

Using Role-Based Composition to Support Unanticipated, Dynamic Adaptation - Smart Application Grids

Christian Piechnick, Sebastian Richly, Sebastian Götz, Claas Wilke and Uwe Aßmann

Technische Universität Dresden

Software Engineering Group

01062 Dresden, Germany

Email: {christian.piechnick, sebastian.richly, sebastian.goetz1, claas.wilke, uwe.assmann}@tu-dresden.de

Abstract—Due to the wide acceptance and distribution of mobile devices, it has become increasingly important that an application is able to adapt to a changing environment. This implies the necessity to integrate varying functionality at runtime being activated depending on the current context. A common approach is to foresee and model all possible influencing factors and to integrate the required software building blocks in advance. But, due to the constant change of the environment, as described by Lehman's laws, it is impossible to anticipate all future situations. Hence, modeling the entire adaptation process at design time prohibits the adaptation to unanticipated scenarios and, thus, is likely to lead to the malfunctioning of the adaptive application in the future. In this paper we focus on unanticipated, *dynamic self-variation* of applications (i.e., without a central coordinator) and propose a *role-based composition system* that enables the adjustment of the structure and functionality of software-objects in a fine-grained manner. Systems following our proposed approach form a Smart Application Grid (SMAG). The SMAGs-Approach is putting emphasis on dynamic collaborations between components within an application and between several different software systems. Therefore, *role-modeling* is used to model and perform dynamic variation of applications at runtime, whereby roles are stored in central repositories. This allows the integration of previously unknown software-building-blocks and the dynamic adaptation to situations that were not foreseen.

Keywords-*Dynamic Variation; Unanticipated Adaptation; Role-Modeling; Composition; Repository.*

I. INTRODUCTION

Software has become ubiquitous and plays an essential role in almost every area of life, ranging from small personal tasks in everyday life to coordination and partial autonomous control of global economic processes. Our current world is characterized by high dynamics and the pressure of constantly adapting to a changing environment. This, however, leads to the requirement that software applications have to adjust themselves to changing requirements just as quickly. These adaptation processes, which in most cases cannot be predicted due to the complexity of open world scenarios, traditionally rely on static software development life cycles. As part of the establishment and widespread acceptance of mobile devices, their applications are becoming a fundamental part of daily life. This further increases the pressure

to automatically adapt an application at runtime to its constantly changing application context (e.g., location, time, task and collaboration partners) [17]. In complex scenarios, the possible values and changes in external conditions are usually not predictable. This makes it impossible to plan all potential adaptation operations at development time.

Unlike conventional product enhancements or patches within the software development process, dynamic context adaptations are small, spontaneous and fine-grained runtime changes in both application structure and functionality. In large, monolithic applications, where only large building blocks can be exchanged, adaptation processes can become very complex and costly. In contrast, dynamic networks of small cooperating applications that are prepared on variability in advance are more suitable for runtime adaptation. In this way, the timespan from the occurrence of a changed requirement to an adequate adaptation of the application is significantly reduced and costly product development cycles are avoided. The finer-grained the interchangeable elements of the application are, the better the impact of change operations on the integrity of the entire system can be estimated. This is because the elements relate to specific functions, which in the ideal case have only limited effect on the correctness of the entire application.

The introduction of application platforms, especially in the field of mobile devices, makes the modeling and description of collaborative relationships between individual apps very important. Apps are usually small, isolated working applications that serve a narrowed specific purpose. Multiple apps form a complete system, whereby each individual application serves one specific concern of the overall system. In this context the modeling of collaborations makes synergies visible. The main problem, however, is that application developers can neither know, which other apps are deployed on a target device nor what interfaces they may offer. In addition, no statement can be made about, which applications will be developed and published in the future, which could be used to extend the overall functionality. Unlike the composition of services in a Service-Oriented Architecture (SOA) using orchestration or choreography [20],

mobile platforms do not offer an additional layer to describe composite processes. More, even if such a mechanism would exist, the definition of such composite processes can always only refer to those apps, which are known at design time of the process. That is why the applications themselves need to be able to establish dynamic collaboration relationships.

To realize dynamic, context-aware adaptation, applications need to be modified at runtime. *Role-Oriented Programming (ROP)* [15][26] is an appropriate basis to realize dynamic variation in a fine-grained and collaboration-based manner: This approach of modeling and programming systems is an extension to the classic object-oriented paradigm. The term *role* (or object role) is not related and cannot be compared to the classical role term in workflow systems - also called Role-Based Access Control model (RBAC). In a role universe, core types (i.e., classes) and a set of role types exist. A core object (i.e., instance) can play role instances, if the respective core type and role type are linked by a *can-play-a* relationship. For example, a *Person* can play a *Father* and a *Customer* role type. Players are able to start and stop playing roles at runtime, without losing their own identity. Notably, roles change the behavior of core objects (like aspects in Aspect-Oriented Programming (AOP)) and are able to store additional data, which is only applicable for the respective role and not for the core itself. Beyond that, the role-oriented approach provides a rich repertoire of modeling concepts, which are suitable to describe the semantics of individual exchangeable components and their relationship with respect to the overall system. With ROP roles can be used to model dynamic variation to manipulate the structure, the functionality and the relationships of objects within a single and between several applications.

In this paper, we present a new kind of software architecture—**SMAG** (Smart Application Grid)—in which applications are no longer monolithic, but composed of many small, distributed applications that link to each other dynamically like in a grid. Role-based modeling is used to adapt these applications at runtime, by (a) fine-grained structural and functional changes of application components and (b) dynamically connecting components within one and between several applications. To deal with changes that the application developer did or could not foresee at design-time, exchangeable building blocks are stored in repositories, which can be retrieved and integrated at runtime. This allows the integration of previously unknown components and enables the adaptation to unanticipated scenarios.

This paper is structured as follows: In Section II, we give a short summary about software adaptation. In Section III, we present an overview about the current state of the art w.r.t. dynamic software composition techniques. The Smart

Application Grid approach is described in Section IV and an example is described in Section V. Finally, Section VI presents our conclusion and future work.

II. CONTEXT-AWARE SOFTWARE ADAPTATION

Applications that operate in highly dynamic environments, in which context changes and resulting modified requirements often cannot be predicted, can hardly be developed with traditional software development processes. Dynamic adaptive systems (DAS) address this problem by explicit specifications of possible context-triggered runtime changes of the application's functionality and structure during the software development process [6]. The *application context* is defined as the sum of all measurable properties of the application itself and its environment. One possibility is to manually change the application structure, which usually implies high efforts, because of the potential high frequency of context changes and the complexity of the required modification processes. Thus, application architectures are required, that provide runtime support automatically detecting situations, where the application does not match the current context, and to determine and execute the required change-operations to adjust the application accordingly (*adaptive applications*) [24]. Nevertheless, in recent years, numerous development methods, software architectures and adaptation systems for DAS have been developed that base on different approaches [10][12][13][14][17][19], whereof most base on the **feedback loop** [11]. Following this concept, DAS need to (a) identify changed external conditions, (b) analyze the application and its context, (c) make decisions based on those findings on how to adapt and (d) manipulate the system that it fits to the current application context. Another distinguishing factor of DAS is the degree of anticipation w.r.t. the adaptation process. Adaptive systems with *anticipated adaptation* define all possible contextual situations, application variants and their relationship at design time, whereby the system variation is performed at runtime according to predefined rules. In addition, an adaptation to situations that have not been considered during the development process is impossible. This contrasts with applications that support *unanticipated adaptation*. Here, the context model, the analysis mechanisms and the adaptation planning must be open and extensible at runtime. Furthermore, new components—that were not considered at design time—need to be integrated into an application dynamically.

As stated in [13], adaptation can be classified as *parameterized* and *compositional adaptation*. Adaptation by parameterization is achieved by manipulating predefined parameters of software entities. Adaptation by composition refers to the replacement or extension of software components (cf. Sect. III). The adjustment of software elements' parameters is the most trivial solution to adapt an application to a changed environment, which makes the sphere of influence rather limited. It is not possible to actually modify the

functional parts of software units at runtime. Nevertheless, it is possible to change their behavior in a predefined manner by setting prescribed parameters. The implementation code then has to be aware of these parameters, which leads to the problem that adaptation logic is scattered over the application logic. The replacement of software component implementations, on the other hand, makes it possible to actually replace functional and structural elements of an application. When only a small functional part of a system component needs to be altered to adapt it to the application context—for example the enlargement of a message buffer of a communication component—the complete replacement of the component might not be suitable. For that reason, a combination of both adaptation mechanisms is desirable.

To actually adapt an application to a changed environment, two things are essential: the application structure has to be (1) queried and analyzed and (2) the application needs to be modifiable. Under this assumption, component-based software development (CBSD) seems to be best suited for the implementation of adaptive systems. A component-based application consists of reusable components, explicit interfaces and connections. All information that is necessary to use a component is defined in its interface description which will not change frequently. Thus, components having the same interface description are syntactically substitutable, which allows changing the application's functionality by replacing individual components. According to Szyperski “a software component can be deployed independently and is subject to third-party composition” [28]. This definition provides the basis to transfer the idea of components-off-the-shelf to DAS. Components are stored in a central repository and grouped by their interface descriptions, so that an application developer can choose from a set of existing components to compose a software system. At runtime those building blocks can be exchanged, based on the current application context as well as the functional and non-functional properties of the different component-implementations.

In this paper, we focus on the process of runtime application modification (i.e., the **act phase** (d) of the feedback loop). We will outline why traditional component replacements lead to problems, how extended software development techniques, like AOP, are able to address them and show the limitations of these extended techniques. Afterwards, we present a novel dynamic software architecture, based on role-oriented modeling, which overcomes these limitations. We will present a role-based composition system, which provides basic functionality for dynamic, unanticipated adaptation.

III. RELATED WORK

This section describes possible methods and state-of-the-art approaches for runtime changes of the structure and functionality of software systems. Our goal is to outline the

problems and limitations of known techniques w.r.t. dynamic adaptation.

A. Classical Component Replacement

Some adaptive systems, such as the three-layer energy auto-tuning runtime environment (THEATRE) of the energy auto-tuning approach (EAT) [14], realize (non-)functional runtime variability by exchanging component implementations. This method is based on architectural modeling, where each component type can have multiple implementations. At runtime, for each component type a concrete implementation is chosen to compose the actual system. If the application context changes and the currently selected component is not best suited for it, the adaptation system will replace the specific instance by another component that implements the same component type.

This, however, leads to several problems. Since software components can become very large units, the replacement of a whole component implementation produces much overhead, especially when only a small set of functionality is subject to change. Beyond that, the state of the replaced component needs to be transferred to its substitute or gets lost [12]. Furthermore, all connections between other components and the replaced one need to be updated accordingly. Depending on how long it takes to setup the new component instance, the system state can be temporarily inconsistent.

B. Aspect-oriented Programming

AOP is a paradigm for object-oriented programming, which enables the separation of code fragments that do not relate to the actual core functionality and occur at several points in the program (e.g., security, logging, transaction) from the actual application logic (separation of concerns) [18]. Run-time weavers allow for dynamic aspect integration at execution time. By this means, the application logic can be changed at execution time, making AOP an excellent foundation for implementing dynamic adaptive systems, such as in the DiVA-System [2]. Traditional aspect implementations (e.g., AspectJ [1]) are implemented as white-box code-composition systems imposing a huge complexity for large systems. To cope with this complexity, Model-Driven Software Engineering (MDSE) is used, because model-based representations of an application at runtime allow for an appropriate degree of abstraction [19]. In the DiVA-Project a combination of model-based techniques and AOP was used to support dynamic application variation through aspects, which is called *SmartAdapters* [19]. The SmartAdapters-approach is a generic composition mechanism, but relies on aspects, which still suffer from some fundamental problems: First of all, aspects in general do not have a state. Let us suppose an aspect, which is implementing a message buffer. This aspect would potentially be deployed if the network bandwidth of a communication component decreases. The buffer size, however, would be

implicitly implemented in the advice code and could neither be queried nor changed after the advice was integrated. Second, aspects do not offer a notation to describe collaboration relationships. If the implementation of an advice needs access to the functionality of another component, this connection would be hidden in the advice code. This makes the explicit change of collaboration partners nearly impossible. Because aspect weaving is a code-composition technique, all dynamic variations are class-based modifications. This hinders an instance-based adaptation, because in programming languages like Java all objects share the same method declaration.

C. Context-oriented Programming

Context-oriented programming (COP) is a programming approach based on object-oriented programming that treats context-awareness as a first-class citizen on the level of a programming language [16]. COP extends the collaboration-based dispatch presented in [25] by another context-dimension. Therefore, within a class definition several layers can be declared. Each layer can contain several methods that override given methods of its enclosing class. At runtime the caller of such a method can either explicitly or implicitly specify, whether a layer is active or not. When a layer is active, the call is dispatched to the method of this layer first. By means of this approach, dynamic adaptation through context-dependent dispatch can be realized. Nevertheless, the adaptation-capabilities are rather limited. First of all, there is neither a mechanism to describe dependencies between several layers, nor how layers relate to the application context. The layer activation/deactivation is based on the code that is calling a method on a layer-enabled object. This enables instance-based adaptation because the layer activation/deactivation depends on the context, in which the method of an individual object is called. Like aspects, layers do neither provide a state nor collaboration models. Furthermore, unanticipated adaptation is not possible because layer declarations have to be specified inside a class definition at design-time.

D. Composition Filter

The composition filters approach [7] enables dynamic adaptation by intercepting messages. In contrast to AOP, the composition filters approach achieves the addition and removal of aspect logic, without changing the core class. Furthermore, the conventional object model is extended so that incoming and outgoing messages can be manipulated.

A composition filter class consists of one or more internal classes. Around the composition filter object, an interface part is introduced, which forms the access layer to attributes and methods of the actual core objects. Within this access layer, filters can be added and removed. For each filter, rules are defined that specify, which messages are actually processed and, which are ignored. When a new message

arrives, each filter checks if the message is relevant to them. If this is the case, the filter can manipulate the message, forward it to other objects, throw an error or run external code. Subsequently, the potentially modified message will be forwarded to the next filter, until it finally arrives at the actual addressee.

Filters can be dynamically composed in a black-box style and are declared and implemented transparently without the necessity of class-code manipulation. That is why they are very suitable for the implementation of a dynamic composition system and realization of dynamic adaptive applications. In Section IV, this approach is extended with explicit collaborative relationships, by combining composition filters with role-based programming.

IV. SMART APPLICATION GRIDS

Smart Application Grids consist of many small, distributed applications that are linked dynamically. To adapt these composite applications at runtime to context changes, the *individual* applications are dynamically modified and *collaboration relationships* are changed depending on external conditions (i.e., conditions of the context) using roles. The main goal is to develop a composition system that is transparent, supports reuse of composition programs and is technology-independent. In addition, through metamodeling it should be possible to investigate the application structure using models at runtime, where model changes should affect the running application transparently. This paper specifically focuses on the underlying dynamic role-based composition system, not on specific adaptation mechanisms. Therefore, first the concepts of role-based modeling are summarized and, subsequently, the concepts of our proposed role-based composition system as well as the associated repository are introduced. According to Aßmann, a composition system consists of a component model, a composition technique and a composition language [5]. In the following, the SMAG composition system is presented according to these three aspects. Finally, an exemplary adaptation architecture and our reference implementation in Java are presented.

A. Role Modeling

Although there is no uniform understanding of the concept of roles in software engineering, a consensus has emerged that roles are an important element of software design. In this work the concept of a *role* is used according to the work of Riehle [23] and Reenskaug [21] and is briefly summarized in this section.

A **role** is a *dynamic view* or a *dynamic service* of an object in a *specified context*, offering the possibility of separation of concerns, interface-structuring, dynamic collaboration description, and access restriction. A role is clearly specified by a **role type** and can be played and removed from an object at runtime. When an object plays a role, they both share the same identity (i.e., a role does not

have its own identity), whereas the number of roles played simultaneously is not limited. It is also possible that a role plays other roles. Let us consider a role *Employee* which is played by an instance of a class *Person*. This role could play another role of the type *Software Developer*. Roles have their own properties and methods, whereas the role-playing object behaves according to the functionality, defined by the role. Furthermore, the object state is extended by the properties of the roles it is playing. If an object of the class *Person* is playing the *Employee* role, it might get an additional attribute *salary* and an additional method *work()*. Any call to the core object is first dispatched to its roles. Because roles can have references to other roles and because they must be played by objects, roles provide means for describing dynamic collaborations, spanning a varying network of dynamic relationships.

In a class diagram, classes and their relations are modeled. Analogously, a **role model** specifies role types and their relationships. In this way, it describes object collaborations, since the instantiated roles necessarily have to be played by objects or other roles. Usually classes expose public methods, which are used to establish interaction between objects (i.e., instances of those classes) by exchanging messages (i.e., calling methods and passing parameters). Associations in class diagrams however, neither provide means for describing under which circumstances objects collaborate nor which part of its interface (i.e., set of attributes and methods) is relevant w.r.t. this relationship. A role model on the other hand, describes only a single concern of the object collaboration which allows for separation of concerns. Role models can then be composed hierarchically to express multi-concern object collaboration [22]. This increases the degree of reuse because domain dependencies can be controlled in a fine-grained manner. In this way, partial architectures can be specified, shared and reused.

Subsequently, role and class models are merged to **role-type-class-models**, by binding all role types to classes. Riehle describes several role restrictions that can be used to control, how role types can be applied to classes [23]. The set of all role types of a class is called role-type set which specifies what types of roles an instance can play. Traditionally in class modeling, static associations are used to express possible interaction relationships between objects. The modeling of object collaborations through role types allows for a fine-grained description of dynamic collaborations between several objects under certain conditions (role context). Through this modeling approach, both interactive relations that change at runtime and the subset of methods that are involved in this interaction can be documented. Figure 1 gives an example. The two role models "Strategy" and "Observer" are composed to new role model "Strategy and Observer", by introducing a role-equivalent restriction. This restriction claims that every object that plays a role of the role type *Strategy* must also be capable to play a role

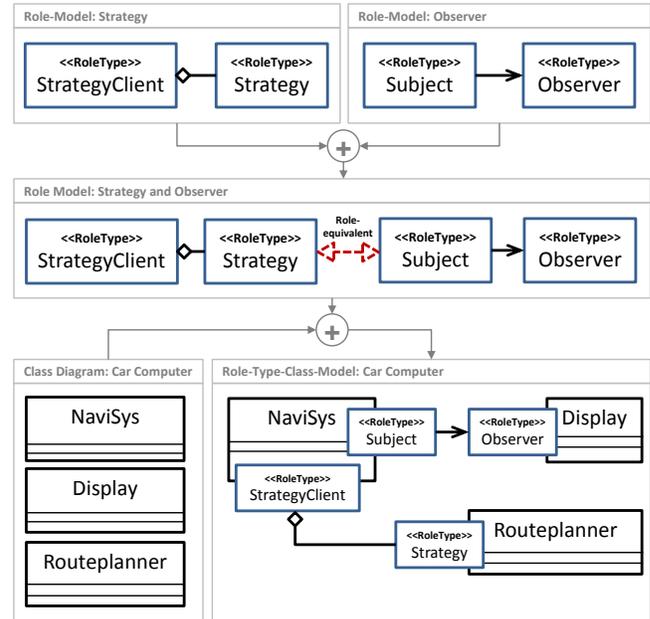


Figure 1. The Relationship between the SMAG Component Model and Role-Based Modeling.

of the role type *Subject* and vice versa. This role model is then combined with a class diagram "Car Computer" to a role-type-class-model. This model describes, which class instances are able to play a given set of roles. Lets consider an instance of the class *NaviSys* is playing a role of the role type *Subject*, it is related an object playing the role *Observer*. After the role has been removed, this relationship is removed either.

The role-based software development approach is primarily an approach on the modeling level, whereas different approaches were suggested on how to implement the role-based approach. The lack of a silver-bullet solution leads to a deep gap between design and implementation. There is the possibility of using classical object-oriented concepts, such as interfaces, multiple inheritance or mixin inheritance to realize the presented concepts [27]. However, this means that roles can not be acquired or removed at runtime. Another solution is the Role Object Pattern (ROP) [8] which separates the core and the role objects into different classes, using delegation to interact between them. This leads to a nontransparent role-binding mechanism, because the core object has to manage its roles. In addition, several attempts have been made to integrate the role concept in programming languages. ObjectTeams/Java (OT/J) [4], an extension of the Java programming language and part of the Eclipse platform, is one of them. Despite the current OT/J implementation does not fully realize the role-concept described earlier, it was still used for the reference implementation of SMAGs for Java (cf. Section IV-E), due to the lack of mature alternatives.

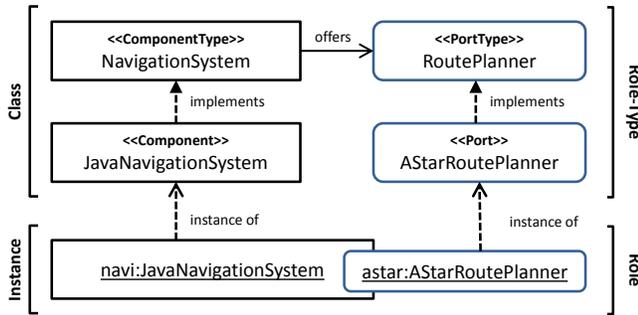


Figure 2. The Relationship between the SMAG Component Model and Role-Based Modeling.

B. Component Model

SMAG Components are stateful, self contained software modules that can be developed and deployed independently. They are described by a **ComponentType** which describes the functional interface of a component by grouping several **PortTypes**. A PortType represents a language-independent interface description which can be offered or required by a component. Each PortType has a unique name and specifies the services of those components that provide this interface. It defines both a functional view by method signatures and a state-based view by a list of externally manipulable parameters. Each PortType is implemented by one or several **Ports** which are the elements that can be added or removed at runtime. There are, however, certain state attributes that have to be preserved in spite of the substitution process. To realize this requirement, without the use of elaborate mechanisms for transferring each individual state value, a PortType can specify a set of attributes (*SharedMemory*) that are managed within a component and not within a replaceable unit. Furthermore, a PortType can specify a number of other PortTypes that are required to provide the desired functionality. Each required PortType is annotated with a multiplicity to specify how many instances can be managed by a Port. In analogy to the ACOE-component model [10], PortTypes are further distinguished in *BehavioralPortTypes* and *EventPortTypes*. A BehavioralPortType allows access to application functionality defined by it, whereas an EventPortType provides an event at which other components can register on. Besides the actual application functionality, a Port which is implementing a given PortType, must declare some metadata to give information about whether it is suitable for the requirements of the specific domain. As mentioned earlier, the composition system is optimized for a possible runtime adaptation. Therefore, metadata should be automatically computable. This can be achieved by using semantic technologies like URIs to concepts of shared ontologies [9]. Finally, a component is implemented, using a specific programming language. First, the language-independent ComponentType declaration is transferred into

a language artifact. This is done automatically using a platform-specific IDL compiler. The component can offer a standard-implementation for any method that was specified in the provided Port-Types. In addition, they can implement an install script and an uninstall script that is executed when the component is instantiated or destroyed. In this way, any required resources can be allocated or released.

As already mentioned, the presented composition system implements the ideas of role-based software development with the semantics of the composition filter approach [7]. Figure 2 shows the relationships of the elements of the component model to the concepts of role-based designs. A PortType corresponds to a Role-Type, a Port to a Role, a ComponentType to a Class and a Component to an object. Under this assumption, the specification of PortTypes coincides with the creation of a role model, whereas the required PortTypes of a given PortType form the collaborations. The declaration of ComponentTypes is similar to the design of a class-role-type-model, as PortTypes, representing role types, are mapped to ComponentTypes. At runtime, an instance (Component) may play roles (Ports), thereby changing its behavior.

The proposed component model defines software components as modular units with explicitly defined interfaces. Thereby, components that implement the same ComponentType as well as Ports that implement the same PortType are syntactically substitutable. Initially when a component calls a method of a connected component via a required/provided PortType, the base implementation of this method within the actual component is executed. When however, a Port is bound to a provided PortType of a component, then the method implementation of this Port is called first. As described in the next section, the Port can process the method call completely or partially, by adding additional functionality and passing the call to the underlying implementation (i.e., another Port or the base implementation of the Component). Ports can be bound to Components dynamically, thus changing their structure and behavior w.r.t. to a specific PortType. Ports in addition, offer internal state variables as well as external parameters, to tailor them w.r.t. to a concrete reuse context.

C. Composition Technique

At runtime, the required PortTypes of a component instance have to be connected with offered PortTypes of other components. This is done by **passive connectors**. In some systems, architectural connectors are active elements which perform type conversions or protocol adjustments. As we will explain, these requirements can still be implemented, using specific Ports. Components can either be extended by inheritance at design-time or by an *extend operator* at runtime. The extend operator introduces new PortTypes, by manipulating the architectural model that is managed

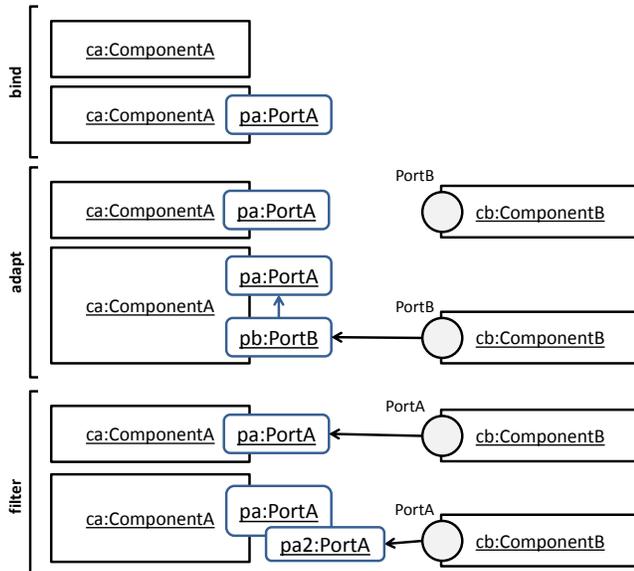


Figure 3. The Bind, Adapt and Filter Composition Operators with an Example.

dynamically. Each change in the architectural model is then transferred to a change of the corresponding application.

The main composition operators of the presented composition system are the *bind* and the *unbind operator*, which bind a Port to a Component respectively removing a Port from a Component. By binding a Port to a Component, all invocations of methods that are specified in its PortType will first be dispatched to the Port. Depending on the implementation, the Port can provide the whole functionality or just performs some extra tasks and then forwards the call to the actual component or an underlying stacked Port. By removing a Port, the internal state gets lost. For this reason, each component manages a shared memory which keeps state information regardless of the existence of bound Ports. In the presented approach, connectors can connect provided and required Ports only, if they share the same PortType. In practice, however, it is common that the interfaces of reused components differ from the needed ones. In this case, an interface adaptation can be carried out by special *AdapterPorts*. An AdapterPort is usually a lightweight Port that implements exactly one PortType and has exactly one required PortType. Figure 3 illustrates this concept. In the implementation of the adapter, the two interfaces can be mapped by translating each call to the offered interface Pa to an appropriate call to the required interface Pb. A special case of adaptation is the remote communication between distributed components. If the required functionality is provided by a component that is not deployed on the same system, an AdapterPort can automatically be generated which translates each local to a remote call.

To separate cross cutting concerns (e.g., persistence, log-

ging, security) from the actual application functionality, which is realized through Ports or the component implementation, *FilterPorts* can be used. A Filter implements a given PortType and requires a Port of the same type. In analogy to the role modeling, in which a role can also play other roles, a port instance can be decorated with several FilterPorts. Depending on the implementation of the filter, both incoming and outgoing messages can be processed. Figure 3 shows this process schematically. Furthermore, FilterPorts can be used to offer a set of methods of a component to other applications by providing a remote interface. Therefore a RemoteFilter would publish some kind of remote interface (e.g., as a web-service) during the binding process. This service can then be published in a central repository, making the offered functionality visible to other applications. This creates a dynamic heterogeneous network of collaborating applications.

D. Composition Language

For developers of SMAG applications, two different modeling levels are available, on which they can describe architectures. On the one hand, there is the possibility to specify metaarchitectures, which are modeled by only using Component- and PortTypes. As explained earlier, this approach is in direct connection with the creation of a role-model and the generation of role-type-class-model. Nevertheless, no statement is made, which concrete Component implementations will be used and what Ports will be deployed at runtime. Since Component- and PortTypes are independent of a concrete programming language, which are implemented by artifacts of a concrete platform, it is possible to specify platform-independent and reusable architectures. Those partial architectures can be reused through model composition.

On the basis of the metamodel, an architectural model can be specified by choosing a component implementation for each ComponentType. At runtime, the architecture description is preserved, so the runtime environment can create a direct connection between the instantiated software artifacts and corresponding model elements. The runtime environment is always able to translate changes in the application model to changes in the application itself. In general, object-oriented programming languages support introspection, enabling the runtime environment to collect information about the objects. In most cases this approach can also be used to derive an application model at runtime which leads to the realization of a complete round-trip. When both strategies are applied, it can be ensured that the architectural model and the structure of the actual application are synchronous.

E. SMAG-Repositories

To reuse SMAG model elements and software artifacts efficiently and finding them at runtime, they must be published in a central location. This idea is not new and

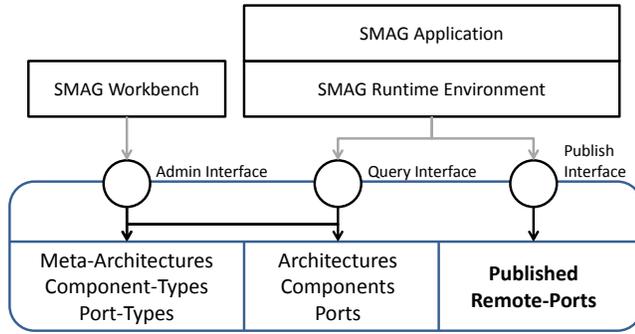


Figure 4. SMAG-Repository Structure.

has already been implemented several times. Thus, web services for example can be published and discovered using a uniform standardized directory service, called Universal Description, Discovery and Integration (UDDI). Although UDDI has been standardized by OASIS, it did not become very popular. One of the main reasons is the required effort implied by large specification and the complex process of service publication. For this reason, we tried to keep the specification of SMAG-Repositories as simple as possible.

A SMAG-Repository has three main tasks. On the one hand, artifacts must be published and managed as well as searched and reused. On the other hand, applications must be able to publish functionality at runtime that can be used by other SMAG applications. SMAG repositories consist of a static directory for Component- and PortTypes, MetaArchitectures, Architectures, Components and Ports. Each artifact is identified by a unique URI and can be described using various metadata that is made computable by referencing OWL concepts. Through an administration interface they can be published, unpublished or modified. Components and PortTypes, MetaArchitectures are stored as platform-independent models, whereas Components and Ports are stored as platform-specific binary packages. A publish interface allows a SMAG application to provide a remote service by specifying a PortType, a calling address and a description of the used communications technology. Other applications are able to query this directory services, which enables the setup of dynamic collaboration between various applications. This corresponds to the implementation of a Trader Service, which is looking for a service based on its functional properties and characteristics. Each repository can reference an unlimited number of partner directories, so queries can be forwarded if a repository is not able to deliver results for a given request. Figure 4 illustrates the structure and relationship of the SMAG repositories. Each SMAG Repository in turn is a SMAG application whose architecture is based on a fixed metaarchitecture. Thus, it can be implemented for different target systems and dynamically changed, according to specific needs. The three public interfaces (a) administrative, (b) search and (c) runtime

publication, are made available through RemotePorts.

F. Reference Implementation

The idea of Smart Application Grids is basically platform- and technology-independent. The model-based description of ComponentTypes and Ports does not make any statements about a specific programming language. However, Components and Ports will be implemented in a specific language using a specific runtime environment. Thus, the component model has to be mapped to concepts of the target language, whereas the runtime environment has to implement the given composition operators.

A reference implementation was created using the Java programming language in combination with the role-based language extension ObjectTeams/Java. Accordingly, ComponentTypes and PortTypes are mapped to Java interfaces, Components to classes and role models to abstract OT/J-Teams. Ports are implemented through specific OT/J-Roles. Since OT/J currently only supports load-time-weaving, theoretically, no new roles can be added after the class, that should play the role, was loaded. This hinders the promised support for the dynamic introduction of previously unknown Ports. To circumvent this drawback, a proxy team is generated, which contains proxy roles for each PortType. The Ports are added to the proxy teams at runtime by delegation. This workaround, however, does satisfy all presented requirements regarding transparent role playing. To perform the composition operations at the application level, the runtime environment must manage a model representation of the application architecture. For both the metaarchitecture and the architecture, Ecore-based metamodels were created. Based on this representation, models are used to query and modify the application at runtime. By model-to-text transformation, all Java artifacts can be generated, that are not reused and all reused elements can be obtained from a repository.

V. EXAMPLE

To demonstrate the presented results, an exemplary adaptation system was developed. Figure 5 shows the basic architecture. First, a distinction is made in a real and a virtual context, whereas sensors (sensor layer) monitor the real-world context and transfer measured values into an object-oriented representation. The inference layer is notified of context changes. It then can put the different context characteristics into relation or generate new knowledge. The adaptation layer makes sure that the running SMAG application fits the current application context. Therefore, the application architecture is queried using the query interface of the SMAG runtime environment and is compared to the requirements that result from the context. Subsequently, if necessary, it creates a reconfiguration script that consists of individual reconfiguration operators that are executed against the manipulation interface of the runtime environment. The

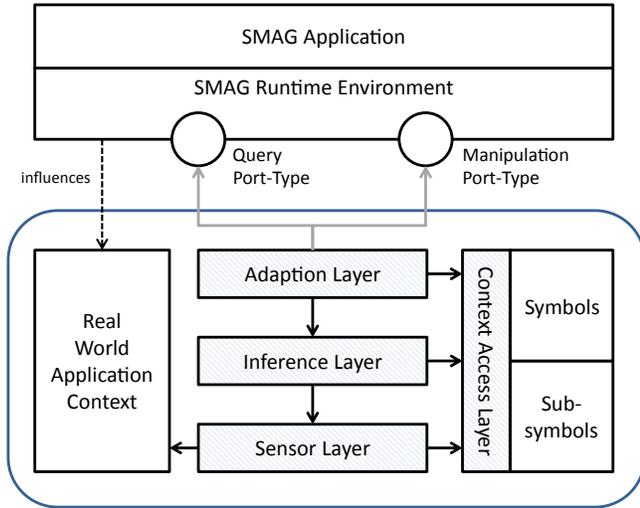


Figure 5. The General Architecture of the SMAGs-Adaption System.

adaptation system was implemented as a SMAG application. This allows to alter the algorithms and mechanisms that are used to gather and process information as well as the execution of adaptation operations at runtime. This, however, in combination with the possibility to integrate Ports that were not known at design-time, enables the realization of unanticipated adaptation. The adaptation methods that were used for evaluation, are based on ECA-rules using the JBoss Drools system [3]. Nevertheless, this approach is neither new nor sophisticated, it is still suited to ensure the proper functioning of the dynamic composition system.

In Figure 6, the runtime architecture of an example system is shown, which consists of four components: CarComputer, NavigationSystem, Radio and SmallDisplay. The NavigationSystem component offers a PortType RoutePlanner, which provides the functionality to calculate a route. In the shown configuration, a port is attached to the component, which is implementing an A*-algorithm, based on locally stored maps. When the memory of the system exceeds a specified threshold and an Internet connection is available, this port is dynamically replaced by an implementation, which is using a web service.

The Radio component provides an interface to query all radio channels that are available. Depending on, which driver was detected and whether any profile information is available, a channel filter can be deployed dynamically, which only shows radio channels relevant to the current driver. When the driver changes, the RadioGenreFilter is either parameterized with another genre or is removed.

The SmallDisplay component however provides an interface to display a list of strings, whereby the CarComputer component requires an interface to display a list of radio channels. Those interfaces can be mapped using an adapter port.

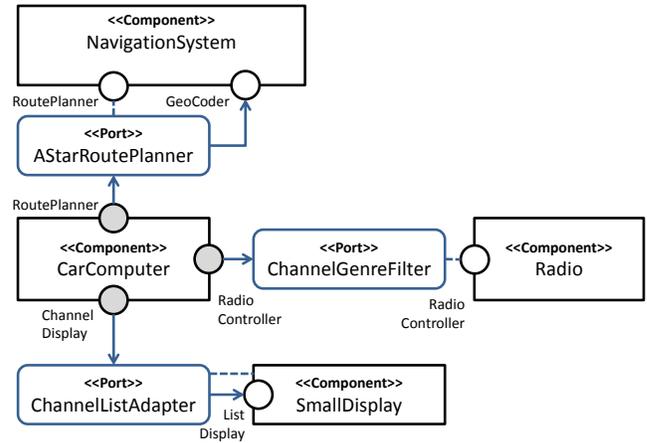


Figure 6. A Simplified Diagram of the Run-Time Architecture of an Example Application.

VI. CONCLUSION AND FUTURE WORK

In this paper, it was shown that role-based modeling can be transferred to component-based software design, in order to create applications that can be manipulated in a very fine-grained manner at runtime. Furthermore role-based modeling is putting emphasize on dynamic collaborations, which makes it possible to create dynamic relationships between several components within one application and between several ones. The proposed SMAGs concept allows to create clearly structured and reusable application architectures using role-based modeling, which makes it an ideal candidate for the realization of dynamic adaptive systems. This is because roles are stateful, functional units with clearly defined interfaces that can be dynamically merged with objects. Notably, the role-based modeling approach can be connected with the basic principle of composition filters very well. The implementation effort of SMAG applications is limited to the implementation of the actual application logic, since the majority of software artifacts can be generated automatically using models. Even though the presented adaptation mechanisms are merely exemplary, the adaptation logic could be clearly separated from business logic and the software system could be changed at runtime, supporting unanticipated dynamic adaptation. The concept is largely based on the use of a central repository to manage all modeling and implementation artifacts. In this way, implementations of ports that were not known during design-time can be integrated into a running application or old versions of existing ports can be replaced with newer ones, enabling hot updates.

However, many open issues remain that need to be refined in future work. The most important aspect is the support of the development process by appropriate tools. Furthermore, it needs to be discussed, how the consistency of an application, in terms of structural changes by composition operations,

can be guaranteed. Since the concept describes structural changes at the level of architectural models, model-based validation would be appropriate. In addition, the semantic substitutability of different port implementations needs to be ensured. This corresponds to the fundamental problem of trust within components-of-the-shelf, so it needs to be evaluated, whether certification can solve this problem. The presented adaptation architecture needs to be equipped with advanced adaptation mechanisms, based on the semantic description of context values. In addition, it may be possible to correlate the role context, in which an object is playing a role, with the application context, to automatically deploy role bundles and maybe enable autonomous and unanticipated dynamic adaptation.

ACKNOWLEDGEMENT

This research has been funded by the European Social Fund and the State of Saxony (Project ZESSY #080951806).

REFERENCES

- [1] AspectJ. <http://www.eclipse.org/aspectj/>. 11.05.2012.
- [2] DiVA-Project. <http://www.ict-diva.eu/>. 11.05.2012.
- [3] JBoss Drools. <http://www.jboss.org/drools/>. 11.05.2012.
- [4] OT/J. <http://www.eclipse.org/objectteams/>. 11.05.2012.
- [5] U. Aßmann. *Invasive Software Composition*. Springer Verlag, New York, USA, 1st edition, 2003.
- [6] N. Bencomo, P. Sawyer, G. S. Blair, and P. Grace. Dynamically adaptive systems are product lines too: Using model-driven techniques to capture dynamic variability of adaptive systems. In *SPLC (2)*, pages 23–32, 2008.
- [7] L. Bergmans. The Composition Filters Object Model. Technical report, Dept. of Computer Science, University of Twente, 1994.
- [8] D. Bumer, D. Riehle, W. Siberski, M. Wulf, and M. Wulf. The role object pattern. In *Washington University Dept. of Computer Science*, 1997.
- [9] M. Cremene, M. Riveill, and C. Martel. Unanticipated dynamic adaptation of context-aware services. *Acta Technica Napocensis*, pages 30–35, 2008.
- [10] B. Ding, H. Wang, D. Shi, and X. Rao. Towards unanticipated adaptation: An architecture-based approach. In *Software Engineering Research, Management and Applications, 2009. SERA '09. 7th ACIS International Conference on*, pages 103–109, 2009.
- [11] S. Dobson, S. Denazis, A. Fernández, D. Gaiti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, and F. Zambonelli. A survey of autonomic communications. *ACM Trans. Auton. Adapt. Syst.*, 1(2):223–259, 2006.
- [12] P. Ebraert, T. D’Hondt, Y. Vandewoude, and Y. Berbers. Pitfalls in unanticipated dynamic software evolution. In *RAMSE*, pages 41–50, 2005.
- [13] J. Fox and S. Clarke. Exploring approaches to dynamic adaptation. In *Proceedings of the 3rd International DiscCoTec Workshop on Middleware-Application Interaction, MAI '09*, pages 19–24, New York, NY, USA, 2009. ACM.
- [14] S. Götz, C. Wilke, S. Cech, and U. Aßmann. Runtime variability management for energy-efficient software by contract negotiation. In *Proceedings of the 6th International Workshop Models@run.time (MRT 2011)*, 2011.
- [15] G. Guizzardi. *Ontological foundations for structural conceptual models*. PhD thesis, Enschede, 2005.
- [16] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology, March-April 2008, ETH Zurich*, 7(3):125–151, 2008.
- [17] M. U. Khan. *Unanticipated Dynamic Adaptation of Mobile Applications*. PhD thesis, University of Kassel, March 2010.
- [18] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, Berlin/Heidelberg, 1997. Springer Verlag.
- [19] B. Morin, O. Barais, G. Nain, and J.-M. Jezequel. Taming dynamically adaptive systems using models and aspects. In *Proceedings of the 31st International Conference on Software Engineering, ICSE '09*, pages 122–132, Washington, DC, USA, 2009. IEEE Computer Society.
- [20] C. Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, Oct. 2003.
- [21] T. Reenskaug, P. Wold, and O. A. Lehne. *Working with objects - the OOram software engineering method*. Manning, 1996.
- [22] D. Riehle. Describing and Composing Patterns Using Role Diagrams. In *Proceedings of the 1996 Ubilab Conference*, Zurich, 1996.
- [23] D. Riehle and T. Gross. Role model based framework design and integration. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA '98*, pages 117–133, New York, NY, USA, 1998. ACM.
- [24] R. O. Rossen and R. Rashev. Adaptability and Adaptivity in Learning Systems, 1997.
- [25] Y. Smaragdakis and D. Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.*, 11:215–255, April 2002.
- [26] F. Steimann. On the representation of roles in object-oriented and conceptual modelling. *Data Knowl. Eng.*, 35(1):83–106, 2000.
- [27] F. Steimann. Role = Interface: a merger of concepts. *Journal of Object-Oriented Programming*, pages 23–32, 2001.
- [28] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.