Component Templates and Service Applications Specifications to Control Dynamic Adaptive System Configurations

Holger Klus

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

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

***Abstract*—Dynamic adaptive systems are systems that change their behavior at run time, based on system, user, environment and context information and needs. System configuration in terms of structure and behavior of open, self-organized systems cannot completely be predicted beforehand: New components may join, others may leave the system, or the behavior of individual components of the system may change over time. But in many cases, it is necessary to ensure the compliance of the resulting overall system configuration to users, environment and context requirements. Therefore we have elaborated an approach to specify those requirements based on so called component templates and service application specifications. These specifications can be described without the necessity to know individual components, their specific interfaces or possible system configurations. Thus, we can control the resulting system configurations of an open, self-organizing system with respect to users, environment and context requirements. Our approach has been implemented on top of our component model and corresponding platform implementation called Dynamic Adaptive System Infrastructure (DAiSI).**

Keywords—self-adaptation; dynamic adaptive systems; component templates and service application specification; adaptive component model; decentralized configuration.

I. Introduction

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 [1].

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 additional value. The challenge is therefore to provide software systems that are correct, stable and robust in the presence of increasing challenges such as change and complexity [5].

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 [2].

Dynamic change, in contrast to evolutionary change, occurs while the system is operational. Dynamic change requires that the system adapts at run time. Therefore we must plan for automated management of adaptation. The systems themselves must be capable of determining what system change is required and initiate and manage the change process wherever needed. This is the aim of self-managed systems [3].

Self-managed systems are those capable of adapting to the current context as required through 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' systems [4].

Providing dynamic adaptive systems is a great challenge in software engineering [5]. 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. DAiSI is a service-oriented and component based platform to implement dynamic adaptive systems [6].

Based on the existing components and their provided and required services DAiSI is able to autonomously find and establish during run time valid system configurations with respect to specific optimization goals and system guarantees. Even if new components join, or others leave DAiSI, or the behavior of individual components within DAiSI changes over time, DAiSI is able to reconfigure the overall system and establish a new valid system configuration at run time.

But in many cases, it is necessary to ensure the compliance of the resulting overall system configuration to users, environment and context requirements. Therefore we have elaborated an approach to specify those requirements based on so called component templates and service application specifications. The basic idea of our approach is to specify services users or the environment may be interested in form of so called “service application” specifications.

A service application specification consists of a set of so called “component templates”. Each template is a placeholder for a set of components with specific properties. The application developer specifies the properties of component templates and service applications during design time. During run time, DAiSI tries to establish the required service applications by assigning autonomously existing components to compatible templates in the corresponding service application specifications.

These specifications can be described without the necessity to know individual components, their specific interfaces or possible system configurations. We have successfully implemented the component templates and service applications specifications on top of the existing component model of DAiSI. Thus, DAiSI is not only able to find and establish valid system configurations but also to find and establish them with respect to users, the environment and context requirements, which have to be explicitly expressed in component templates and service applications specifications.

The development of DAiSI was always motivated through running application examples and demonstrators. As DAiSI has been developed for more than ten years, we have demonstrated the application of our approach and our infrastructure in a couple of different research demonstrators and industrial prototypes and products.

The rest of the paper is structured as follows: In Section II, we present other works, we see as related to the DAiSI and its newest additions. Section III presents the fundamentals of DAiSI as it was prior to the additions, presented in this paper. Section IV describes a small sample application we use to illustrate the need to control possible system configurations in dynamic adaptive systems. Section V presents an approach to describe valid system configurations with regard to applications. In Section VI, we provide a notation for the specification of application requirements. Section VII presents an algorithm that leads to a requirements conform system configuration and explains why DAiSI only produces valid system configurations during run time with respect to users, environment and context requirements. 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 [7].

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 [8].

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 a 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.

REX is a framework for the support of dynamic-adaptive systems. It used the experience gained in the research for CONIC [9] 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 [10][11].

Current frameworks such as ProAdapt [12] and Config.NETServices [13] 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, the framework provides a generic component configuration mechanism. 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.

In [14], the authors presented a solution to ensure syntactical and semantical compatibility of web services. They used the Web Service Definition Language (WSDL) and enriched it with the Web Service Semantic Profile (WSSP) for the semantical information. Additionally they allowed an application architect to further reduce the configuration space through the specification of constraints. While their approach is able to solve the sketched problem of preventing the wiring of components that should not be connected, they only focus on the service definition and compatibility. Our DAiSI approach defines an infrastructure in which components are executed that implement a specific component model. We do want to compose an application out of components that can adapt its behavior at run-time. We achieve this by mapping sets of required services to sets of provided services and thus specifying which provided services depend on which required services. The solution presented in [14] does not offer a component model. All rules regarding the relation between required and provided services would have to be specified as external constraints. The authors in [15] provided a different solution to ensure semantic compatibility of web services. However, the same arguments as for [14] regarding the absence of a high-level component model hold true.

With regard to the application architecture aware adaptation, Rainbow [16][17] is one of the most dominant and well-known frameworks. Rainbow uses invariants for the specification of constraints in its architecture description language. For each invariant a method for the adaptation of the system can be specified. The method is then executed whenever the invariant is violated. This approach however requires the knowledge of all component types at design-time, which is opposing our goal of an open system. Additionally, the developer has to implement the adaptation steps individually for every invariant. This imperative method for adaptation requires the component requiring the adaptation to have a view of the complete system and additionally introduces a big overhead at design-time as well as at run-time.

R-OSGi [18] takes advantage of the features developed for centralized module management in the OSGi platform, like, e.g., dynamic module loading and –unloading. It introduces a way to transparently use remote OSGi modules in an application while still preserving good performance. Issues like network disruptions or unresponsive components are mapped to events of unloaded modules and thus can be handled gracefully – a strength compared to many other platforms. However R-OSGi does not provide means to specify application architecture specific requirements. As long as modules are compatible with each other they will be linked. The module developer has to ensure the application architecture at the implementation level. Opposed to that, our approach proposes a high level description of application architectures through application templates that can be specified even after the required components have been developed.

III. The Core of the Existing DAiSI Platform

This section will introduce the foundations of the DAiSI platform, consisting of a dynamic adaptive component model, a domain architecture model and a decentralized configuration service.

As already briefly mentioned DAiSI components interact with each other through services. Each DAiSI component consists out of a set of component configurations. Each component configuration defines a set of required services and a set of provided services. Figure 1 shows a sketch of a DAiSI component with some explanatory comments for an athlete in the biathlon sports domain.

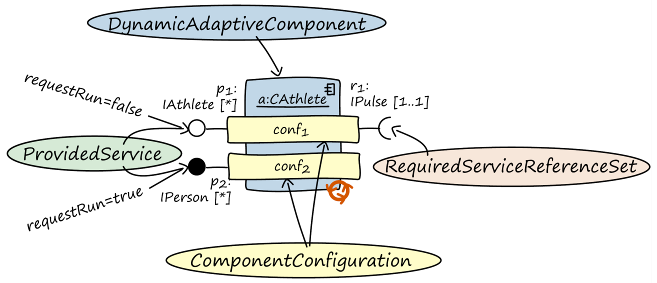


Figure 1. Notation for DAiSI components and corresponding concepts.

A component is depicted as a blue rectangle. Component configurations are bars that extend over the borders of the component and are depicted in yellow here. Associated to the component configurations are the provided and required services. The notation is similar to the UML lollipop notation [19] with full circles resembling provided, and semi circles representing required services. A filled circle indicates that the service is meant to be executed and thereby provided within the system, even if no other service requires its use.

Figure 1 shows the *CAthlete* component, consisting of two component configurations: conf_1 and conf_2. The first component configuration requires exactly one service variable r_1 of the *IPulse* interface. The second component configuration does not require any services to be able to provide its service p_2 of *IPerson*. The service can be used by any number of service users (the cardinality is specified as *). The other component configuration (conf_1) could provide the service p_1 of the type *IAthlete*, which could again be used by any number of users. The small orange circle with the three arrows in the lower right corner indicates that this component is self-organizing, i.e., it does not require a centralized configuration service to resolve its requirements and change its execution state.

Figure 2 shows the DAiSI component model as an UML class diagram [19]. The *DynamicAdaptiveComponent* class represents the component itself, represented as the light blue box in the notation example. It has three types of associations to the *ComponentConfiguration* class, namely *current*, *activatable*, and *contains*. The *contains* association resembles the non-empty set of all component configurations. It is ordered by quality from best to worst, with the best component configuration being the most desirable. Quality refers to the count of provided services, as well as the quality they are provided in. A subset of the contained are the *activatable* component configurations. These have their required services resolved and could be activated. An *active* component configuration produces its provided services. The *active* component configuration is represented by the *current* association in the component model, with the cardinality allowing one or zero current component configurations to be executed for a component.

The required services (represented by a semi-circle in the component notation in Figure 1) are represented by the *RequiredServiceReferenceSet* class. Every component configuration can declare any number of required services. The *resolved* association represents those that are resolved. Provided services (noted as full circles on the left hand side in Figure 1) are represented by the *ProvidedService* class. The flag *requestRun*, represented by the full circle being filled with black in the component notation, indicates that the service should be activated, even if no other service requires its use. This is typically the case for services that provide graphical user interfaces or some functionality directly to the end user.

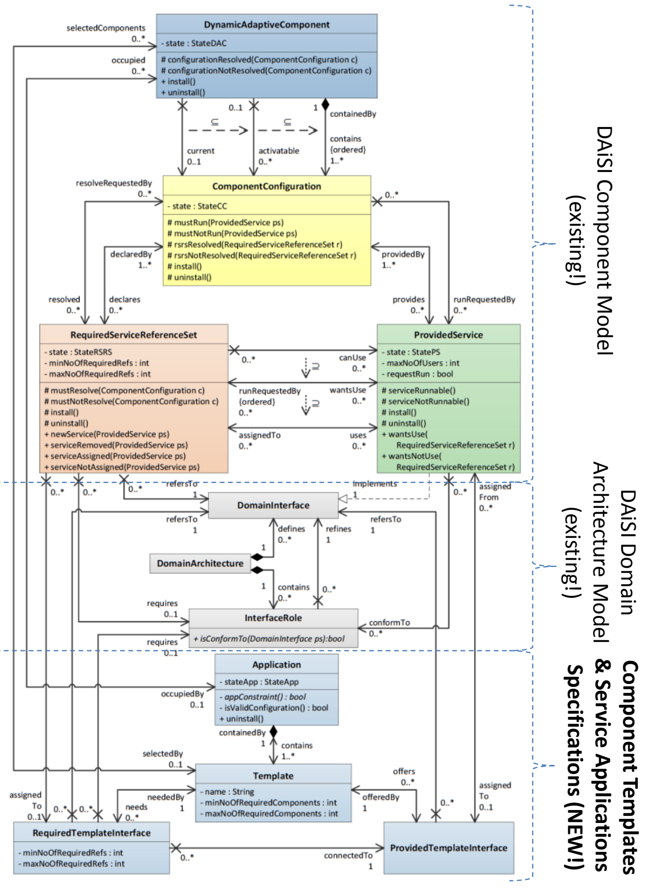

DynamicAdaptiveComponent
- state : StateDAC
# configurationResolved(ComponentConfiguration c)
# configurationNotResolved(ComponentConfiguration c)
+ install()
+ uninstall()
selectedComponents 0..*
occupied 0..*
current 0..1
activatable 0..*
containedBy 1
contains {ordered} 1..*
ComponentConfiguration
- state : StateCC
# mustRun(ProvidedService ps)
# mustNotRun(ProvidedService ps)
# rsrsResolved(RequiredServiceReferenceSet r)
# rsrsNotResolved(RequiredServiceReferenceSet r)
# install()
# uninstall()
resolveRequestedBy 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()
+ newService(ProvidedService ps)
+ serviceRemoved(ProvidedService ps)
+ serviceAssigned(ProvidedService ps)
+ serviceNotAssigned(ProvidedService ps)
canUse 0..*
runRequestedBy {ordered} 0..*
wantsUse 0..*
assignedTo 0..*
uses 0..*
ProvidedService
- state : StatePS
- maxNoOfUsers : int
- requestRun : bool
# serviceRunnable()
# serviceNotRunnable()
# install()
# uninstall()
+ wantsUse(RequiredServiceReferenceSet r)
+ wantsNotUse(RequiredServiceReferenceSet r)
refersTo 1
implements 1
assigned From 0..*
DomainInterface
defines 0..*
refines 1
DomainArchitecture
contains 0..*
requires 0..1
InterfaceRole
+ isConformTo(DomainInterface ps):bool
conformTo 0..*
Application
- stateApp : StateApp
- appConstraint() : bool
- isValidConfiguration() : bool
+ uninstall()
occupiedBy 0..1
containedBy 1
contains 1..*
selectedBy 0..1
Template
- name : String
- minNoOfRequiredComponents : int
- maxNoOfRequiredComponents : int
neededBy 1
needs 0..*
offers 0..*
offeredBy 1
assigned To 0..1
RequiredTemplateInterface
- minNoOfRequiredRefs : int
- maxNoOfRequiredRefs : int
connectedTo 1
ProvidedTemplateInterface
DAiSI Component Model (existing!)
DAiSI Domain Architecture Model (existing!)
Component Templates & Service Applications Specifications (NEW!)

Figure 2. DAiSI component model, DAiSI domain architecture model, and the new additional template & application specifications.

The provided and required services, more precisely their respective classes in the component model, are associated with each other through three associations. The first association *canUse* represents the compatibility between two services. If a provided service can be bound to the service requirement of another class, these two are associated through a *canUse* association. A subset of the *canUse* association is *wantsUse*. At run-time it resembles a kind of reservation of a particular provided service by a required service reference set. After the connection is established and the provided service satisfies the requirement, they are part of the *uses* association, which represents the actual connections. All classes covered to this point implement a state machine to maintain the state of the DAiSI component. If you want to know more about the state machines and the configuration mechanism, please refer to [20].

To this point we have covered the basic building blocks of the DAiSI component model. Another already established part of DAiSI is the domain architecture model. The relation to the actual developed application becomes apparent if you consider the *DomainArchitecture* class. It defines any number of *DomainInterfaces*. These are the interfaces that define the provided and required services. Thus, every *ProvidedService* class implements a domain interface, while each required service reference set refers to exactly one.

There are numerous examples in which the role of a specific domain service has to be considered in order to establish the desired system configuration. For that reason the class *InterfaceRole* enables the specification of additional criteria for the conformance of provided and required services. An interface role references exactly one domain interface and may define additional requirements regarding that domain interface. A provided service only fulfills an interface role if it implements the domain interface and as well complies with the conditions defined in the interface role. Consequently a required service reference set not only requires compatibility of the domain interface, but also of the interface role to be able to use a provided service. For more information about the DAiSI domain architecture model and interface roles consider [21].

Beside the DAiSI component model and the DAiSI domain architecture model a decentralized dynamic configuration mechanism was also already established in the DAiSI platform. The set of services that implement the domain interface referred by the *RequiredServiceReferenceSet* is represented by *canUse*, as stated before. Note, this only guarantees a syntactically correct binding. Interface roles in addition provide a compatibility check with respect to a given common domain architecture. In [22][23] we have shown how this approach can be extended to guarantee behavior correct binding during run time, even in case of changes to the local and global state.

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 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, usage requests are placed at the services in *canUse* by calling *wantsUse* and those service references are copied to the *wantsUse* set.

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 relies on the state machines presented in more detail in [20] and sketched in the following paragraphs.

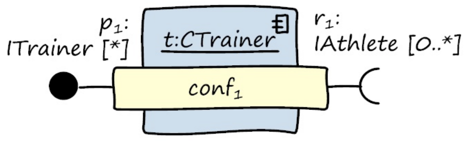


Figure 3. CTrainer component.

Assume a given component as shown in Figure 3. The component *t* of type *CTrainer* has one 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 references 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 4.

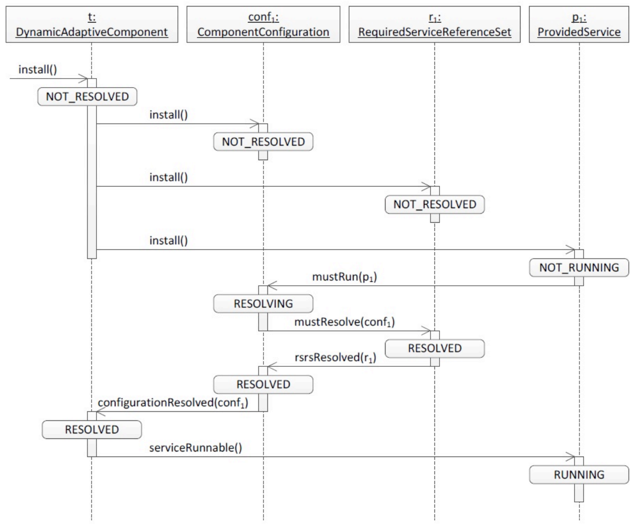


Figure 4. Sequence diagram showing the triggers and states of a stand alone DAiSI component.

Now assume two components: The *CAthlete* component, shown on the left hand side of Figure 5, requires zero or one reference to a service of type *IPulse*. The second component, *CPulse*, shown on the right hand side of Figure 5, provides a service of type *IPulse*. Note, this service can only be exclusively used by a single component. Figure 6 shows the states and triggers of the involved state machines in a sequence diagram for this example.

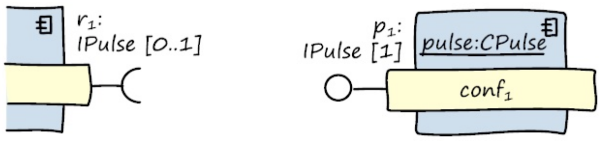


Figure 5. CAthlete and CPulse components.

Once the *CPulse* component is installed, DAiSI integrates the new service in the *canUse* relationship of the *RequiredServiceReferenceSet* r_1 of the component *CAthlete*. Then DAiSI informs the *CAthlete* component that a new service that can be used is available. 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*).

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*).

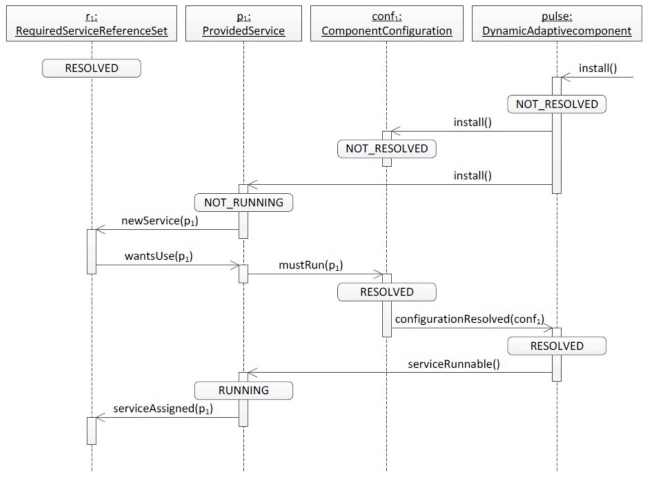


Figure 6. Inter-component configuration mechanism.

A detailed example of the presented configuration algorithms is presented in [20].

IV. Introduction of the Running Example and The Need to Control System Configurations

For this example, we assume that a self-organizing system is to be developed, which supports the training of biathletes, such as briefly described in the previous sections. In this particular case, the system is to provide the services described below.

First, a trainer is to be presented with an overview of his athletes' performance data, where data from at least one athlete should be displayed. For this purpose, it is assumed that the component presented in Figure 7 is available.

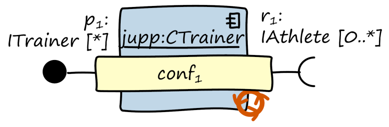


Figure 7. The trainer component available in the system.

The required functionality is provided by the service p_1, which implements the interface *ITrainer*. The service defines a dependency with services that implement the interface *IAthlete*. However, the service can also be run when an athlete system is not available in the system. The implementation of the trainer component would have to be adapted in order to meet the requirement that the trainer service can only be run when it has access to at least one athlete service. Moreover, the attribute *minNoOfRequiredRefs* of r_1 from Figure 7 would have to be set to 1. However, a component code cannot always be modified in this way. In addition, adapting it manually for the specific application purpose contradicts the original purpose of a component. The solution presented in the remainder of this section allows the application-specific specification of the minimum and maximum number of required references for *RequiredServiceReferenceSets* without having to adapt the component source code.

The individual athletes' performance data within the application are provided via the interface *IAthlete*. For the example, it is assumed that the component presented in Figure 8 is available.

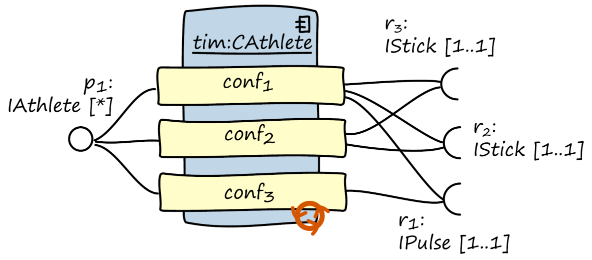


Figure 8. The CAthlete component.

The component defines three *ComponentConfigurations* with *conf$_1$* specified as best configuration and *conf$_3$* as worst. The *conf$_3$* configuration can be activated if *r$_1$* can be connected to a service that implements the interface *IPulse*. The *conf$_2$* configuration can be activated, if *r$_2$* and *r$_3$* are each connected with a ski pole. The *conf$_1$* configuration is activated if the dependencies of all three *RequiredServiceReferenceSets* can be resolved. In all three configurations, the component provides a service that implements the domain interface *IAthlete*. It defines a method *getPulse():int* to query the current pulse and also a method *getSkiingTechnique():String*, which returns the currently used skiing technique (double poling/diagonal technique). If the *conf$_3$* configuration is active, the call *getSkiingTechnique* returns the value *null*. If, in contrast, the *conf$_2$* configuration is active, the call *getPulse* returns the value -1.

For the example application, it is now assumed that the skiing technique is to be analyzed in particular, i.e., only the *conf$_2$ ComponentConfiguration* of the athlete component tim from Figure 8 is relevant. Even if one pulse service and two ski pole services are available, the *conf$_1$* configuration should not be activated even though it is the best component according to the component specification. The framework presented so far, and described in Section III, does not provide the potential to influence the *ComponentConfiguration* of a component from an application-specific point. In this context, it is only possible to implement the component specifically to the application. In this section, expansions of the existing framework are described, which enable such an application-specific influence on the activation of component configurations.

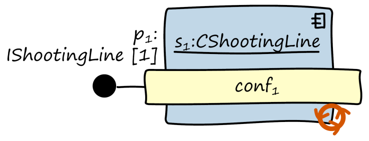


Figure 9. The CShootingLine component.

It should also be possible for the example system described here to allow shooting training. In this case, one shooting lane should be available for each athlete. In the system, each shooting lane should be represented by a service, each implementing the domain interface *IShootingLine*. One example of such a component is presented in Figure 9.

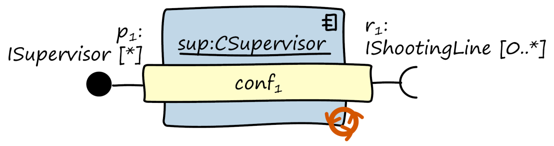


Figure 10. The structure of the component CSupervisor.

In this case, the service p$_1$ of the component also starts when there is no user in form of another component, as the flag *requestRun* is set (indicated by the shaded circle). However, for this example, the system should only allow shooting if a shooting supervisor is present. This is represented in the system by a service that implements the domain interface *ISupervisor*. The component presented in Figure 10 provides such a service.

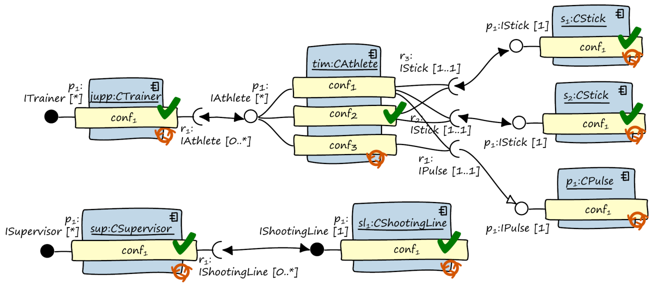


Figure 11. A system configuration that meets the requirements.

At this point, the most complex requirement placed on the system has an influence. The system must guarantee that exactly one shooting lane component is available for each athlete connected to the trainer component. This means that the number of those services used by the shooting supervisor component must be in agreement with the athlete components, which the trainer component accesses.

One system configuration that meets all criteria described above is presented in Figure 11. Here, a trainer component is connected with an athlete component, which in turn is connected to a left and a right ski pole. In addition, the application consists of a shooting supervisor component, which in turn is connected to a shooting lane component.

In the current DAiSI, such system configuration requirements cannot be specified and therefore cannot be guaranteed. Moreover, further requirements would be relevant for this application, such as: if a new athlete component is added to the system in the configuration described above, it should only be integrated into the application when a shooting lane component is available for this athlete. The application is also stopped, for example, when the athlete component from Figure 11 is only connected with one ski pole component.

DAiSI as described in Section III (without the new part for the specification of the component templates and application specifications) is not able to implement requirements relating to the application as a whole. For example, the better *conf$_1$* configuration of the component “tim” from Figure 11 would be activated, although this is explicitly considered undesirable by the application developer.

V. Application Specific System Configuration

This section provides a short overview of the solution approach for the specification of valid system configuration requirements. One application configuration consists of a number of components, as well as connections between these components. Therefore, the primary task of DAiSI is to select the components that can be considered for a configuration conforming to the application architecture out of the number of all available components. In addition, the components must be connected in such a way that all specified requirements are met.

The solution presented below enables requirements specification with which components are considered for use within the application. On the other hand, the manner in which these components are to be connected with each other can be defined. Based on such a specification and number of components available, the framework developed will later be able to create an application architecture-conform configuration. Furthermore, the framework reconfigures the application automatically, if the requirements are no longer met. The solution approach is continuously based on a system of self-organizing components. However, the configuration is accomplished with the assistance of a central but light configuration unit.

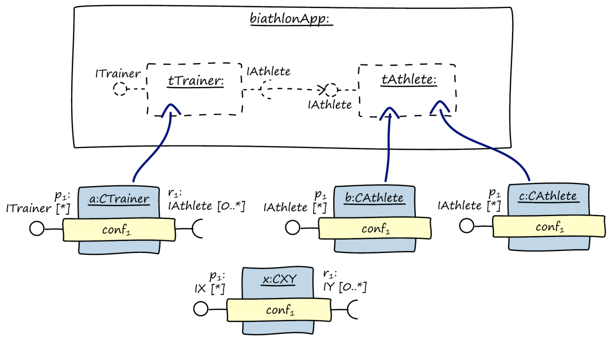


Figure 12. Suitable components for an application configuration.

The criteria for selection of suitable components for an application are defined with the assistance of so-called templates. An application specification consists of one or more of such templates. In this way, the biathlon application described above could, for instance, consist of a template for trainer components, and one for athlete components, one for shooting lane components, etc. For each of these templates, requirements can be stored that specify under which circumstances a component is compatible with a template. For example, constraints can be stored for an athlete template, which specifies that only such components that provide a service that implements the domain interface *IAthlete* are compatible. The framework ensures that for the runtime, only components matching the outline are allocated to the template. From then on, a template will be represented by a rectangle with dashed lines. Requirements related to required and provided component services are represented visually by circles and semi-circles with dashed lines (described in detail below). In Figure 12, two placeholders within an application template can be seen. One or two components can be allocated to the application, while one of the given components remains ignored, as it is not compatible.

The components selected must be connected with each other in the next step, in order to obtain an executable system. For this purpose, in addition to the templates, the links between templates are defined, and represented as dashed arrows (see Figure 12). They provide information on how the allocated components are to be connected with each other. In this way it is possible to define that each component allocated to the *tTrainer* template in Figure 12 must be connected with at least one component, which is allocated to the *tAthlete* template. Later during run time, the framework ensures that the requirements related to the links between the components are considered. Figure 13 shows one possible resulting system configuration.

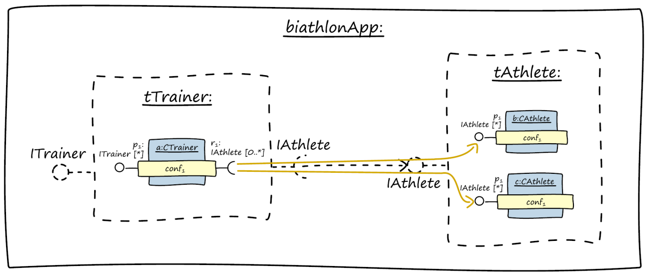
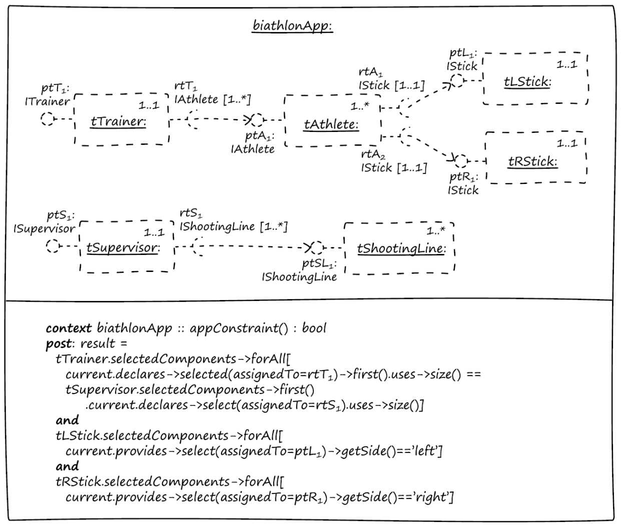


Figure 13. Generation of a valid configuration.

The following paragraphs present the requirements in detail, how they can be specified and how they are implemented in the framework.

VI. Specification of Application Requirements With Component Templates and Service Applications

The DAiSI platform is expanded to describe application-specific requirements for system configurations. These expansions represent the new parts of DAiSI in Figure 2, which are necessary to specify application-specific requirements for the system configurations, with the assistance of component templates and service application specifications.



```
context biathlonApp :: appConstraint() : bool
post: result =
  tTrainer.selectedComponents->forAll[
    current.declares->selected(assignedTo=rtT1)->first().uses->size() ==
    tSupervisor.selectedComponents->first()
        .current.declares->select(assignedTo=rtS1).uses->size()]
  and
  tLStick.selectedComponents->forAll[
    current.provides->select(assignedTo=ptL1)->getSide()=='left']
  and
  tRStick.selectedComponents->forAll[
    current.provides->select(assignedTo=ptR1)->getSide()=='right']
```

Figure 14. Graphical and textual application specification.

In order to be able to present application specifications in a concise manner, a uniform notation for the specification elements introduced above is defined for the remainder of this work. Figure 14 shows a possible application specification for a biathlon application.

An application is represented as rectangle with the name of the application noted at the top. Each Template is represented as a rectangle with dashed lines, which contains the name of the template. Within a template, the contents of the attributes *minNoOfRequiredComponents* and *maxNoOfRequiredComponents* are noted at the top right. A *ProvidedTemplateInterface* is represented as circle with dashed line, which is labeled with the name, as well as the referenced domain interface. *RequiredTemplateInterfaces* are represented correspondingly as semi-circles with dashed lines. They are also labeled with the referenced domain interface, the referenced interface role, if applicable, and with the name. Links between a *RequiredTemplateInterface* and a *ProvidedTemplateInterface* (*connectedTo*) are visualized with a dashed arrow. The predicate *appConstraint* specification is specified in a separate area under the templates.

VII. Requirement Conform Dynamic Adaptive Application Configuration

The aim of the framework is to create an application configuration, which meets all specified requirements. As soon as this is achieved, the applications' state machine transitions from NOT_RUNNING to RUNNING. In other words: if an application is in the state RUNNING, the application configuration created conforms to the application architecture.

This section describes how a valid application configuration can be generated automatically. The method suggested here follows a brute-force approach, which iteratively generates all possible configurations. It is sketeched in Figure 15 as pseudo code. While this is not optimal with regard to resources, it is sufficient to generate a valid system configuration. The focus of this paper is not the configuration algorithm, but the introduction of application templates.

```
1  boolean createValidConfiguration() {
2    while(possibleComponentAssignmentSets.hasNext()) {
3      possibleComponentAssignmentSets.next().realize();
4
5      while(possibleInterfaceAssignmentSets.hasNext()) {
6         possibleInterfaceAssignmentSet.next().realize();
7
8        while(possibleUsageSets.hasNext()) {
9          possibleUsageSets.next().realize();
10
11         if(isValidConfiguration()) {
12           return true;
13         }
14       }
15     }
16   }
17   return false;
18 }
```

Figure 15. createValidConfiguration() method, pseudo code listing.

Since a valid configuration, which meets the requirements can change at any time, in such a way that it no longer conforms to the application architecture, the application configuration is checked cyclical for conformance to the application architecture. Just as the configuration algorithm offers room for improvements with regard to performance, the same holds true for the cyclical application architecture conformance checks.

As soon as the configuration no longer meets the defined application architecture-specific requirements, and therefore the predicate *isValidConfiguration* is evaluated as *false*, the applications' state machine changes back to the state NOT_RUNNING.

The main task of the configuration process is to use a number of components to create an application configuration meeting all the requirements. For this purpose, the framework initially creates a configuration that meets all structural requirements. This configuration is executed and the services commence. It is in the next step a check is made to ensure that the service state requirements are met, since these requirements can only be confirmed when these services are running. If the requirements are not met, a new structurally compatible configuration must be created.

The algorithm is divided into two parts: one part creates an application configuration (lines 2-9 in Figure 15) and the other parts checks the configuration for conformity with the requirements lines (11-12 in Figure 15). Creating a configuration requires three steps. Firstly, selecting the components, then the *ProvidedService*- and *RequiredServiceReferenceSets* must each be allocated to a *ProvidedTemplateInterface* and *RequiredTemplateInterface*, respectively. Therefore, the two *assignedTo* quantities must be defined. Finally, the *uses* set must be determined for each *RquiredServiceReferenceSet*.

The initial situation of the configuration process is a set of available components. A selection must be made to obtain an application configuration. To accomplish this, assignment of the *selectedComponents* set is created for each template, with the component static properties already being considered. The application calculates the set of all possible assignment combinations and makes them available via an iterator (*possibleComponentAssignmentSets* from Figure 15), based on the components available and the application specification, the method *realize* implements the specific assignment.

For clarification, study an application specification with two templates as presented in Figure 16. It is also assumed that five components are available in the system.

In this example, the components *a* and *b* can be allocated to the *tTrainer* template and just one component must be allocated to the template in order to fulfil the application requirements. Both components provide a service that implements the *ITrainer* domain interface and define a *RequiredServiceReferenceSet* that references the *IAthlete* domain interface. Only component *d* can be allocated to the *tAthlete* template since this component is the only one that meets the template structural requirements. A total of two components are available for the *tLStick* and *tRStick* templates and exactly one component must be allocated to each of these two templates, in order to be able to meet the application requirements. This results in a number of possible allocations of components to templates. The configuration algorithm makes a selection, which is then implemented by the framework. In the next step, *ProvidedServices* is allocated to *ProvidedTemplateInterfaces* and *RequiredServiceReferenceSets* to *RequiredTemplateInterfaces*.

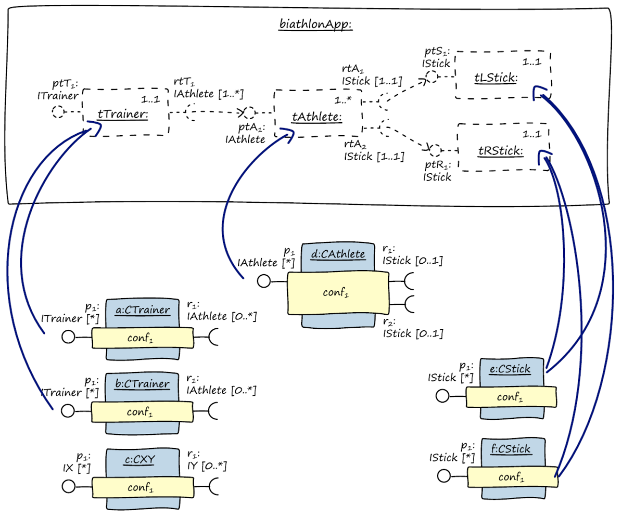


Figure 16. Allocation of components to templates.

ProvidedServices of a component could fit to several *ProvidedTemplateInterfaces*. Since *ProvidedServices* must be allocated to *ProvidedTemplateInterfaces* during run time, the framework must make a decision here. The same applies to *RequiredServiceReferenceSets* and *RequiredTemplate-Interfaces*. For example, the *RequiredTemplateInterfaces* of the *tAthlete* template in Figure 16, do not reference any interface roles but only the *IStick* domain interface as presented in Figure .

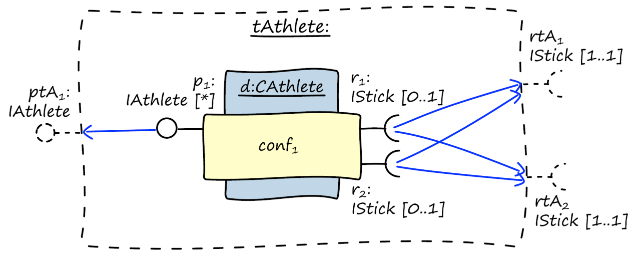


Figure 17. Allocation of component interfaces to template interfaces.

In this example, the *RequiredServiceReferenceSet* r_1 can be allocated to *RequiredTemplateInterface* rtA_1 as well as rtA_2. The same applies to RequiredServiceReferenceSet r_2.

Within the algorithm in Figure 15, all possible allocations, which result from the allocation of components to templates in the previous step are now iterated. In the component model, the allocation between *RequiredServiceReferenceSet* and *RequiredTemplateInterface*, and between *ProvidedService* and *ProvidedTemplateInterface* are represented by the *assignedTo* association. The possibilities are iterated with the *possibleInterfaceAssignmentSets* iterator (lines 5+6). The returned assignment is then implemented by calling *realize*. In the next step, the *uses set* is assigned to the *RequiredServiceReferenceSets* of the components, which were allocated previously to the *selectedComponents* quantity. The last step for the generation of the configuration algorithm consists of creating the *use* relations between the components.

The goal is assignment of the *uses* set for each *Required-ServiceReferenceSet* of all components included in the application, so that the requirements of the application specification can be met.

It often happens that there are several possibilities for the assignment of this set. The following situation is considered for illustration purposes (see Figure 15). In this case, the use of the provided service for both athlete components is considered for the *RequiredServiceReferenceSet* r_1. In this case, the empty quantity would not be an invalid assignment since the value 1 is specified for the attribute *minNoOfRequiredRefs* of the component. In the algorithm in Figure 15, these possible assignments are iterated with *possibleUsageSets* iterator, in order to create valid application configurations.

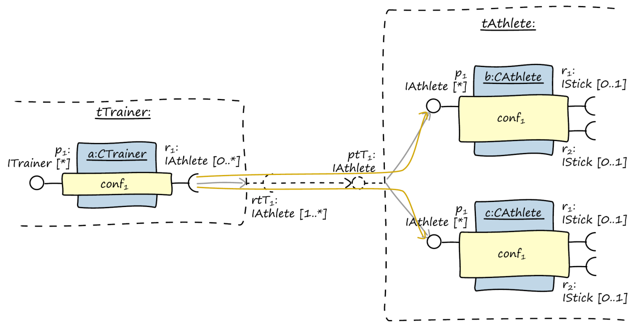


Figure 15. Example for possible assignments of the quantity *uses*.

After making a component selection, subsequently allocating the services and then assigning the *uses* set of all *RequiredServiceReferenceSets*, a running configuration is created automatically. For this purpose, the self-configuring components are informed at each stage if they are part of the application, to which template they should allocate themselves, to which template interfaces their services and *RequiredServicesReferenceSets* should be allocated and with which service they should connect. The individual iterators of the algorithms are realized for individual components.

After creating a configuration with the procedure described above, the remaining applications of the application specification can now also be checked for conformity. The predicate *isValidConfiguration* must now be evaluated. Only if this predicate is evaluated to *true*, the application changes its state to RUNNING. Otherwise, a new configuration must be created. The algorithm presented here is only a sketch of the procedure for creating a configuration, which conforms to the defined application architecture-specific requirements. Other algorithms are possible and can be found in [24].

VIII. Conclusion and Future Work

In this paper we introduced a major extension of our dynamic adaptive system infrastructure called DAiSI. DAiSI enables applications to adapt themselves automatically during run time. It is able to integrate new components during and handle the loss of components by reconfiguration. In the former version, DAiSI tried to find a configuration, which is optimal for each individual component. As this may lead to applications where each component is running in its optimal configuration, but where the application as a whole does not meet the requirements, we presented an extension of DAiSI, which enables the specification of application-specific requirements on the one hand, and its automatic realization during run time on the other.

Our concept introduced so called templates, which define fitting-criteria for component instances. Furthermore, the concept enables users to specify requirements regarding connections between components. Our infrastructure is able to interpret this specification, and realize a suitable application configuration based on available components in the system. One of the major characteristics of our approach is, that during design-time no knowledge about existing components and their instances is required. The match of components to templates is performed automatically during runtime based on provided/required interfaces, interface roles and predicates.

In the future, we will further extend our concept and our implementation by providing more specification capabilities regarding component selection and component interconnection. There is for example a possibility missing to specify an order on available components to enforce the use of, e.g., the best three components. Furthermore, there still exists potential for improvements of our prototypical implementation.

However, the extension presented in this paper provides a sustainable concept towards the realization of decentralized, dynamic adaptive systems, while satisfying application-specific requirements, which has been implemented as a proof of concept.

References

[1] L. Northrop, et al., “Ultra-large-scale systems—the software challenge of the future,” Software Engineering Institute, Carnegie Mellon, Tech. Rep., June 2006.

[2] D. Leffingwell and D. Widrig, “Managing Software Requirements: A Unified Approach,” Addison-Wesley Professional, 2003.

[3] J. Kramer and J. Magee, “Self-managed systems: an architectural challenge,” in FOSE ‘07 Future of Software Engineering, IEEE Computer Society, Washington DC, USA, 2007, pp. 259–268.

[4] B. H. C. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee, “Software engineering for self-adaptive systems: A second research roadmap,” in Software Engineering for Self-Adaptive Systems II, Springer, Heidelberg, 2013, pp. 1–26.

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

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

[7] C. Szyperski, “Component Software,” Addison Wesley Publishing Company, 2002.

[8] 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.

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

[10] 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), Tel-Aviv, Israel, 1990, pp. 374–384.

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

[12] 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.

[13] 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.

[14] T. Kawamura, J.-A. De Blasio, T. Hasegawa, M. Paolucci, and K. Sycara, “Public deployment of semantic service matchmaker with uddi business registry,” in The Semantic Web ISWC 2004, ser. Lecture Notes in Computer Science, S. McIlraith, D. Plexousakis, and F. van Harmelen, Eds. Springer Berlin Heidelberg, 2004, vol. 3298, pp. 752–766.

[15] T. Haselwanter, P. Kotinurmi, M. Moran, T. Vitvar, and M. Zaremba, “Wsmx: A semantic service oriented middleware for b2b integration,” in International Conference on Service-Oriented Computing. Springer, 2006, pp. 4–7.

[16] D. Garlan, S. Cheng, A. Huang, B. Schmerl, and P. Steenkiste, “Rainbow: Architecture-based self-adaptation with reusable infrastructure,“ in Computer 37, 2004, No. 10, pp. 46–54.

[17] S. Cheng, “Rainbow: Cost-effective software architecture-based self- adaptation,“ Pittsburgh, Carnegie Mellon University, School of Computer Science, Dissertation, 2008.

[18] J. Rellermeyer, G. Alonso, and T. Roscoe, “R-Osgi: distributed applications through software modularization,” in Proceedings of the ACM/IFIP/USENIX 2007 Conference on Middleware (Middleware ’07), Newport Beach, California, 2007, pp. 1–20.

[19] OMG, OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1, Object Management Group Std., Rev. 2.4.1, August 2011. [Online, retrieved: 06, 2015], available from: http://www.omg.org/spec/UML/2.4.1

[20] H. Klus, A. Rausch, and D. Herrling, “DAiSI-dynamic adaptive system infrastructure: component model and decentralized configuration mechanism,“ in International Journal On Advances in Intelligent Systems, vol. 7, no. 3 & 4, 2014, pp. 595–608.

[21] H. Klus, A. Rausch, and D. Herrling, “Interface roles for dynamic adaptive systems,” in Proceedings of ADAPTIVE 2015, The Seventh International Conference on Adaptive and Self-Adaptive Systems and Applications, 2015, pp. 80–84.

[22] 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, pp. 7–12.

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

[24] H. Klus, “Anwendungsarchitektur-konforme konfiguration selbstorganisierender softwaresysteme,” translates to: Application architecture conform configuration of self-organizing softwaresystems, Clausthal-Zellerfeld, Technische Universität Clausthal, Department of Informatics, Dissertation, 2013.