcs: First, the role is in a bound state on each layer – we assume the role to be persistent, too, otherwise the process would stop after the removal of the role from the Runtime Environment Layer – and has to be transitioned to a bound and passive state on the Application Layer. This step is necessary in order to make sure the role finished all internal processes and stopped exchanging messages with other roles and players. If the role was still active, the loss of data or undesired system behavior would be the consequence. After the role was passivated, it will be unbound from its player and compartment entering the *not existent* state on the Application Layer. As soon as the role is in the *unbound* state on the Runtime Environment Layer, the role is removed from the Persistence and Database Layer using a delete command on the Persistence Layer. The sequence of visited global role states for the removal is shown in Figure 3.

As a last example we discuss the restoration of roles from the database. Assume the reference to a player has been nulled, because the instance was no longer needed, which would result in a role that is in the *bound* state on the Database Layer, potentially on the Persistence Layer, but nowhere else. Figure 4 displays the state changes of this scenario.

When a player is restored from the database, the required role information is retrieved as well. The initialization process

Final edited form was published in "International Conference on Autonomic Computing (ICAC). Würzburg 2016", S. 380 - 387. ISBN: 978-1-5090-1654-9
https://doi.org/10.1109/ICAC.2016.17

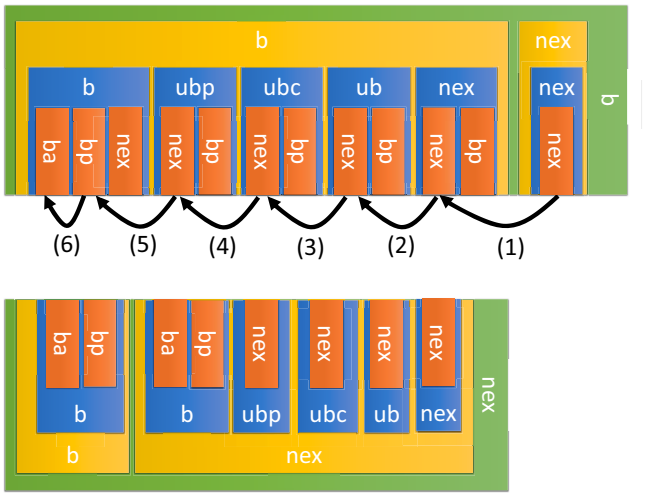

Fig. 4. Global Role State Changes for Loading a Role from the Database

of the role differs in this case from the first discussed scenario in a way that the Persistence Layer is asked to retrieve the information from the database. Consequently, the role joins the Persistence Layer in the *bound* state as soon as it has been loaded. The role instance is forwarded to the Runtime Environment layer where the role enters the *unbound* state immediately. Subsequently, the role is attached to its destined player and compartment, thus, moving to the *bound* state in the Runtime Environment Layer. Through the binding, the role is furthermore visible as *bound passive* on the Application Layer. The role's compartment's activity status decides if the role directly transitions into an active state or remains passive for the time being.

VI. Discussion

Within this section we provide discussions on available solutions and possible conceptual extensions. First, we will focus the discussion on technical solutions currently available and their limitations for each layer. Second, our architecture model and possible additional layers is discussed. Finally, we elaborate the impact of variation in the metamodel on the concerned layers.

A. Technical Solutions

This section gives an overview on how to technically implement the aforementioned role states throughout the different layers. The research field is highly fragmented and suffers from continuous reinventions of the same concepts over and over again [12], which leads to a missing common understanding of roles and, ultimately, no common model applicable the various approaches. Nevertheless, one can find certain commonalities in the basic concepts among the competitive approaches to achieve the implementation of our role states.

On the *Application Layer*, most of the languages are translations to Java (e.g., Chameleon [6], Rava [7], powerJava [1] or JavaStage [3]) or JVM-Bytecode (e.g., OT/J [8]), but without any actual representation of role (-binding) states. Sadly speaking, often there is no compiler available for them anymore or they are abandoned projects [20]. Others are library approaches and actually usable. For a detailed comparison please refer to [12]. Except the two library approaches (ScalaRoles [17] and SCROLL [14]) none of them allows to have unbound and/or compartment-less roles at any time. Because ScalaRoles and SCROLL use simple classes, case classes, and traits for implementing roles with Scala[4] as host language, they can be instantiated at any given point in time, regardless whether or not there is a player or compartment object available to bind them to. This violates the constraints set up in CROM but gives more flexibility, e.g., when dealing with legacy code or recovering from a failed system state.

To our knowledge, no role-aware runtime environment for the *Runtime Environment Layer* is available yet. Certain features for dynamic objects from VMs are promising (like InvokeDynamic from the JVM [16]) for implementing roles at this layer. Furthermore, with modular compiler systems like LLVM[5], one may add the required functionality without too much effort. For future work, it is necessary to investigate if the given features of those systems are sufficient to implement roles and their states on top of them or if there are certain requirements that impose new functionality at this layer.

From a *Persistence Layer* perspective there have been only few investigations regarding the persistence of role-based runtime objects. A Role Relational Mapping approach is presented in [5] enabling role-based programming languages like OT/J to automatically persist runtime objects in a relational DBMS. Within this approach, the role semantics will be discarded during the mapping process onto relational tables. However, the local states identified for this layer (not existent and bound) can be applied to the Role Relational Mapping approach. The DOOR concept presented in [21] combines the Persistence and Database Layer within a single one. It has been designed as an object database featuring roles for persistence purposes. Unfortunately, there is neither a query language nor result representation available, hence, objects can be stored and retrieved only without capabilities of external views on the data stored in a DOOR-DBMS.

Database systems are often neglected as integral part of software systems, which is feasible in a single-application scenario, but for multi-application scenarios the Database Layers acts as single point of truth for various applications. Thus, role semantics need to be represented in this layer, too. As mentioned previously, DOOR is a database system featuring roles, but lacks a well-defined external interface. Another approach is RSQL [10], [9], [11], a role-based contextual database system based on the CROM metamodel. RSQL provides role semantics including the notion of compartments for both the query language and the result representation. Thus, role-based consistency is ensured by RSQL and the identified local role states are natively supported, which directly facilitates

[4]See: http://www.scala-lang.org/
[5]See: http://llvm.org/

Final edited form was published in "International Conference on Autonomic Computing (ICAC). Würzburg 2016", S. 380 - 387. ISBN: 978-1-5090-1654-9
https://doi.org/10.1109/ICAC.2016.17

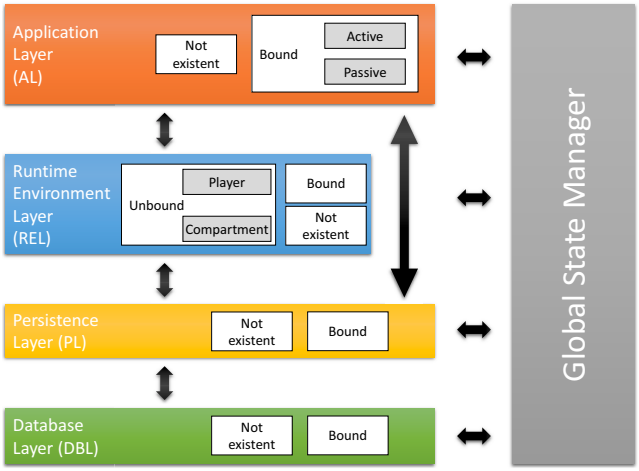

Fig. 5. Global State Manager

RSQL to be used with our global state manager. Consequently, mapping roles and compartments in the Persistence Layer becomes obsolete.

B. Architecture Model

Multilayered and n-tier architectures are a common principle to separate different concerns and to increase reusability and maintainability of the system. The 3-tier architecture consisting of (i) a *presentation tier*, which is the top-most tier of the application and usually provides the user with an interface through which the application can be accessed, (ii) a *logic tier* that provides the business logic of the application, processes commands or makes logical decisions and (iii) a *data tier* that stores and retrieves non-volatile data of the application is a typical representative of such architectures. If the second tier is tiered itself, the resulting architecture is called an n-tier architecture.

Although no dedicated role runtime is existing at the moment, we decided to introduce the *Runtime Environment Layer* as distinct tier in the application. A great majority of the previously discussed technical solutions that introduce roles to the Application Layer can be considered a *framework* solution, i.e., albeit providing abstractions for application developers typical tasks of runtimes, e.g., method resolution and dispatch, are provided, too. Consequently, it is safe to state that these solutions are bridging the role concept to already existing runtime environments, e.g., the Java Virtual Machine (JVM). Applications often intend to store data and persist runtime objects. Role-based applications are no exception, hence, the Persistence and Database Layer are also present in our discussed architectural approach. Especially the idea to follow the role concept through the entire application stack to investigate the implications on each respective layer was a main driver to incorporate these two layers.

As we have seen, roles exist in several layers of the application sharing a common identity to relate instances of a role through all layers of the application. A single role instance, however, has no knowledge of its state within this layer and is entirely oblivious to its representation on other layers. Consequently, a role cannot maintain its global state in the system itself. The global state of a role is a rather cross-cutting concern, cf. Figure 5, where each layer can potentially influence the role's state independently of all other layers that could potentially result in invalid configurations. To tackle the issue, we introduced the *Global State Manager* to the architecture of the role-based system, which is responsible for maintaining the global state of a role. Complex and cross-cutting tasks, e.g., restore a role from database, are supervised by the Global State Manager to ensure that (1) no invalid global state is reached and (2) only valid transitions of global role states are conducted within and across single layers.

C. Metamodel Variations

The Compartment Role Object Model (CROM) we assume as metamodel for the layers has strict constraints for valid instances [13]. Thus, no layer, except for the Runtime Environment Layer, allows for an unbound state, otherwise, a consistent and coherent role life cycle along the layers is not possible. Consequently, relaxing the metamodel in some constraints would enable more role states that help to handle and describe special situations more precisely. For instance, imagine a *president* role in a club when the person suddenly dies. In CROM the *president* role has to die as well, but for legal reasons there must be a president at any point in time. Thus, modelers and software engineers need to introduce workarounds when using the current CROM specifications. But, allowing an unbound state as valid situation in the metamodel would raise consequences for all layers. On the one hand, the Application Layer needs an additional register and bookkeeping for unbound roles. On the other hand, extra runtime methods are required to dispatch method calls on bound roles only.

In terms of the Database Layer a constraint relaxation would impact the query language as well as the result representation. In our scenario users and the Persistence Layer have to be able to query for unbound roles in particular. Adding unbound roles extends the query language by an extra dimension, resulting in more complex queries. In current role-based query languages, like RSQL [11], there is only the bound state, whereas roles in unbound state would need to be marked separately. Furthermore, result representation would become more complex, because unbound roles can neither be accessed by a player nor by a compartment.

A second variation of the metamodel is allowing deep roles, i.e., a role can play other roles. For example, a person who is a student at a university becomes a student assistant, too. The student assistant role obviously depends on the student role, hence, the person plays a student role and this particular role plays the student assistant role. This variance would lead to marginal changes on the Application Layer, Persistence Layer, and Database Layer, where only the relation between a player and the played role needs to be extended. The Runtime Environment Layer would need to additionally ensure a rigid

Final edited form was published in "International Conference on Autonomic Computing (ICAC). Würzburg 2016", S. 380 - 387. ISBN: 978-1-5090-1654-9
https://doi.org/10.1109/ICAC.2016.17

type (see Section II) in the role object's transitive closure. However, the states of a role remain the same on each layer and in the Global State Manager.

In sum, metamodel variations may affect all corresponding layers of the architecture independent of the particular variance. As shown in the first variation, small changes like an unbound state on the Application Layer lead to dramatic changes on the layers underneath. However, to enable a coherent global role state model the Application Layer, Persistence Layer, and Database Layer always have to share a common role notion.

VII. Conclusions

Software developers use the role-based software development to implement adaptive and flexible applications. However, the role-based concept is no longer limited to the application layer. We see upcoming role-based systems at many levels of the software stack, like runtime environments and databases. Each system itself works on one particular layer of the software stack and maps to existing technologies at the borders of its specific layer.

So far, the role-based layers of a software system were considered in isolation, each maintaining its own states for runtime objects. The valuable role-based abstraction is thereby lost immediately. This poses several problems like inconsistent representations on each layer, when the system needs to be self-aware and adapt to new situations and environments autonomously. In this paper, we proposed a runtime model for a Global State Manager component that ensures a coherent object representation along the software stack. At first, we identified several local role states on each considered layer. This foundation has been utilized to describe valid global states, which are a subset of all local role state combinations. Finally, we discussed currently available technical solutions for each layer and variations from a software system's metamodel and architecture perspective.

The argumentation throughout this paper is focused on the role concept, however, our global crosscutting state management approach is applicable to various domains and modeling paradigms. For example, aspect-oriented, context-oriented, and component-based software systems would benefit from our approach by maintaining the specific states globally.

In sum, the proposed runtime model in combination with the Global State Manager helps to realize adaptivity and self-awareness for role-based software systems in general. Developers can holistically model the system, as the role-based concepts are established at all layers and interactions are clearly defined. This greatly increases robustness as well as maintainability and paves the way for role-based software systems.

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] E. Arnaudo, M. Baldoni, G. Boella, V. Genovese, and R. Grenna. An implementation of roles as affordances: powerJava, Aug. 31 2009.

[2] C. W. Bachman. The programmer as navigator. *Commun. ACM*, 16(11):635–658, Nov. 1973.

[3] F. S. Barbosa and A. Aguiar. Modeling and programming with roles: introducing javastage. Technical report, Instituto Politécnico de Castelo Branco, 2012.

[4] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. Manifesto for Agile Software Development, 2001.

[5] S. Götz, S. Richly, and U. Aßmann. Role-based object-relational coevolution. In *Proceedings of 8th Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE 2011)*, 2011.

[6] K. B. Graversen and K. Østerbye. Implementation of a role language for object-specific dynamic separation of concerns. In *AOSD03 Workshop on Software-engineering Properties of Languages for Aspect Technologies*, 2003.

[7] C. He, Z. Nie, B. Li, L. Cao, and K. He. Rava: Designing a java extension with dynamic object roles. In *Engineering of Computer Based Systems, 2006. ECBS 2006. 13th Annual IEEE International Symposium and Workshop on*, pages 7–pp. IEEE, 2006.

[8] S. Herrmann. A Precise Model for Contextual Roles: The Programming Language ObjectTeams/Java. *Applied Ontology*, 2(2):181–207, 2007.

[9] T. Jäkel, T. Kühn, S. Hinkel, H. Voigt, and W. Lehner. Relationships for Dynamic Data Types in RSQL. In *Datenbanksysteme für Business, Technologie und Web (BTW)*, 2015.

[10] T. Jäkel, T. Kühn, H. Voigt, and W. Lehner. Towards a Role-based Contextual Database. https://wwwdb.inf.tu-dresden.de/misc/rsql/paper/rsql_crom.pdf.

[11] T. Jäkel, T. Kühn, H. Voigt, and W. Lehner. RSQL - A Query Language for Dynamic Data Types. In *Proceedings of the 18th International Database Engineering & Applications Symposium*, pages 185–194, 2014.

[12] T. Kühn, M. Leuthäuser, S. Götz, C. Seidl, and U. Aßmann. A Metamodel Family for Role-based Modeling and Programming Languages. In *7th International Conference on Software Language Engineering (SLE)*. Springer, 2014.

[13] T. Kühn, B. Stephan, S. Götz, C. Seidl, and U. Aßmann. A Combined Formal Model for Relational Context-Dependent Roles. In *International Conference on Software Language Engineering*, pages 113–124. 2015.

[14] M. Leuthäuser. Scroll - a scala-based library for roles at runtime. Technical report, 2015.

[15] P. R. Lewis, A. Chandra, S. Parsons, E. Robinson, K. Glette, R. Bahsoon, J. Tørresen, and X. Yao. A survey of self-awareness and its application in computing systems. In *Fifth IEEE Conference on Self-Adaptive and Self-Organizing Systems, SASOW 2011, Ann Arbor, MI, USA, October 3-7, 2011, Workshops Proceedings*, pages 102–107. IEEE Computer Society, 2011.

[16] Oracle. The invokedynamic instruction. https://docs.oracle.com/javase/8/docs/technotes/guides/vm/multiple-language-support.html#invokedynamic, march 2016. [Online; accessed 18-March-2016].

[17] M. Pradel and M. Odersky. Scala roles: Reusable object collaborations in a library. In *Software and Data Technologies*, pages 23–36. Springer Berlin Heidelberg, 2009.

[18] R. Seiger, S. Huber, and T. Schlegel. PROtEUS: An Integrated System for Process Execution in Cyber-Physical Systems. In *Enterprise, Business-Process and Information Systems Modeling - 16th International Conference, BPMDS 2015, 20th International Conference, EMMSAD 2015, Held at CAiSE 2015, Stockholm, Sweden, June 8-9, 2015, Proceedings*, pages 265–280, 2015.

[19] F. Steimann. On the Representation of Roles in Object-oriented and Conceptual Modelling. *Data & Knowledge Engineering*, 35(1):83 – 106, 1999.

[20] G. Wielenga. On powerjava: "roles" instead of "objects". https://blogs.oracle.com/geertjan/entry/on_powerjava_roles_instead_of, jan 2013. [Online; accessed 28-May-2014].

[21] R. Wong, H. Chau, and F. Lochovsky. A Data Model and Semantics of Objects with Dynamic Roles. In *Data Engineering, 1997. Proceedings. 13th International Conference on*, pages 402–411, Apr 1997.