DAiSI—Dynamic Adaptive System Infrastructure: Component Model and Decentralized Configuration Mechanism

Holger Klus
ROSEN Technology & Research Center GmbH
Lingen (Ems), Germany
hklus@rosen-group.com

Andreas Rausch, Dirk Herrling
Technische Universität Clausthal
Clausthal-Zellerfeld, Germany
{andreas.rausch, dirk.herrling}@tu-clausthal.de

Abstract— **Dynamic adaptive systems are systems that change their behavior according to the needs of the user at run time, based on context information. Since it is not feasible to develop these systems from scratch every time, a component model enabling dynamic adaptive systems is called for. Moreover, an infrastructure is required that is capable of wiring dynamic adaptive systems from a set of components in order to provide a dynamic and adaptive behavior to the user. In this paper we present just such an infrastructure or framework—called Dynamic Adaptive System Infrastructure (DAiSI). Because DAiSI has been developed for a number of years, we will cover as well the history of DAiSI as the newest advances. We will present an example illustrating the adaptation capabilities of the framework we introduce. The focus of the paper is on the underlying component model of DAiSI and the decentralized configuration mechanism.**

Keywords— dynamic adaptive systems; component model; component composition; adaptation; componentware; component container; decentralized configuration.

I. INTRODUCTION

This paper is an extended version of a paper presented at the Sixth International Conference on Adaptive and Self-Adaptive Systems and Applications [1].

Software-based systems pervade our daily life—at work as well as at home. Public administration or enterprise organizations can scarcely be managed without software-based systems. We come across devices executing software in nearly every household. The continuous increase in size and functionality of software systems has made some of them among the most complex man-made systems ever devised [2].

In the last two decades the trend towards “everything, every time, everywhere” has been dramatically increased through a) smaller mobile devices with higher computation and communication capabilities, b) ubiquitous availability of the Internet (almost all devices are connected with the Internet and thereby connected with each other), and c) devices equipped with more and more connected, intelligent and sophisticated sensors and actuators.

Nowadays, these devices are increasingly used within an organically grown, heterogeneous, and dynamic IT environment. Users expect them not only to provide their primary services but also to collaborate autonomously with each other and thus to provide real added value. The challenge is therefore to provide software systems that are robust in the presence of increasing challenges such as change and complexity [3].

The reasons for the steady increase in complexity are twofold: On the one hand, the set of requirements imposed on software systems is becoming larger and larger as the extrinsic complexity increases, in the form of, for example, additional functionality and variability. In addition, the structures of software systems—in terms of size, scope, distribution and networking of the system among other things—are themselves becoming more complex, which leads to an increase in the intrinsic complexity of the system.

Change is inherent, both in the changing needs of users and in the changes, which take place in the operational environment of the system. Hence, it is essential that our systems are able to adapt to maintain the satisfaction of the user expectations and environmental changes in terms of an evolutionary change. Dynamic change, in contrast to evolutionary change, occurs while the system is operational. Dynamic change requires that the system adapts at run time.

Since the complexity and change may not permit human intervention, we must plan for automated management of adaptation. The systems themselves must be capable of determining what system change is required, and in initiating and managing the change process wherever possible. This is the aim of self-managed systems.

Self-managed systems are those capable of adapting to the current context as required though self-configuration, self-healing, self-monitoring, self-tuning, and so on. These are also referred to as self-x, autonomic systems. Additionally, new components may enter or leave the system at run time. We call those systems ‘dynamic adaptive’.

Providing dynamic adaptive systems is a great challenge in software engineering [3]. In order to provide dynamic adaptive systems, the activities of classical development approaches have to be partially or completely moved from development time to run time. For instance, devices and software components can be attached to a dynamic adaptive system at any time. Consequently, devices and software components can be removed from the dynamic adaptive system or they can fail as the result of a defect. Hence, for dynamic adaptive systems, system integration takes place during run time.

To support the development of dynamic adaptive systems a couple of infrastructures and frameworks have been developed, as discussed in a related work section, Section II. In our research group we have also developed a framework for dynamic adaptive (and distributed) systems, called DAiSI.

We believe that service oriented, component based, dynamic adaptive systems need to address at least three kinds of adaptation:

1. Component service implementation adaptation: The implementation of a service is changed within the component that provides it. Thus, only the output of the component is affected but not the overall structure of the application. A simple example is the sorting order of search results that can be modified by the end user.
2. Component service usage adaptation: A component that uses a service of another component switches the service provider, i.e., another component that provides the same service. This can happen, for example, because the service quality of the now used service is superior compared to the formerly used one. Component service usage adaptation may yield to a better service quality of the component switching services. Only the directly involved components are affected by the change in system configuration.
3. System configuration adaptation: If the provided services of a component change, these changes can cascade through the complete system because optional dependencies could be resolved or mandatory dependencies can no longer be resolved. We speak of system configuration adaptation in such cases.

The development of the DAiSI was always motivated through running application examples and demonstrators. Based on the evaluation results a couple of drawbacks were identified. I) DAiSI's component model was not able to handle service cardinalities, such as exclusive and shared use of a specific service or service reference sets. Most of the applications realized needed service cardinalities. Due to the absence of service cardinalities we had to create workarounds. II) DAiSI's dynamic configuration mechanism was realized as a centralized component. The centralized configuration component was easy to implement but obviously it turned out to be a bottleneck.

For those reasons we have developed and implemented an improved version of the DAiSI framework. It contains a sophisticated component model including service cardinalities and a decentralized system configuration mechanism. In this paper, the new version of the DAiSI framework will be presented.

The rest of the paper is structured as follows: After a short description of the related work we provide an overview of the DAiSI framework in Section III. DAiSI's main essential, a domain model, an adaptive component model, and a decentralized dynamic configuration mechanism are introduced in this section. Then we describe a small sample application to illustrate the decentralized dynamic configuration mechanism of the adaptive components in Section IV. A short conclusion will round the paper up.

II. Related Work

Component-based software development, component models and component frameworks provide a solid approach to support evolutionary changes to systems. It is a well understood method that proved useful in numerous applications. Components are the units of deployment and integration. This allows high flexibility and easy maintenance. During design time components may be added or removed from a system [4].

However, the early component models did not provide means of adding or removing components from a running system. Also, the integration of new interaction links (e.g., component bindings) was not possible. Service-oriented approaches stepped up to the challenge. These systems usually maintain a service repository, in which every component that enters the system is registered. A component that wants to use such a component can query the service register for a matching service and connect to it, if one is found. For the domain of dynamic systems this means that a component can register its provided and required services. If a suitable service provider for one of the required services registers itself, it can be bound to satisfy the required service [5].

Service-oriented approaches have the inconvenient characteristic of not dealing with the adaptability of components. A component developer is solely responsible for the implementation of the adaptive behavior. This starts at the application logic and stretches to the discovery of unresponsive services, the discovery of newly available service, the discovery of services with a better quality of service, and so on. A couple of frameworks have been developed to support dynamic adaptive behavior, while, at the same time, making it easier for the developer to focus on implementing the behavioral changes in his component.

One of the first frameworks to support dynamic adaptive components was CONIC. It defines a description language that can be used to change the structure of modules of a running system. It allows the termination of running and the instantiation of new modules and the creation or deletion of links between them. CONIC was controlled through a centralized management console where the procedures for the reconfiguration could be entered [6].

REX is another framework for the support of dynamic-adaptive systems. It used the experience gained in the research for CONIC and aimed at dynamic adaptive, parallel, distributed systems. The concept was that such systems consist of components that are linked by interfaces. A new interface description language was invented, to be able to describe the interfaces. Components were seen as types, allowing multiple instances of every component to be present at run-time. Just like CONIC, REX allowed the creation and termination of component instances and the links between them. Both, CONIC and REX share the disadvantage that they support dynamic reconfiguration only through explicit reconfiguration programs. These need to be different for every situation that is detected and intended.

The approach moves the adaptation logic out of the component, but nevertheless, the developer has to deal with the adaptation strategy for every possible occurring change [7], [8].

Current frameworks such as ProAdapt [9] and Config.NETServices [10] have a more generic adaption and configuration mechanism. Components that were not known during the design-time of the system can be added or removed from the dynamic adaptive system during run-time. Therefore, a generic component configuration mechanism is provided by the framework. As with our first version of the DAiSI framework, these frameworks are based on a centralized configuration mechanism. Moreover, the underlying component model is restricted—for instance the exclusive usage of services cannot be described.

These are the two main issues we address with this paper. On the one hand, a self-organizing infrastructure is needed to remove the centralized configuration service as a single point of failure. On the other hand, service cardinalities are called for. Many service should only be used exclusively (e.g., security relevant services) or might only be used by a certain number of service consumers (e.g., a component that is running on a node with limited computing power). In the following section we will therefore present the DAiSI approach that addresses these issues.

III. DAiSI – Dynamic Adaptive System Infrastructure

Our approach for self-organizing systems is based on a specific framework called DAiSI [11], [12], [13], [14]. DAiSI consists of three main parts or elements: a domain model, an adaptive component model, and a decentralized dynamic configuration mechanism. In the following section, we will cover the history of the DAiSI approach, which gives an overview of the underlying concepts and realized industry projects and prototypes.

A. History of the DAiSI Approach

The research towards DAiSI started in 2004 [15] and the first version was implemented and published in 2006/07 [11], [12], [13], [14]. Based on the DAiSI framework, a couple of dynamic adaptive systems (research and industrial demonstrators) were developed and evaluated. Some of them were developed into successful business applications, for example [16] and [17]. The demonstrators that have been built were summarized in [18] and are briefly sketched in the following paragraphs:

Assisted Bicycle Training: In 2005, we proposed an ambient intelligence system for the training of a cycling group [19]. The individual training of one cyclist in the group is optimized based on the readings of numerous sensors, which evaluate his physical condition. Based on this information and the physical condition of the other cyclists in the group an optimal position for every cyclist is calculated, which has an influence on the training mainly because of the slipstream. We published the results of the research regarding the simulation of a cycling group in [20]. DAiSI was used in this scenario to connect the cyclist among each other, as well as the cyclists with a team cycling trainer, if he belongs to the same team. The resulting demonstrator has been exhibited at the CeBIT fair in 2005.

Assisted Living: In the assisted living scenario the focus lay on the monitoring of elderly people – more specific on their food and beverage consumption and their overall health status. The research started in 2005 under the assumption that our society is aging continuously and the expenses for health care are steadily increasing. On the other hand, more and more people prefer to stay in their known environment when they are aging. To address this issue, we proposed an apartment equipped with a multitude of sensors and intelligent devices (e.g., a fridge that monitors its contents). All these devices monitor and evaluate the state of the elderly person (e.g., vital data, did the person fall, did he/she drink enough, etc.) and the apartment (e.g., if the food in the refrigerator is still edible or already spoilt). DAiSI was used in this scenario to connect all sensors and devices so that new components can be installed and removed at run time [21], [12], [22].

Assisted Cross Country Skiing: In 2008, at the CeBIT fair, the Sport Information System (SiS) was exhibited. It was targeted at the cross country skiing domain. The system allowed a skiing trainer to analyze the skiing technique of one or more cross country skiers. If no radio connection between the trainer and the trainee could be established, the DAiSI configured the system in a way that the trainee got feedback regarding his technique based on an automated analysis of the movement of his skiing sticks. The analysis was possible because the skiing sticks were equipped with sensor nodes and the trainee carried a personal digital assistant (PDA) with himself.

Emergency Management System: In 2009, the Emergency Management System was exhibited at the CeBIT fair. Its goal was to support rescue workers in a mass casualty incident, also known as major incident. In these incidents, the rescue workers are usually outnumbered by the amount of casualties. To be able to deal best with such incidents, a good overview of the incident site and the casualties is mandatory. The casualties' treatment needs to be prioritized to save as many lives as possible. The emergency management system includes sensor nodes for every casualty and tablet-like computers for medics. These devices allow the computation of an interactive map that displays all rescue workers and casualties. Additional special hardware allows the monitoring of vital data of casualties and therefore the automatic suggestion of treatment priorities [23], [24], [25], [26]. The system is highly dynamic, because casualties and medic are continuously entering or leaving the system.

SmartSchank: In 2008, the project launched under the name "HomeS". It was designed to reduce the loss of beverage in the gastronomy through drawn, but not sold drinks. As hardware components, the system featured intelligent beer taps and registers with a credit/debit system. One requirement was the installation of the system without the need to manually configure it. Additionally, it was supposed to work in small pubs as well as in large arenas. DAiSI was used as a conceptual platform that could fulfill these requirements. In 2010, the prototype was exhibited at the CeBIT fair [16]. It was later evolved to a commercial product by the project partner [17].

Smart City and Smart Airport: In the context of the NTH Focused Research School for IT Ecosystems a demonstrator for a smart airport as an example for an IT ecosystem was built. All users within the smart airport were supposedly represented by devices called SmartFolks. These devices also served as an interface to the ecosystem in the airport. Two sub projects in this project were the Smart CheckIn and the RuleIT methodology. While the first scenario enables travelers to choose if they would rather pay less, but wait longer in a line for a standard check in, or if they would prefer to pay a little bit extra and get a guaranteed time slot to check in without needing to wait at all, the latter scenario focused on rule based application configuration and introduced user decisions into the application configuration process [14], [27], [28], [29].

Pac-Man: In the OPEN project (in 2010), the migration of an application at run time was researched. The focus lay on preserving the internal state of the application so that the end user can migrate his application from one device to the other. Additionally, the graphical user interface was supposed to adjust to the device the application was migrated to. The classic game Pac-Man was chosen as an application example, while DAiSI served as the underlying system infrastructure [30], [31].

Biathlon Training: This is the sample application that is used in this publication. The details will be introduced in Section IV.

We found that the three basic adaptation requirements that have been mentioned above, as well as some additional features can best be realized by a number of different architectural concepts:

Component Model: DAiSI was invented to support the development of service oriented, component based systems. Components communicate with each other through provided and required services. A subset of the required services can enable the component to provide a subset of the provided services. This relation is expressed by the so-called component configuration. The component model features all these elements since early development stages and will be further explained throughout the rest of this paper.

Configuration Service: The configuration service composes the application at run time of the present DAiSI components, by binding required and provided services to each other and configuring every component.

Registry Service: The configuration service requires knowledge of the present component in order to configure the system. The registry service maintains a database of all installed DAiSI components and the nodes they are deployed on. It monitors their responsiveness to discover and remove unresponsive components from the system.

Device Bay: Small scale devices do not always have the necessary computing power or memory to be able to participate directly in a DAiSI application. The device bay concept enables nodes with a higher computing power to represent those devices using an adapter pattern.

Dependability: In early DAiSI implementations provided services could satisfy required services if the interfaces describing the services were syntactically compatible. Further research enabled the requirement of semantic compatibility that can be ensured based on run time equivalence class testing.

Migrateability: Moving a running DAiSI component from one node to another was a requirement the DAiSI needed to satisfy for a particular research project. Through an extension of the component model, which allows the extraction and insertion of a consistent internal state, this requirement can be met. The migration is realized in three steps. At first the internal state of the component is extracted. In the second step, the component is stopped on the old node and started on the new node. Before it is activated, the internal state is inserted into the freshly started component in a third step.

Self-Organization: In early versions of the DAiSI, the system was configured by the configuration service, which followed an optimization algorithm and configured one component after the other. With large scale applications the configuration through one central configuration service becomes a bottleneck. Therefore, the configuration logic was moved into the individual components and they became self-organizing components. Details to the self-organization capabilities of the DAiSI can be found in Sections III and IV.

Architecture Awareness: Although syntactic and semantic compatibility already limit, which components can use which services of other components, this is often not enough. Dependent on the application domain, additional architectural rules may need to be enforced. For example, should a cross country skier only be linked to two skiing sticks; both need to belong to him. Just ensuring that he is connected to two skiing sticks does not fulfill the requirements of the end user. How the self-organizing components ensure that the application architecture requirements are met can be found in Sections III and IV.

Component Market: In conflicting cases where two or more components would like to use a particular service that cannot be used by all of them, a decision needs to be made. A component market solves this issue by introducing a currency into the system. Components that want to use service of a different component can use their currency to bid for a service. If they are chosen, they transfer their currency, and the component providing the service has both, the newly earned and its default currency. It can use that to bid for other services to improve its service quality. The underlying idea is that overall system quality improves if a component that has its provided services used by many other components improves its service quality.

User Decisions: The composition of an application out of the available components follows either an optimization goal (e.g., build the application that integrates the most components), or a set of rules. In different scenarios this can lead to more than one possible solution, which are to be considered as of the same quality. In these cases the end user can be involved to decide, which way the application should be configured.

Numerous publications discuss these architectural concepts with regard to a specific demonstrator or industry application. Table I shows a matrix with the different concepts in the top row and the demonstrators in the first column. The bibliography entries in the intersections indicate, which publication explains which concepts with the help of which demonstrators. In cases where a concept was used for a particular demonstrator but no results have been published, a "yes" is written in the matrix to state the fact.

TABLE I. MATRIX MAPPING ARCHITECTURAL CONCEPTS AND DEMONSTRATORS TO PUBLICATIONS

	Component Model	**Configuration Service**	**Registry Service**	**Device Bay**	**Dependability**	**Migrateability**	**Self-Organization**	**Architecture Awareness**	**Component Market**	**User Decisions**
Assisted Bicycle Training	[19]	[19]		[19]						
Assisted Living	[32] [12] [13] [33]	[32] [12] [13]	[13] [22]	[32] [12] [13]						
Assisted Cross Country Skiing	yes	yes	yes	yes						
Emergency Management System	[25] [35] [36] [26]	[25] [35] [36] [26] [34]			[25] [35] [36] [26]				[34]	
SmartSchank	yes	yes	yes							
Smart City / Smart Airport		[28] [29]								[29] [37]
Pac-Man	[38]					[38] [31]				
Biathlon Training	[39] [1]	[39] [1]					[39] [1]	[39] [1]		
None of the above	[11]	[11]	[11]							

B. *Domain Model*

The three elements of the DAiSI framework – the domain model, an adaptive component model, and a decentralized dynamic configuration mechanism will be introduced in this section. The three elements and their relationship to each other are depicted in Figure 1 using a UML class diagram. Note, a complete description of the DAiSI framework can be found in [39].

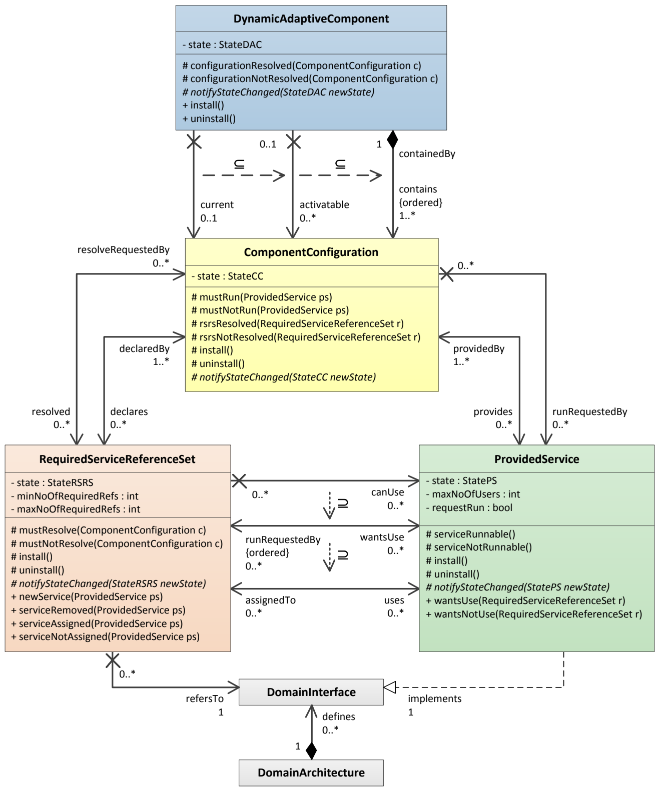

DynamicAdaptiveComponent
- state : StateDAC
# configurationResolved(ComponentConfiguration c)
# configurationNotResolved(ComponentConfiguration c)
# notifyStateChanged(StateDAC newState)
+ install()
+ uninstall()
0..1
1
containedBy
⊆
⊆
contains {ordered} 1..*
current 0..1
activatable 0..*
ComponentConfiguration
- state : StateCC
# mustRun(ProvidedService ps)
# mustNotRun(ProvidedService ps)
# rsrsResolved(RequiredServiceReferenceSet r)
# rsrsNotResolved(RequiredServiceReferenceSet r)
# install()
# uninstall()
# notifyStateChanged(StateCC newState)
resolveRequestedBy 0..*
0..*
declaredBy 1..*
providedBy 1..*
resolved 0..*
declares 0..*
provides 0..*
runRequestedBy 0..*
RequiredServiceReferenceSet
- state : StateRSRS
- minNoOfRequiredRefs : int
- maxNoOfRequiredRefs : int
# mustResolve(ComponentConfiguration c)
# mustNotResolve(ComponentConfiguration c)
# install()
# uninstall()
# notifyStateChanged(StateRSRS newState)
+ newService(ProvidedService ps)
+ serviceRemoved(ProvidedService ps)
+ serviceAssigned(ProvidedService ps)
+ serviceNotAssigned(ProvidedService ps)
ProvidedService
- state : StatePS
- maxNoOfUsers : int
- requestRun : bool
# serviceRunnable()
# serviceNotRunnable()
# install()
# uninstall()
# notifyStateChanged(StatePS newState)
+ wantsUse(RequiredServiceReferenceSet r)
+ wantsNotUse(RequiredServiceReferenceSet r)
0..*
canUse 0..*
⊇
runRequestedBy {ordered} 0..*
wantsUse 0..*
⊇
assignedTo 0..*
uses 0..*
0..*
refersTo 1
DomainInterface
implements 1
defines 0..*
1
DomainArchitecture

Figure 1. Core elements of the DAiSI framework.

As in other domains, such as the network domain, physical connectors (like the RJ 45 connector) and their pin configurations are standard and well known by all component vendors. A similar situation can be found in the operating system domain: The interface for printer drivers is standardized and published by the operating system vendor. Third-party printer vendors adhere to this interface specification to create printer drivers that are plugged into the operating system during run time.

The same principle is used in the DAiSI framework: The domain model contains standardized and broadly accepted interfaces in the domain. The domain model defines the basic notions and concepts of the domain shared by all components. This means the domain model provides the foundation for the dynamic configuration of the adaptive system and the available components.

The domain model, as shown in Figure 1, consists of the *DomainInterface* and *DomainArchitecture* classes. The domain model itself is represented by an instance of the *DomainArchitecture* class. A domain model contains a set of domain interfaces, represented by an instance of the class *DomainInterface*.

Domain interfaces contain syntactical information like method signatures or datatypes occuring in the interfaces. In addition they may also contain a behavioral specification of the interface following the design by contract approach, for instance using pre- and postconditions and invariants to describe the functional behavior of a domain interface [25].

Usually, components need services from other components to provide their own service within the dynamic adaptive system. To indicate, which services a component provides and requires it refers to the corresponding *DomainInterface*. As components providing services and components requiring services refer to the same domain interface description DAiSI is able to identify those and bind these components together during run time.

Using simple domain interface descriptions the correctness of the binding can only be guaranteed on a syntactical level. Once the domain interface descriptions contain additional information about the functional behavior, the correctness of the binding can also be guaranteed on the behavioral level. Therefore, we have developed a sophisticated approach based on run-time testing. Further information of DAiSI's solution to guarantee functional correctness of dynamic adaptive systems during run time can be found in [25], [26].

C. *Adaptive Component Model*

Each component in the system is represented by the *DynamicAdaptiveComponent* class. Each component may provide services to other components or use services, provided by other components. The services a component provides are represented by the *ProvidedService* class. The services a component requires are specified by the *RequiredServiceReferenceSet* class, where each instance represents a set of required services for exactly one domain interface. The *ComponentConfiguration* class of the component model represents a mapping between services required and provided. If all the required services of a component configuration are available, the provided services of that component configuration can in turn be provided to other components. In the following subsections the individual parts of the component model are introduced in more detail. Afterwards, the interplay of these parts during the configuration process will be explained.

1) *Dynamic Adaptive components*

Each component instance within the system is represented by an instance of the class *DynamicAdaptiveComponent*, see Figure 2.

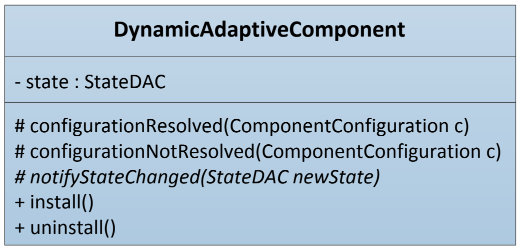

DynamicAdaptiveComponent
- state : StateDAC
# configurationResolved(ComponentConfiguration c)
# configurationNotResolved(ComponentConfiguration c)
# notifyStateChanged(StateDAC newState)
+ install()
+ uninstall()

Figure 2. DynamicAdaptiveComponent class.

By calling the install or uninstall methods, a component is, respectively, published or removed from the system. If install is called, all other parts of that component are informed by calling the trigger install. The framework then starts trying to resolve dependencies on other components in order to run ProvidedServices and provide them to other components within the system. Each *DynamicAdaptive-Component* realizes a state machine, as shown in Figure 3 whose current state is stored in a variable called state.

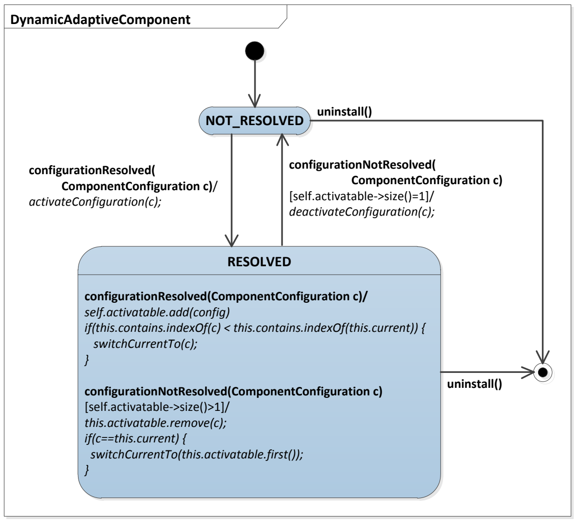

DynamicAdaptiveComponent
NOT_RESOLVED
uninstall()
configurationResolved(ComponentConfiguration c)/ activateConfiguration(c);
configurationNotResolved(ComponentConfiguration c) [self.activatable->size()=1]/ deactivateConfiguration(c);
RESOLVED
configurationResolved(ComponentConfiguration c)/
self.activatable.add(config)
if(this.contains.indexOf(c) < this.contains.indexOf(this.current)) {
switchCurrentTo(c);
}
configurationNotResolved(ComponentConfiguration c)
[self.activatable->size()>1]/
this.activatable.remove(c);
if(c==this.current) {
switchCurrentTo(this.activatable.first());
}
uninstall()

Figure 3. State machine - DynamicAdaptiveComponent class.

Two states are distinguished for *DynamicAdaptive-Component*, namely RESOLVED and NOT_RESOLVED. In the beginning a component is in the NOT_RESOLVED state. If, for a single *ComponentConfiguration*, all dependencies to services of other components are resolved, the trigger *configurationResolved* of *DynamicAdaptive-Component* is called and the state machine switches to state RESOLVED. Every time a state transition takes place, the abstract method, *notifyStateChanged*, is called. A component developer can override this method in order to react to certain state transitions, e.g., by showing or fading out a graphical user interface.

2) *Component Configuration*

Each component defines at least one Component-Configuration.

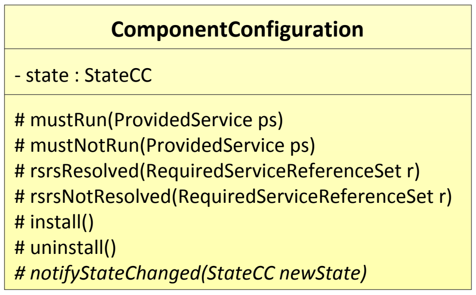

ComponentConfiguration
- state : StateCC
# mustRun(ProvidedService ps)
# mustNotRun(ProvidedService ps)
# rsrsResolved(RequiredServiceReferenceSet r)
# rsrsNotResolved(RequiredServiceReferenceSet r)
# install()
# uninstall()
# notifyStateChanged(StateCC newState)

Figure 4. ComponentConfiguration class.

Figure 4 shows the corresponding class diagram for *ComponentConfiguration*. The defined *ComponentConfigu-rations* are connected to a component by the association *contains*. Each *ComponentConfiguration* represents a mapping between a set of required and provided services. If all services required by a *ComponentConfiguration* are available, the corresponding provided services can be provided to other components. That configuration is then marked as *activatable*. In case a component has more than one *ComponentConfiguration*, an order must be defined by the component developer. During run time, at most one *ComponentConfiguration* can be active. That one is then marked as current and only those provided services are executed that are connected to *ComponentConfiguration*, which is marked as current.

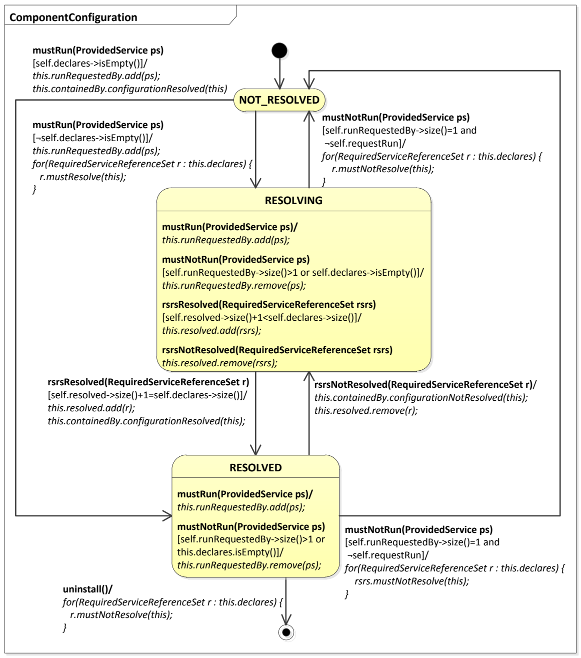

ComponentConfiguration
mustRun(ProvidedService ps)
[self.declares->isEmpty()]/
this.runRequestedBy.add(ps);
this.containedBy.configurationResolved(this)
NOT_RESOLVED
mustRun(ProvidedService ps)
[¬self.declares->isEmpty()]/
this.runRequestedBy.add(ps);
for(RequiredServiceReferenceSet r : this.declares) {
r.mustResolve(this);
}
mustNotRun(ProvidedService ps)
[self.runRequestedBy->size()=1 and
¬self.requestRun]/
for(RequiredServiceReferenceSet r : this.declares) {
r.mustNotResolve(this);
}
RESOLVING
mustRun(ProvidedService ps)/
this.runRequestedBy.add(ps);
mustNotRun(ProvidedService ps)
[self.runRequestedBy->size()>1 or self.declares->isEmpty()]/
this.runRequestedBy.remove(ps);
rsrsResolved(RequiredServiceReferenceSet rsrs)
[self.resolved->size()+1<self.declares->size()]/
this.resolved.add(rsrs);
rsrsNotResolved(RequiredServiceReferenceSet rsrs)
this.resolved.remove(rsrs);
rsrsResolved(RequiredServiceReferenceSet r)
[self.resolved->size()+1=self.declares->size()]/
this.resolved.add(r);
this.containedBy.configurationResolved(this);
rsrsNotResolved(RequiredServiceReferenceSet r)/
this.containedBy.configurationNotResolved(this);
this.resolved.remove(r);
RESOLVED
mustRun(ProvidedService ps)/
this.runRequestedBy.add(ps);
mustNotRun(ProvidedService ps)
[self.runRequestedBy->size()>1 or
this.declares.isEmpty()]/
this.runRequestedBy.remove(ps);
mustNotRun(ProvidedService ps)
[self.runRequestedBy->size()=1 and
¬self.requestRun]/
for(RequiredServiceReferenceSet r : this.declares) {
rsrs.mustNotResolve(this);
}
uninstall()/
for(RequiredServiceReferenceSet r : this.declares) {
r.mustNotResolve(this);
}

Figure 5. State machine - ComponentConfiguration class.

Each *ComponentConfiguration* realizes a state machine, as shown in Figure 5, with three states, namely NOT_RESOLVED, RESOLVING and RESOLVED. If a *ProvidedService* has to be executed (e.g., because another component needs it), the trigger mustRun of *ComponentConfiguration* is called. Afterwards the trigger *mustResolve* is called at each *RequiredServiceReferenceSet* in order to initiate the resolving of dependencies to other components. A *RequiredServiceReferenceSet* informs the *ComponentConfiguration* of the current status of the dependency resolution by calling the triggers *rsrsResolved* and *rsrsNotResolved*.

A *ComponentConfiguration* is in RESOLVED state if the dependencies of all required services are resolved, i.e., all connected *RequiredServiceReferenceSets* have called the trigger *rsrsResolved*. The *ComponentConfiguration* in turn calls *configurationResolved* to inform the *DynamicAdaptive-Component*.

3) Provided Service

A component's provided services are represented by the class *ProvidedService* shown in the class diagram in Figure 6. Each one implements exactly one domain interface. For each *ProvidedService* the number of service users who are allowed to use the service in parallel can be specified. This is done by setting the variable *maxNoOfUsers* to the required value. In our component model, a service is executed for only two reasons. The first reason is that there exist one or more components that want to use that service. Requests for service usage can be placed by calling the method *wantsUse*, or *wantsNotUse* if the usage request has become invalid. If there is a usage request for a *ProvidedService*, the connected *ComponentConfigurations* are informed by calling the trigger *mustRun*. The second reason that a service might have to be executed is that it provides some kind of direct benefit for end users. A component developer can set the flag *requestRun* in this case (e.g., because the service realizes a graphical user interface).

A *ProvidedService* realizes a state machine with three states namely NOT_RUNNING, RUNNABLE and RUNNING, as illustrated in Figure 7. A service is in RUNNABLE state if it is exclusively connected to *ComponentConfigurations* whose dependendies are resolved but none of them is marked as current. This is the case for a *ComponentConfiguration* that has higher priority and that is marked as *activatable*. However, a service is in RUNNING state if it is connected to a *ComponentConfiguration*, which is marked as *current*. If a *ComponentConfiguration* becomes current, all connected *ProvidedServices* are informed by calling the *serviceRunnable* trigger.

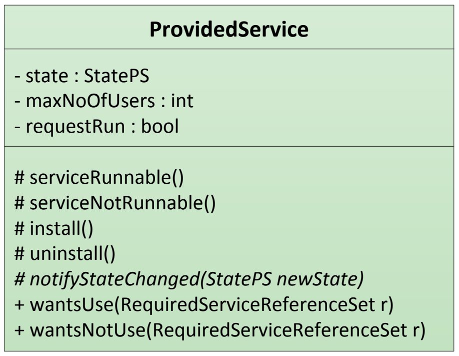


Figure 6. ProvidedService class.

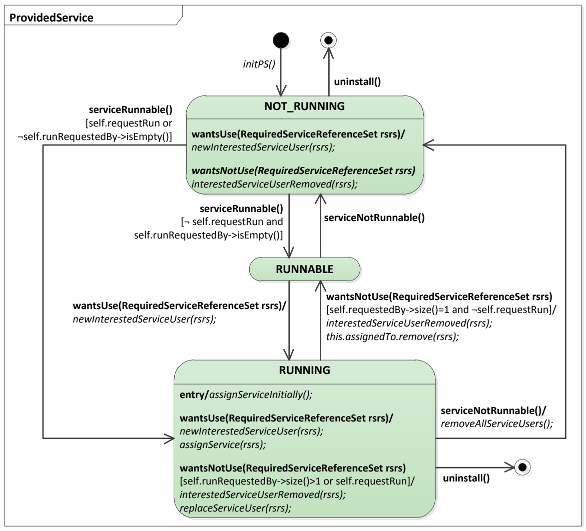

ProvidedService
initPS()
uninstall()
NOT_RUNNING
wantsUse(RequiredServiceReferenceSet rsrs)/ newInterestedServiceUser(rsrs);
wantsNotUse(RequiredServiceReferenceSet rsrs) interestedServiceUserRemoved(rsrs);
serviceRunnable() [self.requestRun or ¬self.runRequestedBy->isEmpty()]
serviceRunnable() [¬ self.requestRun and self.runRequestedBy->isEmpty()]
serviceNotRunnable()
RUNNABLE
wantsUse(RequiredServiceReferenceSet rsrs)/ newInterestedServiceUser(rsrs);
wantsNotUse(RequiredServiceReferenceSet rsrs) [self.requestedBy->size()=1 and ¬self.requestRun]/ interestedServiceUserRemoved(rsrs); this.assignedTo.remove(rsrs);
RUNNING
entry/assignServiceInitially();
wantsUse(RequiredServiceReferenceSet rsrs)/ newInterestedServiceUser(rsrs); assignService(rsrs);
wantsNotUse(RequiredServiceReferenceSet rsrs) [self.runRequestedBy->size()>1 or self.requestRun]/ interestedServiceUserRemoved(rsrs); replaceServiceUser(rsrs);
serviceNotRunnable()/ removeAllServiceUsers();
uninstall()

Figure 7. State machine - ProvidedService class.

4) *Required Service Reference Set*

A component may need functionality provided by other components in the system. In our component model those dependencies are specified with the *RequiredService-ReferenceSet* class, shown in Figure 8. Each instance of *RequiredServiceReferenceSet* represents dependencies on a set of services that implement the same domain interface. That domain interface is specified by the association *refersTo*. A component representing a trainer for example, may define a *RequiredServiceReferenceSet* that refers to a domain interface called *IAthlete* in order to get access to the training data of athletes. The minimum and maximum number of required references to services can be specified by setting the variables *minNoOfRequiredRefs* and *maxNoOf-RequiredRefs*.

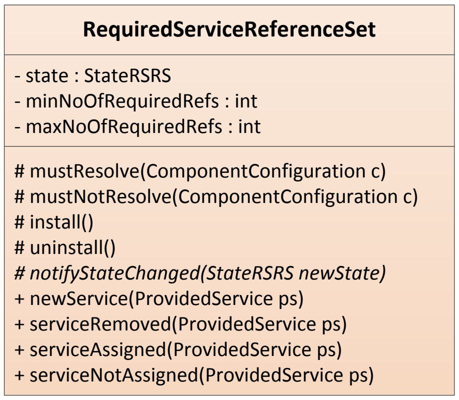

RequiredServiceReferenceSet
- state : StateRSRS
- minNoOfRequiredRefs : int
- maxNoOfRequiredRefs : int
# mustResolve(ComponentConfiguration c)
# mustNotResolve(ComponentConfiguration c)
# install()
# uninstall()
# notifyStateChanged(StateRSRS newState)
+ newService(ProvidedService ps)
+ serviceRemoved(ProvidedService ps)
+ serviceAssigned(ProvidedService ps)
+ serviceNotAssigned(ProvidedService ps)

Figure 8. RequiredServiceReferenceSet class.

A *RequiredServiceReferenceSet* realizes a state machine with three states, namely NOT_RESOLVED, RESOLVING and RESOLVED. Figure 9 visualizes this state machine. As soon as there is a request for resolving dependencies, the state switches to RESOLVED or RESOLVING, depending on the value of *minNoOfRequiredRefs*. If it is zero, then the requirements are fulfilled and it can switch directly to RESOLVED. A request for dependency resolution is placed by calling the *mustResolve* trigger.

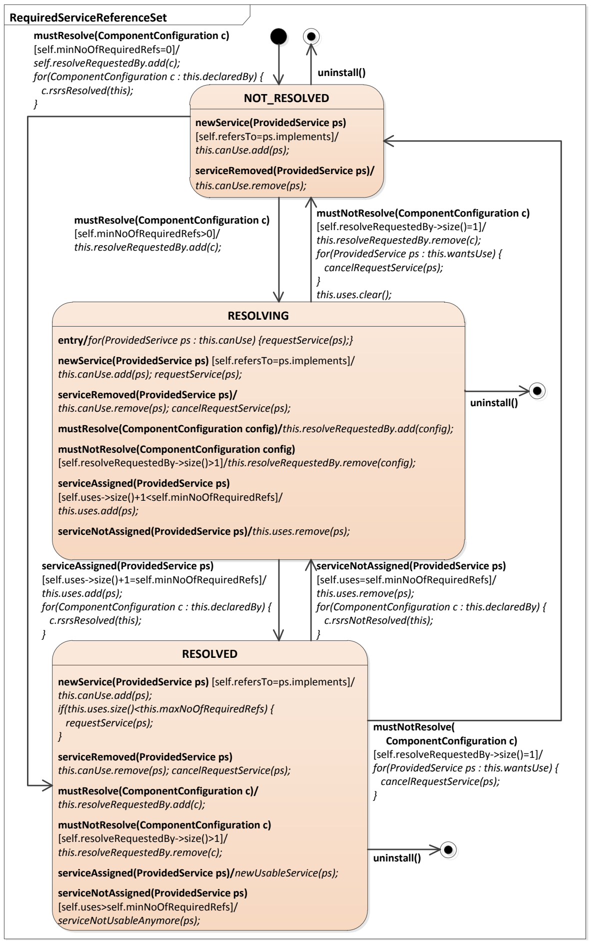

RequiredServiceReferenceSet
mustResolve(ComponentConfiguration c) [self.minNoOfRequiredRefs=0]/ self.resolveRequestedBy.add(c); for(ComponentConfiguration c : this.declaredBy) { c.rsrsResolved(this); }
uninstall()
NOT_RESOLVED
newService(ProvidedService ps) [self.refersTo=ps.implements]/ this.canUse.add(ps);
serviceRemoved(ProvidedService ps)/ this.canUse.remove(ps);
mustResolve(ComponentConfiguration c) [self.minNoOfRequiredRefs>0]/ this.resolveRequestedBy.add(c);
mustNotResolve(ComponentConfiguration c) [self.resolveRequestedBy->size()=1]/ this.resolveRequestedBy.remove(c); for(ProvidedService ps : this.wantsUse) { cancelRequestService(ps); } this.uses.clear();
RESOLVING
entry/for(ProvidedSerivce ps : this.canUse) {requestService(ps);}
newService(ProvidedService ps) [self.refersTo=ps.implements]/ this.canUse.add(ps); requestService(ps);
serviceRemoved(ProvidedService ps)/ this.canUse.remove(ps); cancelRequestService(ps);
mustResolve(ComponentConfiguration config)/this.resolveRequestedBy.add(config);
mustNotResolve(ComponentConfiguration config) [self.resolveRequestedBy->size()>1]/this.resolveRequestedBy.remove(config);
serviceAssigned(ProvidedService ps) [self.uses->size()+1<self.minNoOfRequiredRefs]/ this.uses.add(ps);
serviceNotAssigned(ProvidedService ps)/this.uses.remove(ps);
uninstall()
serviceAssigned(ProvidedService ps) [self.uses->size()+1=self.minNoOfRequiredRefs]/ this.uses.add(ps); for(ComponentConfiguration c : this.declaredBy) { c.rsrsResolved(this); }
serviceNotAssigned(ProvidedService ps) [self.uses=self.minNoOfRequiredRefs]/ this.uses.remove(ps); for(ComponentConfiguration c : this.declaredBy) { c.rsrsNotResolved(this); }
RESOLVED
newService(ProvidedService ps) [self.refersTo=ps.implements]/ this.canUse.add(ps); if(this.uses.size()<this.maxNoOfRequiredRefs) { requestService(ps); }
serviceRemoved(ProvidedService ps) this.canUse.remove(ps); cancelRequestService(ps);
mustResolve(ComponentConfiguration c)/ this.resolveRequestedBy.add(c);
mustNotResolve(ComponentConfiguration c) [self.resolveRequestedBy->size()>1]/ this.resolveRequestedBy.remove(c);
serviceAssigned(ProvidedService ps)/newUsableService(ps);
serviceNotAssigned(ProvidedService ps) [self.uses>self.minNoOfRequiredRefs]/ serviceNotUsableAnymore(ps);
mustNotResolve(ComponentConfiguration c) [self.resolveRequestedBy->size()=1]/ for(ProvidedService ps : this.wantsUse) { cancelRequestService(ps); }
uninstall()

Figure 9. State machine - RequiredServiceReferenceSet class.

5) *Notation for DAiSI Components*

To describe DAiSI components we use a compact notation, illustrated in Figure 10. Provided services are notated as circles, required services as semicircles, component configurations are depicted as crossbars, and the component itself is represented by a rectangle. Provided services that are intended to be activated (flag *requestRun* is true) are shown as a black circle.

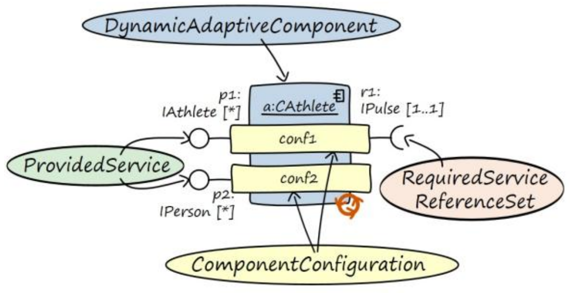


Figure 10. Notation for DAiSI components.

The component depicted in Figure 10 thus specifies two component configurations. The first requires exactly one service, which implements the *DomainInterface IPulse*. If such a service is available, the service variable p_1 of type *IAthlete* can in turn be provided to other components in the system. If no pulse service is available, the second configuration can still be activated because that one defines no dependencies to other services. In that case, the athlete component provides the service variable p_2 to other components.

D. Decentralized Dynamic Configuration Mechanism

There exist three types of relations between *RequiredServiceReferenceSets* and *ProvidedServices*, represented by the associations *canUse*, *wantsUse* and *uses*. The set of services that implement the domain interface referred by the *RequiredServiceReferenceSet* is represented by *canUse*. Note, this only guarantees a syntactically correct binding. In [25] and [26], we have shown how this approach can be extended to guarantee functional-behaviorally correct binding as well during run time using a run-time testing approach.

The *wantsUse* set holds references to those services for which a usage request has been placed by calling *wantsUse*. And the *uses* set contains references to those services, which are currently in use by the component or by *RequiredServiceReferenceSet*.

Each time a new service becomes available in the system, the newService method is called with a reference to the service as parameter. The new service is added to all *canUse* sets, if the corresponding *RequiredServiceReferenceSet* refers to the same *DomainInterface* as the *ProvidedServices*. If there is a request for dependency resolution (by a call of the *mustResolve* trigger), usage requests are placed at the services in *canUse* by calling *wantsUse* and those service references are copied to the *wantsUse* set. *ProvidedServices*

The management of these three associations—*canUse*, *wantsUse* and *uses*—between *RequiredServiceReferenceSets* and *ProvidedServices* is handled by DAiSI's decentralized dynamic configuration mechanism. This configuration mechanism relays on the state machines, presented in the previous sections, of the corresponding classes in the DAiSI framework and their interaction. In the following section, we will first describe the local configuration mechanism component and then the interaction between two components for inter-component configuration.

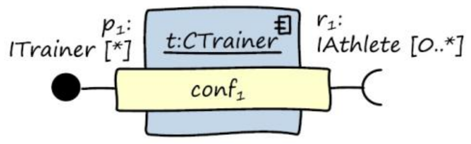


Figure 11. CTrainer component.

1) Local Configuration Mechanism

Assume a given component as shown in Figure 11. The component t of type *CTrainer* has a single configuration. It provides a service of type *ITrainer* to the environment, which can be used by an arbitrary number of other components. The component requires zero to any number of references to services of type *IAthlete*.

The boolean flag *requestRun* is true for the service provided. Hence, DAiSI has to run the component and provide the service within the dynamic adaptive system to other components and to users. As the component requires zero reference to services of type *IAthlete*, DAiSI can run the component directly and thereby provides the component service to other components and users as shown in the sequence diagram in Figure 12.

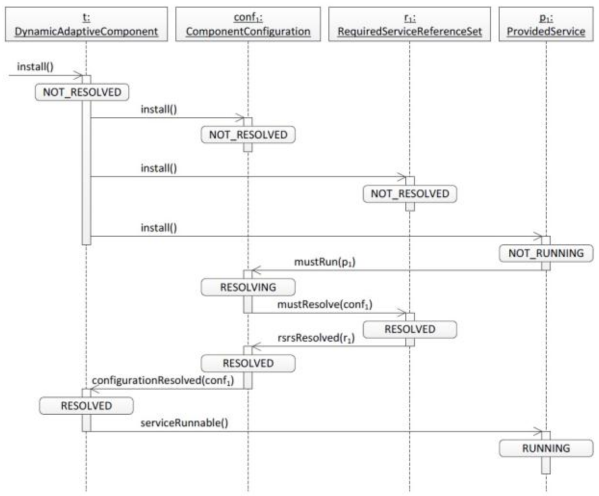


Figure 12. Local configuration mechanism component.

2) Inter-Component Configuration Mechanism

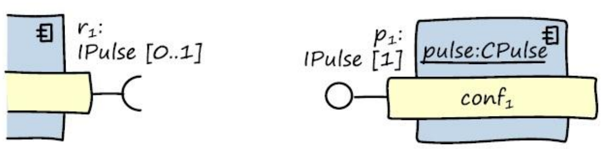


Figure 13. CAthlete and CPulse components.

Now assume two components: The *CAthlete* component, shown on the right hand side of Figure 13, requires zero or one reference to a service of type *IPulse*. The second component, *CPulse*, shown on the left hand side of Figure 13, provides a service of type *IPulse*. Note, this service can only be exclusively used by a single component.

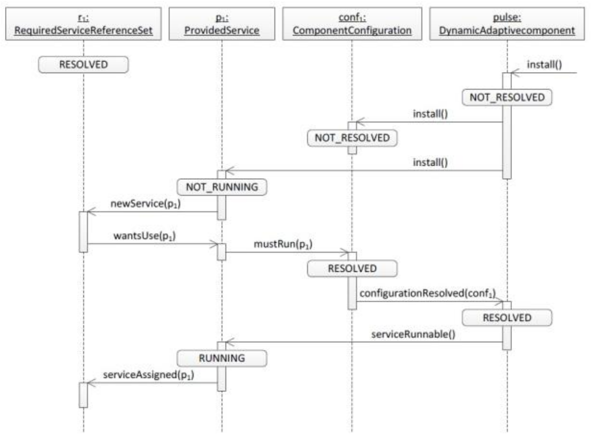


Figure 14. Inter-component configuration mechanism.

Once the *CPulse* component is installed or activated within the dynamic adaptive system, DAiSI integrates the new service in the canUse relationship of the *RequiredServiceReferenceSet* r_1 of the component *CAthlete*. Then DAiSI informs (calling the method *newService*) the *CAthlete* component that a new service that can be used is available as shown in Figure 14. DAiSI indicates that *CAthlete* wants to use this new service by adding this service in the set of services that *CAthlete* wants to use (set *wantsUse* of *CAthlete*). Once the service runs it is assigned to the *CAthlete* component, which can use the service from now on (added to the set uses of *CAthlete*).

IV. Sample Application – Smart Biathlon Training System

As already mentioned, we have realized and used a couple of dynamic adaptive systems based on DAiSI. One of the first domains for which we developed dynamic adaptive systems was training systems for athletes. For that reason we have chosen this domain to implement the first dynamic adaptive system on top of the new DAiSI version.

A. Domain Model

In the desired dynamic adaptive system, athletes (*IAthlete*) and trainers (*ITrainer*) can supervise the pulse (*IPulse*) of the athlete (see Figure 15). Moreover, athletes might use ski sticks (*IStick*), which have gyro sensors. Once connected with the sticks the athlete as well as the trainer can monitor the technically appropriate use of the sticks during skiing for the required skiing style. Once the biathlete has reached a shooting line (*IShootingLine*) he is allowed to use the shooting line only if a supervisor is available (*ISupervisor*).

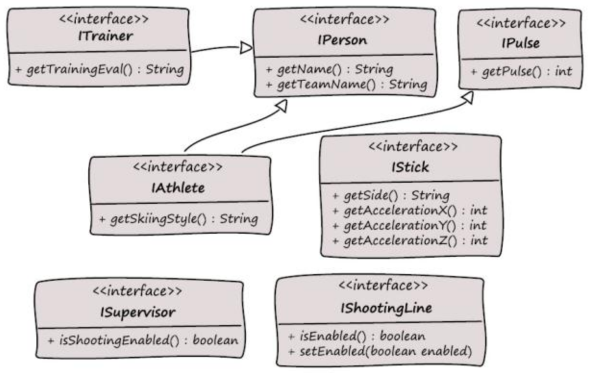

<<interface>> ITrainer
+ getTrainingEval() : String
<<interface>> IPerson
+ getName() : String
+ getTeamName() : String
<<interface>> IPulse
+ getPulse() : int
<<interface>> IAthlete
+ getSkiingStyle() : String
<<interface>> IStick
+ getSide() : String
+ getAccelerationX() : int
+ getAccelerationY() : int
+ getAccelerationZ() : int
<<interface>> ISupervisor
+ isShootingEnabled() : boolean
<<interface>> IShootingLine
+ isEnabled() : boolean
+ setEnabled(boolean enabled)

Figure 15. Domain model - "Smart Biathlon Training System".

B. Available Components

For a simple version of the system only three component types have been realized (see Figure 16): *CPulse*, *CAthlete*, and *CTrainer*. Note that additional components have been realized and evaluated for more sophisticated systems. For the purposes of this paper we only use these three components to show the decentralized configuration mechanism.

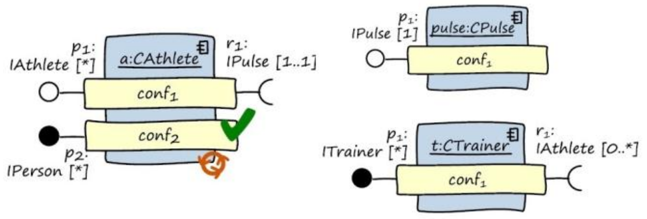


Figure 16. Adaptive components: CPulse, CAthlete, CTrainer.

The *CPulse* component provides an exclusive usable service *IPulse* and requires no other services from the dynamic adaptive system. The *CAthlete* component provides two services: *IPerson* and *IAthlete*. In *conf*$_2$ it provides the service, *IPerson*, which has the flag, *requestRun*, and requires no service from the environment. In conf1 it provides the service, *IAthlete*, but therefore requires a service, *IPulse*. And finally the *CTrainer* component may supervise an arbitrary number of athletes and thus provides a corresponding number of *ITrainer* interfaces to the real trainer, supporting him with the online training information of the supervised athletes.

C. Decentralized Dynamic Configuration Mechanism

Assume the following situation in the dynamic adaptive system. The component, *CPulse*, is activated and the component, *CAthlete*, is activated (see Figure 17). As the *requestRun* flag of the provided service of *conf*$_2$ is set and no additional service references are needed, this configuration is activated and the service is provided within the dynamic adaptive system.

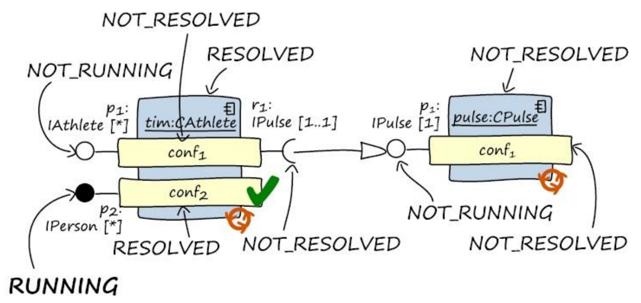


Figure 17. Initial situation in the Dynamic Adaptive System.

For the better configuration, *conf$_1$*, *CAthlete* requires a reference to a service of type *IPulse*. The *CPulse* component is able to provide this service. As the provided service, *IAthlete*, of configuration *conf$_1$* of component *CAthlete* is not requested by any other component and has not set the *requestRun* flag, this higher configuration is not activated.

Figures 18 to 29 show the following situation: A component, *CTrainer*, has been activated and integrated into our dynamic adaptive system.

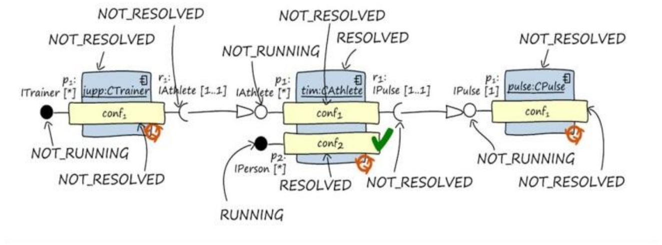


Figure 18. The CTrainer component “jupp” is deployed.

In the following the decentralized dynamic configuration mechanism is shown. Based on the interaction between the state machines of the adaptive components the dynamic adaptive system is reconfigured and the component is dynamically integrated into the system.

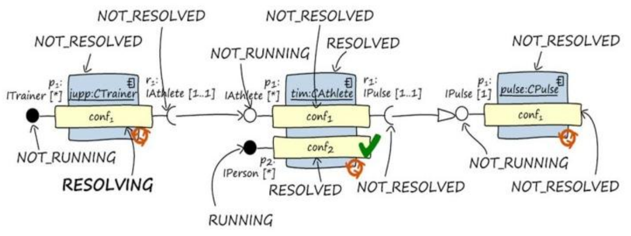


Figure 19. The component configuration conf$_1$ of Jupp’s CTrainer component switches its state to RESOLVING.

The configuration strategy is then as follows. Each service with *requestRun* flag set—in Figure 18 the new service *ITrainer* of the *CTrainer* component—resolves the required services transitively from the root to the leaf. Figures 18 to 23 show how the involved components are switched to the state RESOLVING.

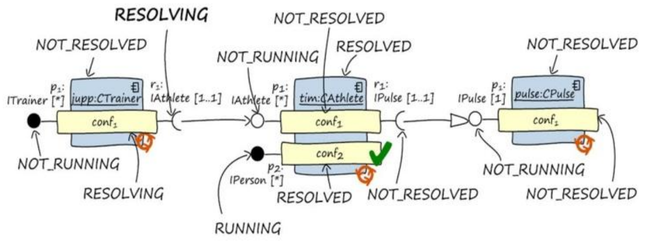


Figure 20. The interface IAthlete of Jupp’s CTrainer component switches its state to RESOLVING.

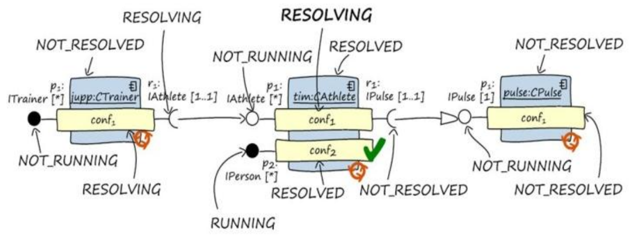


Figure 21. The component configuration conf$_1$ of Tim’s CAthlete component switches its state to RESOLVING.

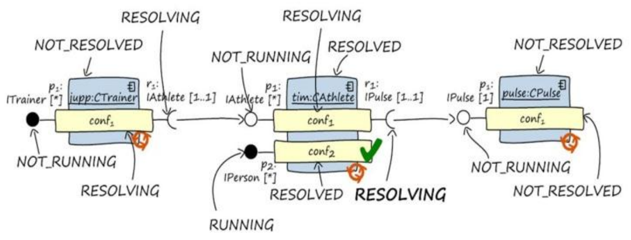


Figure 22. The interface IPulse of Tim’s CAthlete component switches its state to RESOLVING.

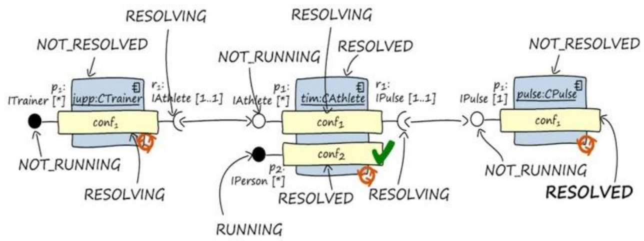


Figure 23. The component configuration conf$_1$ of the pulse component is marked as RESOLVED, because it has no required services.

Once all required services are resolved these services are activated (RUNNING) from the leaf to the root. This can be seen in Figures 24 to 29 of the application example. If not all required services were resolvable, the resolved services are set back to NOT_RESOLVED. This allows other services to resolve these services and frees the reserved resources.

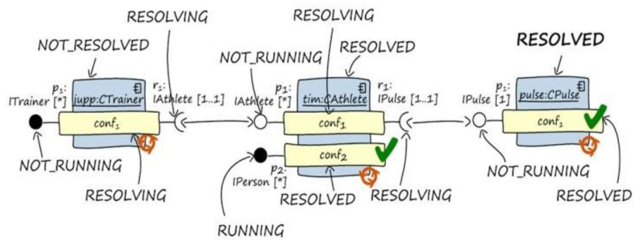


Figure 24. The pulse component is marked as RESOLVED.

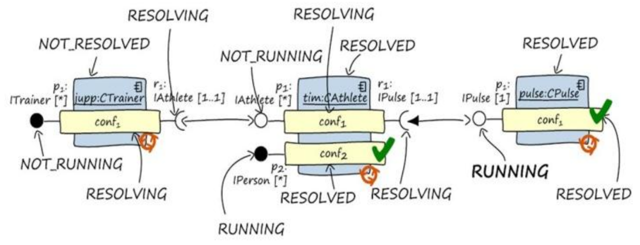


Figure 25. The IPulse interface is now RUNNING, because its requirements are resolved and a consumer (Tim's athlete component) is present.

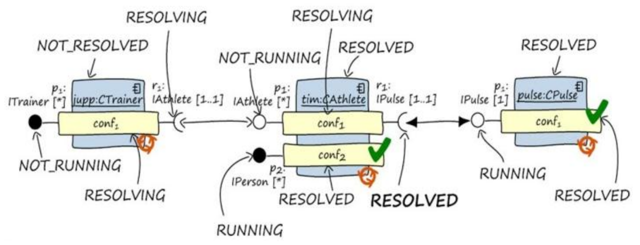


Figure 26. The required IPulse service of Tim's CAthlete component switches its state to RESOLVED.

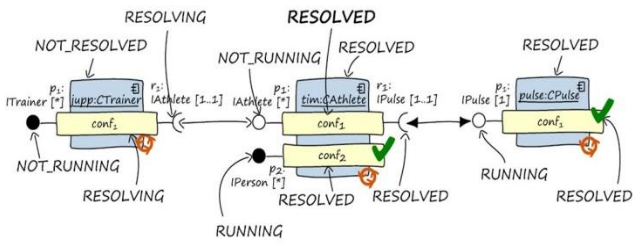


Figure 27. The component configuration $conf_1$ of Tim's CAthlete component is marked as RESOLVED.

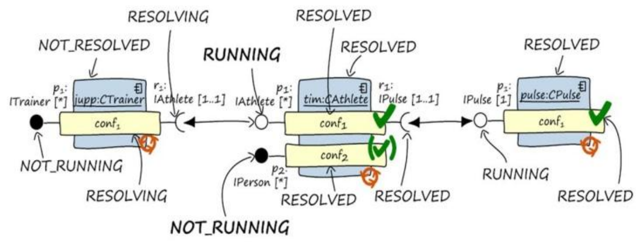


Figure 28. The provided IAthlete interface is marked as RUNNING, IPerson is now NOT_RUNNING, as the active component configuration changed from $conf_2$ to $conf_1$.

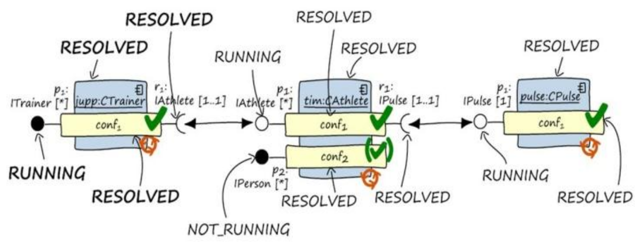


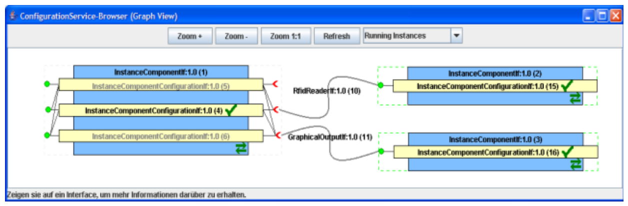
Figure 29. The configuration process is finished. Jupp's CTrainer component is now in the state RESOLVED, together with its component configurations and required services. Jupp's ITrainer interface is RUNNING.

V. Conclusion and Future Work

The DAiSI approach is that a developer does not have to implement a whole dynamic adaptive system on his own. Instead the developer can develop one or more components for a specific domain. This is only possible if a domain model is available as described. This domain model has to define the interfaces between the adaptive components of the dynamic adaptive system in the specific domain.

Based on this, the developer can develop even a single component and define which interfaces from the domain architecture are required or provided in the different configurations of this component. Moreover, one can develop mock-up components providing the required interfaces in order to test the new component during development.

To support the component development DAiSI comes with two implementation frameworks. These frameworks provide several helper classes enabling a quick implementation of dynamic adaptive systems in Java as well as in C++, concentrating on the functional features of the component to be developed. DAiSI-based dynamic adaptive systems can be distributed across various machines. DAiSI is also able to establish dynamic adaptive systems across language barriers—Java- and C++-based DAiSI components can be linked together through DAiSI to form a dynamic adaptive system.


ConfigurationService-Browser (Graph View)
Zoom +
Zoom -
Zoom 1:1
Refresh
Running Instances
InstanceComponentII:1.0 (1)
InstanceComponentConfigurationII:1.0 (5)
InstanceComponentConfigurationII:1.0 (4)
InstanceComponentConfigurationII:1.0 (6)
RfidReaderII:1.0 (10)
GraphicalOutputII:1.0 (11)
InstanceComponentII:1.0 (2)
InstanceComponentConfigurationII:1.0 (15)
InstanceComponentII:1.0 (3)
InstanceComponentConfigurationII:1.0 (16)

Zeigen sie auf ein Interface, um mehr Informationen darüber zu erhalten.

Figure 20. DAiSI Dynamic Adaptive System Monitor.

In order to monitor and debug a DAiSI-based dynamic adaptive system during development, the developer may use the so called "Dynamic Adaptive System Configuration Browser." This allows viewing the internal structure of the dynamic adaptive system in a graphical tree view.

As discussed in the introduction, DAiSI was used to realize and evaluate a couple of different applications. This allowed two main drawbacks of DAiSI to be identified: lack of service cardinalities and the centralized configuration mechanism.

In this paper, we have shown DAiSI's new component model supporting service cardinalities and the new decentralized dynamic configuration mechanism. The decentralized configuration mechanism is needed, in order to improve performance and fault-tolerance, because of the omitted centralized configuration service. Service cardinalities are called for to increase applicability, because real-life systems often have limitations regarding the amount of service users of their provided services, and may require more than exactly one service of a given type. A first dynamic adaptive system has been successfully implemented in the assisted sports training domain.

Consequently, further systems will be realized based on the new DAiSI version. Additional research is required to establish concepts to provide a proper balance between controllability of the system's applications and the autonomy of the system components participating in these applications. To further increase applicability, more research will be put into the introduction of interface roles, to be able to make additional constraints on available provided services, used to satisfy required services.

REFERENCES

[1] H. Klus and A. Rausch, "DAiSI-a component model and decentralized configuration mechanism for dynamic adaptive systems," in Proceedings of ADAPTIVE 2014, The Sixth International Conference on Adaptive and Self-Adaptive Systems and Applications, 2014.

[2] L. Northrop, P. Feiler, R. P. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, D. Schmidt, K. Sullivan, and K. Wallnau, "Ultra-large-scale systems—the software challenge of the future," Software Engineering Institute, Carnegie Mellon, Tech. Rep., June 2006.

[3] J. Kramer and J. Magee, "A rigorous architectural approach to adaptive software engineering," Journal of Computer Science and Technology, vol. 24, no. 2, pp. 183–188, 2009.

[4] C. Szyperski, "Component Software," Addison Wesley Publishing Company, 2002.

[5] M. P. Papazoglou, "Service-oriented computing: concepts, characteristics and directions," in Proceedings of the 4th International Conference on Web Information Systems Engineering (WISE 2003). 10-12 December, Rome, Italy: IEEE Computer Society Press, 2003, pp. 3–12.

[6] J. Magee, J. Kramer, and M. Sloman, "Constructing distributed systems in conic," in IEEE Transactions on Software Engineering vol. 15, no. 6, pp. 663–675, 1989.

[7] J. Kramer, "Configuration programming: a framework for the development of distributable systems," in Proceedings of IEEE International Conference on Computer Systems and Software Engineering (COMPEURO 90). 8-10 May 1990, Tel-Aviv, Israel: IEEE Computer Society Press, 1990. ISBN 0818620412, pp. 374–384.

[8] J. Kramer, J. Magee, M. Sloman, and N. Dulay, "Configuring objectbased distributed programs in rex," Software Engineering Journal, vol. 7, no. 2, pp. 139–149, 1992.

[9] R. R. Aschoff and A. Zisman, "Proactive adaptation of service composition," in: H. A. Müller, L. Baresi (Eds.): Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'12): Zürich, Switzerland, June 4-5, 2012. Los Alamitos, California: IEEE Computer Society Press, 2012, pp. 1–10.

[10] A. Rasche and A. Polze, "Configuration and dynamic reconfiguration of component-based applications with microsoft .NET," in Proceedings of the 6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2003). 14-16 May 2003, Hakodate, Hokkaido, Japan: IEEE Computer Society Press, 2003. ISBN 0-7695-1928-8, pp. 164–171.

[11] M. Anastasopoulos, H. Klus, J. Koch, D. Niebuhr, and E. Werkman, "DoAmI—a middleware platform facilitating (re-) configuration in ubiquitous systems," in Proceedings of the Workshop on System Support for Ubiquitous Computing (UbiSys), 2006.

[12] H. Klus, D. Niebuhr, and A. Rausch, "A component model for dynamic adaptive systems," in Proceedings of the International Workshop on Engineering of software services for pervasive environments (ESSPE 2007), 2007.

[13] D. Niebuhr, H. Klus, M. Anastasopoulos, J. Koch, O. Weiß, and A. Rausch, "DAiSI—dynamic adaptive system infrastructure," Technical Report Fraunhofer IESE, 2007.

[14] H. Klus, D. Niebuhr, and A. Rausch, "Dependable and usage-aware service binding," in Proceedings of the third International Conference on Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE 2011), 2011.

[15] D. Niebuhr, C. Peper, and A. Rausch, "Towards a development approach for dynamic-integrative systems," in Proceedings of the Workshop for Building Software for Pervasive Computing. 19th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), 2004.

[16] "Intelligent beer dispensing system," Webpage of the cebit exhibit 2010. [Online]. Available from: http://www2.in.tu-clausthal.de/~smartschank/systembeschreibung.php, accessed 2014.12.01.

[17] „DIRMEIER SmartSchank, intelligent beer dispensing system," DIRMEIER GmbH. [Online]. Available from: http://www.dirmeier.de/DIRMEIER-0-0-0-1-1-1.htm, accessed 2014.12.01.

[18] D. Herrling, "Deriving a framework from a number of dynamic adaptive system infrastructures," Master Thesis, TU Clausthal, Clausthal-Zellerfeld, 2014.

[19] C. Bartelt, T. Fischer, D. Niebuhr, A. Rausch, F. Seidl, and M. Trapp, "Dynamic integration of heterogeneous mobile devices," in Proceedings of the Workshop in Design an Evolution of Autonomic Application Software (DEAS 2005), ICSE 2005, St. Louis, Missouri, USA, 2005.

[20] T. Jaitner, M. Trapp, D. Niebuhr, and J. Koch, "Indoor simulation of team training in cycling," in ISEA 2006, E. Moritz and S. Haake, Eds. Munich, Germany: Springer, Jul. 2006, pp. 103–108.

[21] "Bilateral German-Hungarian collaboration project on ambient intelligent systems." [Online]. Available from: http://www.belami-project.hu/~micaz/belamiproject/history /part1, accessed 2014.12.01.

[22] M. Anastasopoulos, C. Bartelt, J. Koch, D. Niebuhr, and A. Rausch, "Towards a reference middleware architecture for ambient intelligent systems," in Proceedings of the Workshop for Building Software for Pervasive Computing, 20th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), 2005.

[23] M. Schindler and D. Herrling, "Emergency assistance system," Webpage of the cebit exhibit 2009. [Online], available from: http://www2.in.tu-clausthal.de /~Rettungsassistenzsystem/en/, accessed 2014.12.01.

[24] A. Rausch, D. Niebuhr, M. Schindler, and D. Herrling, „Emergency management system," In Proceedings of the International Conference on Pervasive Services 2009 (ICSP 2009), 2009.

[25] D. Niebuhr and A. Rausch, "Guaranteeing correctness of component bindings in dynamic adaptive systems based on run-time testing," in Proceedings of the 4th Workshop on Services Integration in Pervasive Environments (SIPE 09) at the International Conference on Pervasive Services 2009 (ICSP 2009), 2009.

[26] D. Niebuhr, "Dependable dynamic adaptive systems: approach, model, and infrastructure," Clausthal-Zellerfeld, Technische Universität Clausthal, Department of Informatics, Dissertation, 2010.

[27] A. Rausch and D. Niebuhr, "DemSy—a scenario for an integrated demonstrator in a smart city," ECas News Journal, 2010.

[28] C. Deiters, M. Köster, S. Lange, S. Lützel, B. Mokbel, C. Mumme, and D. Niebuhr, "DemSy—a scenario for an integrated demonstrator in a smart city," NTH computer science report, 2010.

[29] S. Lange, "Projektarbeit: regelüberwachung und regelbasierte konfiguration auf basis der ruleIT-methodik: modellierung einer Fallstudie," Unpublished Work, TU Clausthal, Clausthal-Zellerfeld, 2011.

[30] F. Paternò (Ed.), "Open pervasive environments for migratory iNteractive Services – Project Final Report," 2010.

[31] D. Herrling, "Projektarbeit: realisierung von zustandserhal-tung bei der migration von OSGi bundles," Unpublished Work, TU Clausthal, Clausthal-Zellerfeld, 2011, available from: URL: http://sse-world.de/index.php/download_file/view_inline/24/, accessed 2014.12.01.

[32] H. Klus, D. Niebuhr, and O. Weiss, "Integrating sensor nodes into a middleware for ambient intelligence," in S. Schäfer, T. Elrad, and J. Weber-Jahnke (Eds.): Proceedings of the Workshop on Building Software for Sensor Networks. Portland, Oregon, USA: ACM 2006. ISBN 1–59593–491–X.

[33] H. Klus, D. Niebuhr, and A. Rausch, "Towards a component model supporting proactive configuration of service-oriented systems," in ICEBE '07: Proceedings of the IEEE International Conference on e-Business Engineering. Hong Kong, China: IEEE Computer Society, 2007.

[34] C. Bartelt, B. Fischer, and A. Rausch, "Towards a decentralized middleware for composition of resource-limited components to realize distributed applications," in Proceedings of PECCS 2013, 3rd International Conference on Pervasive and Embedded Computing and Communication Systems, 2013, ISBN 978–989–8565–43–3.

[35] D. Niebuhr and A. Rausch, "Guaranteeing correctness of component bindings in dynamic adaptive systems," in Proceedings of the 35th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), Track on Service and Component Based Software Engineering (SCBSE). 2009.

[36] D. Niebuhr, A. Rausch, C. Klein, J. Reichmann, and R. Schmid, "Achieving dependable component bindings in dynamic adaptive systems – a runtime testing approach," in Porceedings of the 3rd IEEE International Conference on Self-Adaptive and Self-Oranizing Systems (SASO 2009), 2009.

[37] A. Rausch, J. P. Müller, D. Niebuhr, S. Herold, and U. Goltz, "IT ecosystems: a new paradigm for engineering complex adaptive software systems," in 6th IEEE International Conference on Digital Ecosystems Technologies (DEST 2012), 2012, ISSN 2150-4938.

[38] H. Klus, B. Schindler, and A. Rausch, "Dynamic reconfiguration of application logic during application migration," Version: 2011, in F. Paternò (Ed.): Migratory Interactive Applications for Ubiquitous Environments, Springer-Verlag London Limited, 2011, DOI 10.1007/978–0–85729–250–6_7, ISBN 978–0–85729–249–0.

[39] H. Klus, "Anwendungsarchitektur-konforme konfiguration selbstorganisierender softwaresysteme," Clausthal-Zellerfeld, Technische Universität Clausthal, Department of Informatics, Dissertation, 2013.