Extending Interface Roles to Account for Quality of Service Aspects in the DAiSI

Dirk Herrling, Andreas Rausch and Karina Rehfeldt

Technical University Clausthal
Julius-Albert-Straße 4,
38678 Clausthal-Zellerfeld, Germany
email: dirk.herrling@tu-clausthal.de
andreas.rausch@tu-clausthal.de
karina.rehfeldt@tu-clausthal.de

Abstract—Dynamic adaptive systems are systems that change their behavior according to the needs of the user at run time. 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. To ensure a wanted, emergent behavior of the overall system, the components need to be wired according to the rules an application architecture defines. In this paper, we present the Dynamic Adaptive System Infrastructure (DAiSI). It provides a component model and configuration mechanism for dynamic adaptive systems. To address the issue of application architecture conform system configuration, we introduce interface roles that allow the consideration of component behavior during the composition of an application. Moreover, we extend the interface roles and application specifications by a quality of service concept. This enables a component to not only require a syntactical and semantical correct wiring, but also to demand the – from its viewpoint – best service.

Keywords—dynamic adaptive systems; component model; adaptation; interface roles; application architecture awareness.

I. INTRODUCTION

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

Software-based systems are present at all times in our daily life. This ranges from our private life where nearly everyone owns and uses a smart mobile phone to large scale business applications and the public administration that are managed entirely by software systems. In every household, dozens of devices run software and a modern car will not even start its engine without the proper software. Some software systems have grown to be among the most complex systems ever made by mankind [2], due to their increase in size and functionality.

Through smaller mobile devices with accurate sensors and actuators and the ubiquitous availability of the Internet, the number of integrated devices in large scale applications has increased drastically within the last twenty years. These devices and the software running on them are used in organically grown, heterogeneous, and dynamic information technology (IT) environments. Users expect them to not only provide their primary services, but also to collaborate with each other to provide some kind of emergent behavior. The challenge is therefore to be able to build systems that are robust enough to withstand changes in their environment, deal with a steadily increasing complexity, and match requirements that might be defined in the future [3].

Due to the increasing complexity of large systems, be it in size or in functionality, those systems are no longer developed from scratch by one vendor. While the development usually takes place in a component-based way [4], it is usually split among a number of companies. Additional components for mobile devices are often developed against documented or reverse-engineered interfaces by independent developers.

To ease the development of dynamically integratable components, a common component model is called for. The development of DAiSi started in 2004 to address this issue [5]. Over the years, a component model was defined that allows developers to implement a component for dynamic adaptive systems easily. In contrast to adaptive systems, which adapt to changes in their environment only, dynamic adaptive systems can also integrate components into themselves, which were not known at design-time. DAiSi provides a component model and run-time infrastructure for such dynamic adaptive systems. The latter can run and integrate DAiSi components by linking required services with compatible provided services and thus forming one ore more DAiSi applications. Compatibility has been only syntactical at first, requiring that for every method in the required service, a method with the same signature (name, parameters, return types, etc.) is defined in the corresponding provided service [6]. The aspect was later extended to support semantic compatibility by additionally requiring equivalent behavior of each method [7].

Obviously, an application is more than just the sum of its components. This already becomes evident in very small examples. Consider cross country skiers and their trainer. A dynamic adaptive application connects vital data monitoring devices of the athletes to the management system of their trainer. In a competition with a competing team on the same track, athletes and trainers should only be connected to each other if they belong to the same team. While it is possible to work around this issue by, e.g., ensuring in the implementation of components that they only exchange data if they belong to the same team, this is just that – a work around.

An application architecture that is enforced by the infrastructure can define rules that can address the challenge our athlete– and trainer-components face. It can specify that only components of members of the same team are allowed to be connected to each other. More generically, the consideration of an application architecture during system configuration helps
to ensure wanted, emergent behavior of dynamic adaptive systems. It does that by enabling application architects to limit the configuration space and thus prevent the connection of components that should not be connected. This paper will show the introduction of an application architecture into the field of dynamic adaptive system configuration and how we integrate it with the DAiSI infrastructure.

The rest of this paper is structured as follows: In Section II, we present an overview of other works in the field of dynamic adaptive systems. This is followed by an introduction to the DAiSI component model and the notation of DAiSI components in Section III. As a first step towards architecture conform configuration, we introduce interface roles and the consideration of local quality of service (QoS) aspects in Section IV. Afterwards, we briefly discuss the DAiSI configuration mechanism in Section V. In Sections VI and VII, an introduction of applications and application templates together with application-wide quality of service optimization follows. The paper ends with a conclusion in Section VIII.

II. RELATED WORK

Component-based development is one of the state-of-the-art techniques in modern software engineering. Components as units of deployment and their component frameworks provide a well-understood, solid approach for the development of large-scale systems. This is not surprising, considering that components can be added to, or removed from the system at design-time easily. This allows high flexibility and easy maintenance [4].

If components should be added to, or removed from a system at run-time things get a little bit more difficult, as techniques for this were not implemented in early component models. However, service oriented approaches allowed the dynamic integration of components at run-time. Those systems usually maintain a service directory. Components entering the system register their provided– and query for candidates of their required services. Once a suitable service provider is identified for a required service, it can be easily connected to the component [8].

Service-oriented approaches are capable of handling dynamic behavior. Components that have not necessarily been previously known to the system can be integrated into it. However, they have the uncomfortable characteristic that the system itself does not care for the dynamic adaptive behavior. The component needs to register and integrate itself. Also, it has to monitor itself whether the used services are still available and adapt its behavior accordingly, if that is no longer the case. To address these issues a couple of frameworks have been developed to support dynamic adaptive reconfiguration.

CONIC was one of the first frameworks for dynamic adaptive, distributed applications. It provided a description technique that could be used to change the structure (and thus the architecture) of the integrated modules of an application. A CONIC application was maintained by a centralized configuration manager [9]. It allowed to spawn new component instances and to link them to each other.

Another framework, building on the knowledge gained through the research in CONIC, was a framework for Reconfigurable and Extensible Parallel and Distributed Systems (REX). It provided support for dynamic reconfiguration in distributed, parallel systems. It visioned those systems as component instances, connected through interfaces for which an own interface description language was defined. Components were considered as types, allowing multiple instances of any component to be present at run-time. The framework allowed the dynamic change of the number of running instances and their wiring [10], [11]. Both, the CONIC– and REX framework allowed the dynamic adaptation of distributed applications, but only through explicit reconfiguration programs for every possible reconfiguration.

This issue was addressed in [12]. They took a more abstract approach and defined sets of valid application configurations. Following this approach, a system can then adapt itself from one valid application configuration to another, whenever the system changes. The declaration of reconfiguration steps became obsolete.

Another framework to build dynamic adaptive systems upon is ProAdapt. It is set in the field of service-oriented architectures and reacts to four classes of situations:

- Problems that stop the execution of the application
- Problems that require the execution of a non-optimal system configuration
- New requirements
- Presence of services with a better service quality

ProAdapt is capable of replacing certain services and can, together with its service composition capabilities, replace composed services [13].

In [14], [15], a framework for the dynamic reconfiguration of mobile applications on the basis of the .NET framework was introduced. Applications are composed of components, and application configurations are specified initially in XML. A centralized configuration manager interprets this specification and instantiates and connects the involved components. The specification can include numerous different configurations which are distinguished by conditions under which they apply. The framework monitors its surroundings with the help of a special Observer component and evaluates which application configuration is applicable. The framework allows the dynamic addition and removal of components and connections.

In [16], the authors present 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 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 their 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 [16] 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 [17] provided a different solution to ensure semantic compatibility of web services. However, the same arguments as for [16] regarding the absence of a high level component model hold true.

With regard to the application architecture aware adaptation, Rainbow [18], [19] 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 an operation for the adaptation of the system can be specified. The operation is then executed whenever the invariant is violated. However, this approach 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 [20] takes advantage of the features developed for centralized module management in the OSGi platform, like 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 DAISI COMPONENT MODEL

This section will introduce the foundations of the DAISI component model. As already briefly mentioned in the introduction of this paper, DAISI components communicate with each other through services. Different component configurations map required– to provided services.

As an example, we assume that a self-organizing system is to be developed, which supports the training of biathletes. A biathlon team consists of several athletes and trainers. Each trainer requires an overview of his athletes’ performance data, which includes the current pulse and skiing-technique of the athlete. Based on this data, the trainer can give guidance to his athletes. Figure 1 shows a sketch of a DAISI component with explanatory comments for an athlete in the biathlon sports domain.

A component is depicted as a rectangle, in this example of a light blue color. Component configurations are bars that extend over the borders of the component and are depicted in yellow here. Associated to component configurations are the provided and required services. The notation is similar to the Unified Markup Language (UML) lollipop notation [21] with full circles resembling provided, and semi circles representing required services. A filled circle indicates that the associated service is directly requested by the end user and thus should be provided, even if no other service requires its use.

Figure 1 shows the CAthlete component, consisting of two component configurations: conf1 and conf2. The first component configuration requires exactly one service variable r1 of the IPulse interface. The second component configuration does not require any services to be able to provide its service p1 of IPerson. The service could be used by any number of service users (the cardinality is specified as +). The other component configuration (conf2) could provide the service p1 of the type IAthlete, which could again be used by any number of users.

Figure 2 shows the DAISI component model as an UML class diagram [21]. The component itself, represented as the light blue box in the notation example, is represented by the DynamicAdaptiveComponent class. 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, e.g., because of best service qualities of the provided services. The order is defined by the component developer. 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. At run-time, only one or zero component configurations per component can be active. The active component configuration is represented by the current association in the component model, with the cardinality allowing one or zero current component configurations for each 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. Those that are resolved are represented by the resolved association. The cardinalities of the required service are stored in the attributes minNoOfRequiredRefs and maxNoOfRequiredRefs. Provided services (noted as full circles on the left hand side in Figure 1) are represented by the ProvidedService class. They can be associated to more than one component configuration, if more than one component configuration provides the same service. The runRequestedBy association is relevant at run-time and resembles the component configuration that actually wants the provided service to be produced.

Not all provided services can be used any number of times. The attribute maxNoUsers indicates the maximum number of allowed users. The flag requestRun, represented by the
Figure 2. DAiSI component model.
full circle being filled with black in the component notation, indicates that the service should be produced, even if no other service requires its use. This is typically the case for services that provide graphical user interfaces or that provide some functionality directly requested by the end user.

The provided and required service, 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. It does not already use the provided service, but would like to use it. 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 our last years paper [22].

To this point, we have covered the building blocks of a DAiSI component. An application in a dynamic adaptive environment is composed of any number of such components that are linked with each other through services. Those services are defined through DomainInterfaces. Required services (represented by the RequiredServiceReferenceSet class) refer to exactly one domain interface, while provided services (represented by the ProvidedService class) implement a domain interface. The set of all defined domain interfaces composes the DomainArchitecture. The interface roles, which will be presented in the next section, are contained by the domain architecture. They define domain interfaces and are required by any number of required service reference sets. Any provided service can conform to an interface role. However, this is not a static information, but changes during run-time.

For our example, it is assumed that the component presented in Figure 3 is available.

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 stick. 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). Additionally, it provides a method getLocation():double[2] to query the current location of the athlete. 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.

The component presented in Figure 4 represents the trainer.

The required functionality is provided by the service p_1, which implements the interface ITrainer. The service defines a dependency on services that implement the interface IAthlete. The individual athletes’ performance data within the application are provided by components providing this interface. However, ITrainer can also be run when no athlete is available in the system.

Shooting training is another requirement of our example system. Each shooting line is represented by a component, providing a service described by the domain interface IShootingLine. Figure 5 presents the DAiSI component CShootingLine.

The shooting line may be monitored by a supervisor. This is represented in the system by a service that implements the domain interface ISupervisor. The component presented in Figure 6 provides such a service. It can make use of a shooting line component.

All provided services of these components (trainer, shooting line, and supervisor) start, even if there is no other component requiring them. The flag requestRun is set, which is indicated by the filled circle.

### Interface Roles

With the RequiredServiceReferenceSet class many component local requirements can be specified. However, this...
is not sufficient for self-organizing systems. To illustrate the problem, let us consider Figure 7. It shows a simplified DAiSI component for an athlete. It specifies only one of the three configurations we defined in the previous section. The component provides a service of the type \textit{IAthlete} and requires two \textit{IStick} services to be able to do so. The provided service calculates the current skiing technique and needs measurement data of the sticks movements, which is provided by the two required services. However, with the component model as presented in Section III, a binding between only the left ski stick with both required service reference sets would be possible and allow the component to run. Of course the domain interface \textit{IStick} provides a method to query at which side a ski stick is being used. However, this information is not considered in the configuration process. Obviously, the \textit{IAthlete} service cannot perform as expected as the measurement data of the right ski stick is missing.

![Figure 7. A DAiSI component for a biathlon athlete.](image1)

There are numerous other examples in which return values of operations of domain services have to be considered in order to establish the desired system configuration. For that reason, we extended the component model by the class \textit{InterfaceRole}. In our previous understanding, provided and required services were compatible, if they referred to the same domain interface. Those interfaces can be seen as a contract between service provider and service user. We now extended this contract by interface roles. An interface role references exactly one domain interface and can define additional requirements regarding the return values of specific methods defined in that domain interface. A provided service only fulfills an interface role if it implements the domain interface and as well complies to 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.

Figure 8 shows the same DAiSI component as Figure 7, but with specified interface roles. With this addition it can be ensured that the athlete component in fact is connected with one left and one right stick. The \textit{LeftStickRole} interface role refines the \textit{IStick} domain interface and compares the return value of the method that returns the side of the ski stick is used on against a reference value for left ski sticks. This could be implemented by a method called \textit{getSide():String} and the return value would be compared against the string “left”. The interface role \textit{RightStickRole} can be implemented accordingly.

This solution introduces new challenges for the configuration process of dynamic adaptive systems. Was it previously sufficient to connect a pair of required service reference set and provided service, this decision has to be monitored now. As the interface roles take return values of services into account, the fulfillment of an interface role is not static. The provided service supposedly conforming to the interface role has to be evaluated either cyclically, or event based whenever relevant return values change. For our implementation, we took a cyclic approach, however, in [7] we described a way to re-evaluate the semantic compatibility of services whenever return values change equivalence classes.

The interface roles provide a possibility to include run-time information in the configuration process. But the true or false expressiveness of interface roles is not enough in some cases. Consider the following example: Every trainer is able to train three athletes at once. In addition, the energy consumption of the \textit{IAthlete} service depends on the distance between trainer and athlete. To maximize the outcome of the training and minimize the energy consumption each trainer should see and analyze performance data of the three nearest athletes instead of the ones miles away. In Figure 9 the trainer should see the performance data of athletes A, B and C.

![Figure 8. A DAiSI component for a biathlon athlete with interface roles.](image2)

![Figure 9. Locale optimization of energy consumption and training outcome.](image3)

The DAiSI component model and domain architecture presented above are not sufficient to express a constraint like this. There are other examples when the services referenced in \textit{RequiredServiceReferenceSets} should be ordered by requirements of the component.

To achieve this, we expand the interface role by a comparator. The method \textit{compare(DomainInterface \textit{ps}1, DomainInterface \textit{ps}2, DomainInterface \textit{req}) : int} takes three domain interfaces as parameter. \textit{ps}1 and \textit{ps}2 are provided services which will be compared. The last domain interface \textit{req} is an optional parameter, which may be used to take run-time information of the requiring component into account. It may be a provided service of the requiring component, which provides important run-time information for the comparator. In our example of trainer and athletes, the current position of the trainer is crucial for the ordering of \textit{IAthlete} services.

The compare method returns either a negative integer, zero
or a positive integer. In the manner of known comparators, like Java’s comparator, a negative value, zero, or a positive value mean the quality of \( ps_1 \) is less than, equal or greater than the quality of \( ps_2 \). Whether there are finer granularities in return values in the sense of -1000 is much worse than -1 depends on the particular implementation of the compare–method. It is important to notice that only provided services which are conform to the interface role are comparable. The standard implementation for the compare method treats every service as equal and returns always zero.

Now, the RequiredServiceReferenceSet of a component has the possibility to order the provided services by a chosen criterion. The available criteria are determined by the domain. Consequently, the configuration process has to take the order into account when configuring the system. All possibly usable services are represented by canUse. These are the services which implement the domain interface and which are conform to a required interface role. If there is a request for dependency resolution, the services in canUse are ordered with the help of compare. The resulting list of ordered services is then used to place a usage request for the currently best services. Their service references are copied to the wantsUse set. The configuration mechanism will be explained in more detail in the next section.

Consider again the biathlon example. Assume a given CTrainer component as shown in Figure 11. It has one single

Figure 10 shows our example from before with ordered components. The interface role NearestAthlete is now required by \( t \), the CTrainer component. The compare method of NearestAthlete compares two IAthlete services with the help of one ITrainer service. Both IAthlete and ITrainer provide a method getLocation():double[2], which returns the current GPS coordinates of the athlete or trainer. The compare method of NearestAthlete compares the distance of given athletes to the trainer. Of course, regarding our example a shorter distance is considered better, as a longer distance. To fulfill the specified meaning of the compare method’s return value, \( ps_2 \)’s distance minus \( ps_1 \)’s distance is returned.

This results in the situation pictured in Figure 10. \( t \) can use all CAthlete components but in the end it uses \( b, a, \) and \( c \), since this is the resulting order from sorting by distance to \( t \). Concluding, the interface role comparator provides a rather simple method for the component to state which services it would like to use “the most”.

V. Configuration Mechanism

Besides the DAiSI component model and the DAiSI domain architecture model, a decentralized dynamic configuration mechanism was also already established in the DAiSI platform. Three types of relations between RequiredServiceReferenceSets and ProvidedServices exist, 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. Interface roles in addition provide a compatibility check with respect to a given common domain architecture.

The wantsUse set holds references to those services for which a usage request has been placed. Last, 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, it is added to all canUse sets, if the corresponding RequiredServiceReferenceSet refers to the same DomainInterface as the 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 relies on the state machines, presented more detailed in [23] and sketched in the following paragraphs.

Consider again the biathlon example. Assume a given CTrainer component as shown in Figure 11. It has one single
configuration and 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 users. 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 13. CAthlete and CPulse components.](image)

Now assume two components: The CAthlete component, shown on the left hand side of Figure 13, requires zero or one reference to a service of type IPulse. The second component, CPulse, shown on the right hand side of Figure 13, provides such a service of type IPulse. Figure 14 shows the states and triggers of the involved state machines in a sequence diagram for this example.

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 usable service is available. DAiSI indicates that CAthlete wants to use this new service by adding this service in the set wantsUse of CAthlete. Once the service runs, it is assigned to the CAthlete component, which uses the service from now on (added to the set uses of CAthlete).

![Figure 14. Inter-component configuration mechanism.](image)

### VI. APPLICATION SPECIFIC SYSTEM CONFIGURATION

Imagine the following situation: the example biathlon application already introduced shall now fulfill a number of new requirements:

- The ITrainer service can only be run when it has access to at least one athlete service
- Each athlete must have access to a shooting line
- Shooting lines can only be used when they are supervised
- The skiing technique is to be analyzed in particular

Currently, the implementation of the components would have to be adapted in order to meet the requirements. For example, the attribute minNoOfRequiredRefs of \( r_1 \) from Figure 11 would have to be set to 1. However, a component’s code cannot always be adapted in this way, for example because it is proprietary software and the source code not available. In addition, adapting it manually for the specific application purpose contradicts one of the original purposes of component based software development – re-use of components among different applications. 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’s source code.

Since the skiing technique is to be analyzed in particular, only the \( \text{conf}_2 \) ComponentConfiguration of the athlete component ‘tim’ of Figure 3 is relevant. Even if one pulse service and two ski stick services are available, the \( \text{conf}_1 \) configuration should not be activated. In this section, expansions of the existing framework are described, which enable such an application-specific influence on the activation of component configurations.

The system must guarantee that exactly one shooting line 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 accordance with the athlete components, which the trainer component accesses. One system configuration that meets all criteria described above is presented in Figure 15. A trainer component is connected with an athlete component, which in turn is connected to a left and a right ski stick. In addition, the application consists of a shooting supervisor component, which in turn is connected to a shooting line component.

In the DAiSI as it was presented on the previous pages, such system configuration requirements cannot be specified and, therefore, not 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 line component is available for this athlete. The application also needs to be stopped for example, when the athlete component from Figure 15 is only connected with one ski stick component.

To address these issues, we introduce application configurations. An application configuration consists of a number of components, as well as connections between these components. 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 criteria for the 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 line components, etc. For each of these templates, requirements can be stored that specify under which circumstances a component is compatible with a template. The framework ensures that at runtime, only components matching the outline are allocated to the template. Graphically, a template is 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 16, 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 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 16). 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 \( t_{\text{Trainer}} \) template in Figure 16 must be connected with at least one component, which is allocated to the \( t_{\text{Athlete}} \) template. Later (during runtime), the framework ensures that the requirements related to the links between the components are considered. Figure 17 shows one possible resulting system configuration, while Figure 18 presents a possible application specification for the complete biathlon application.

An application itself is graphically represented as a rectangle with the name of the application noted at the top. As stated before, each Template is represented as a rectangle with dashed outer lines and identified by a name. Within a template, the contents of the attributes minNoOfRequiredComponents and maxNoOfRequiredComponents are noted at the top.
right. A **ProvidedTemplateInterface** is represented as a dashed circle, which is labeled with its name, and the referenced domain interface. **RequiredTemplateInterfaces** are represented correspondingly as dashed semi-circles. They are also labeled with the referenced domain interface, the referenced interface role, if applicable, and a name. Links between **RequiredTemplateInterfaces** and **ProvidedTemplateInterfaces** are visualized with a dashed arrow. The predicate specification (`appConstraint`) is specified in a separate area beneath the templates.

The aim of the DAiSI run–time infrastructure 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.

The following paragraphs describe 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 sketched in Listing 1 as pseudo code. While this is not optimal with regard to resources, it is sufficient to generate a valid system configuration.

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

Listing 1. `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 cyclically for conformance to the application architecture. 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 algorithm is divided into two parts: the first part creates an application configuration (lines 2-9 in Listing 1), while the second checks the generated configuration for conformity with the requirements (lines 11-12 in Listing 1). 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 `RequiredServiceReferenceSet`.

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, an assignment of the `selectedComponents` set is created for each template, with the static properties already being considered. The configuration mechanism then calculates the set of all possible assignment combinations and makes them available via an iterator (`possibleComponentAssignmentSets` from Listing 1), based on the components available and the application specification, the method `realize` implements the specific assignment.

![Figure 19. Allocation of components to templates.](image)

In the example in Figure 19, the components `a` and `b` can be allocated to the `tTrainer` template. Only one component needs to be allocated to the template in order to fulfill the application requirements. Both components provide a service that implements the `tTrainer` domain interface and define a `RequiredServiceReferenceSet` that references the `tAthlete` domain interface. Only component `d` can be allocated to the `tAthlete` template since this component is the only one that meets the structural requirements of the template. A total of two components are available for the `tStick` and `tStick` templates and exactly one component must be allocated to each of these, 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 realized by the configuration mechanism.

`ProvidedServices` of a component can fit to several `ProvidedTemplateInterfaces`. Since `ProvidedServices` must be allocated to `ProvidedTemplateInterfaces` during run–time, the framework needs to decide which to use. This applies accordingly to `RequiredServiceReferenceSets` and `RequiredTemplateInterfaces`. For example, the `RequiredTemplateInterface` of the `tAthlete` template in Figure 19, do not reference any interface roles but only the `tStick` domain interface as presented in Figure 20. In this example, the `RequiredServiceReferenceSet r1` can be allocated to `RequiredTemplateInterface rtA1` as well as `rtA2`. The same applies to `RequiredServiceReferenceSet r2`.

Within the algorithm in Listing 1, all possible allocation configurations, which result from the allocation of components to templates in the previous step are now iterated. In the
In this section, we present an extension to the previously introduced application configuration, which allows to define such global optimization criteria. The application from DAiSI's component model is extended by a compare method. The method compareTo(Application a):int compares the current application with the application given by a. As before, the return value is either negative, zero or positive meaning the current application’s quality is less than, equal to, or greater than a’s quality. The method is implemented by the application (and therefore the person defining the application template). The standard implementation treats all applications as equal.

The biathlon application for minimized energy consumption is described by the application specification seen in Figure 22. Some parts like the pulse or stick services are missing but it shows the most important components. Listing 2 shows an implementation of compareTo for our desired energy minimization. Firstly, the overall distance between trainers and athletes in the current application is calculated. For all chosen CTrainer components in the template tTrainer the distance to its used IAthlete services is summed up. Afterwards, the same is done for the given application. The return value is the resulting difference between the compared and current application’s sum of distances.

Listing 2. compareTo method for biathlon application with minimized energy consumption, pseudo code listing.

```java
int compareTo(Application a) {
    distanceCurrent = 0;
    while (tTrainer.selectedComponents.hasNext()) {
        CTrainer t = tTrainer.selectedComponents.next();
        while (t.declares.iterates.hasNext()) {
            distance += t.declares.iterates.next().getLocation().distanceTo(t.getLocation());
        }
    }
    distanceOther = 0;
    while (a.tTrainer.selectedComponents.hasNext()) {
        CTrainer t = a.tTrainer.selectedComponents.next();
        while (t.declares.iterates.hasNext()) {
            distance += t.declares.iterates.next().getLocation().distanceTo(t.getLocation());
        }
    }
    return distanceOther - distanceCurrent;
}
```

Listing 2. compareTo method for biathlon application with minimized energy consumption, pseudo code listing.

Remember the createValidConfiguration() method from Listing 1. We will extend this algorithm with the compareTo method. Instead of realizing and accepting the first valid configuration, the algorithm will compare all possible valid configurations and realize the best one. Of course, this brute-force approach is not practical. The testing of sophisticated

In this section, we present an extension to the previously introduced application configuration, which allows to define such global optimization criteria. The application from DAiSI's component model is extended by a compare method. The
configuration algorithms and determining suitable heuristics are future work.

The extended configuration algorithm is sketched in Listing 3. It realizes the behavior described before. It is important to notice that lines 19 and 27 hide more complicated technical aspects. In line 19, the currently realized application configuration is saved. This could be done by saving all realized components, interfaces and usages, i.e., connections between components and services. This saved configuration is realized again in line 27.

```java
boolean createValidConfiguration() {
    Application best = null;
    while (possibleComponentAssignmentSets.hasNext()) {
        possibleComponentAssignmentSets.next().realize();
    }
    while (possibleInterfaceAssignmentSets.hasNext()) {
        possibleInterfaceAssignmentSets.next().realize();
    }
    while (possibleUsageSets.hasNext()) {
        possibleUsageSets.next().realize();
    }
    if (isValidConfiguration()) {
        if (compareTo(best) > 0) {
            best = this;
        }
    }
    if (best != null) {
        best.realize();
        return true;
    }
    return false;
}
```

Listing 3. createValidConfiguration() method expanded by usage of compareTo method, pseudo code listing.

This concludes the section about application specific quality criteria. The `compareTo` provides a tool to the application designer, which allows him to specify which (of a set of syntactically and semantically correct) application configuration is considered “best” and therefore should be realized. Since this depends on run–time information the configuration has to be checked whenever run–time information changes. Like before, a cyclical or more advanced approach could be taken.

VIII. CONCLUSION

This paper presented an extended version of the DAiSI framework. While the system configuration, more precisely the component wiring, in older versions of DAiSI and other dynamic adaptive system infrastructures were only considering syntactic and semantic compatibility, the newest findings enable developers to specify interface roles and application templates. These open the possibility to define local and application–wide constraints on the configuration.

We introduced a concept, which takes quality and service aspects into account. A service comparator defined on interface roles enables components to define not only which semantic domain interface they require, but also which quality criteria they prefer. Global quality of service is achieved on the application level.

Currently, DAiSI only supports service–oriented architectures. Upcoming technologies and paradigms, like the Internet of Things (IoT) or cyber–physical systems demand for other
improvements, like pub/sub and message–based communication. We will extend concept and implementation of the DAiSI to account for these developments.

However, the extension presented in this paper provides a sustainable concept for the realization of decentralized, dynamic adaptive systems, while considering quality of service aspects.

REFERENCES