Interface Roles for Dynamic Adaptive Systems

Holger Klus

ROSEN Technology & Research Center GmbH
Am Seitenkanal 8
49811 Lingen (Ems), Germany
email: hklus@rosen-group.com

Dirk Herrling and Andreas Rausch

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

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

I. Introduction

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 is 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 [1], 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 a large scale application 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 not only to provide their primary services, but also to collaborate with each other and 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 [2].

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 company. While the development usually takes place in a component-based way [3], 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 integrateable components, a common component model is called for. The development of the DAiSI started in 2004 to address this issue [4]. Over the years a component model was defined that allows developers to implement a component for a dynamic adaptive system easily. In this DAiSI component model, every component contains an ordered set of component configurations which each map a set of required services to a set of provided services.

Additionally, a run-time infrastructure was described and implemented that can run and integrate DAiSI components by linking required services with compatible provided service 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 [5]. The aspect was later extended to support semantic compatibility by additionally requiring equivalent behavior of each method [6].

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 track, obviously not every athlete should be connected to every trainer. Also, the connection should not be made randomly. Each athlete should only be connected to the trainer that belongs to the same team. While it is possible to work around this issue by, e.g., ensuring in the implementation of the component that only athletes exchange data with trainers from 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 problem our athletes and trainers have. It can specify that only components of members of the same team are allowed to be bound 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 a first step towards the introduction of an application architecture into the field of dynamic adaptive systems and how we integrate it with the DAiSI infrastructure.

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-391-9

The rest of this paper is structured as follows: In Section II, we will 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 in Section IV. The paper ends with a conclusion in Section V.

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 [3].

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 and components entering the system register their provided, and query their required services at the directory. Once a suitable service provider is found for a required service, it can be easily connected to the component [7].

Service-oriented approaches are capable of handling dynamic behavior. Components that have not necessarily been previously known to the system can be integrated into the system. 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 if 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. It was maintained through a central configuration manager [8]. With this description techniques, new component instances could be spawned and linked to each other.

Another framework, building on the knowledge gained through the CONIC development, 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 connected component instances with 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 [9], [10]. Both, the CONIC and the REX framework allowed the dynamic adaptation of distributed applications, but only through explicit reconfiguration programs for every possible occurring change.

This issue was addressed in [11]. They took a more abstract approach and defined valid application configurations. The 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
- Arising of new requirements
- Providing 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 [12].

In [13], [14], 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 through 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 [15] 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 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 [15] 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 [16] provided a different solution to ensure semantic compatibility of web services. However, the same arguments as for [15] regarding the absence of a high level component model hold true.

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 which required services are needed by the associated provided services. Figure 1 shows a sketch of a DAiSI component with some explanatory comments for an athlete in the biathlon sports domain.

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-391-9

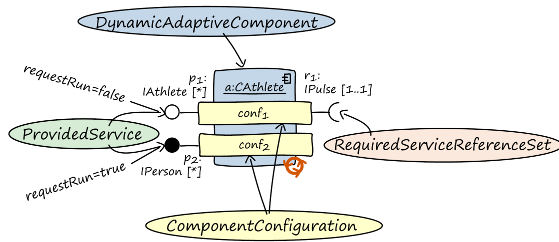

Figure 1. Example notation of a DAiSI component with explanatory comments.

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 the component configurations are the provided and required services. The notation is similar to the Unified Markup Language (UML) lollipop notation [17] 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: 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 could 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.

Figure 2 shows the DAiSI component model as an UML class diagram [17]. 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

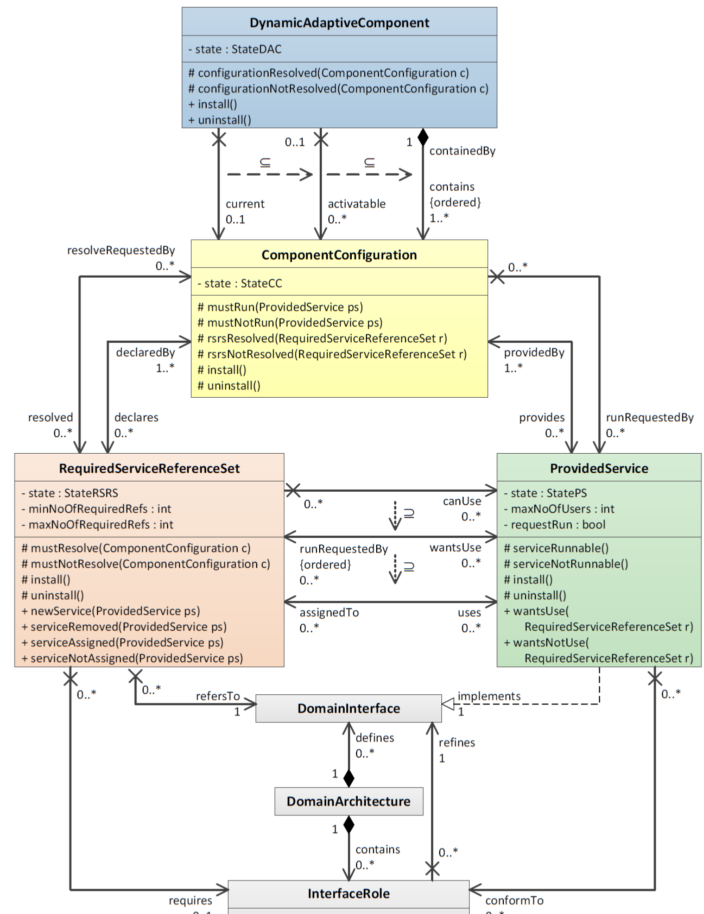

Figure 2. DAiSI component model.

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 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 [18].

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 refine 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. The next section will explain why.

IV. 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 3. It shows a DAiSI component for an athlete in the biathlon sports domain. It does specify one component configuration which provides a service of the type *IAthlete* and requires two *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 the previous paragraphs 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 *IStick* does provide 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 *IAthlete* service can not perform as expected as the measurement data of the right ski stick are missing.

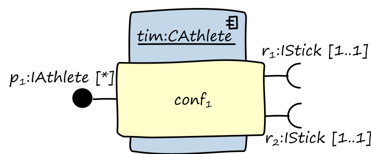

Figure 3. A DAiSI component for a biathlon athlete.

There are numerous other examples in which return values 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 *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 4 shows the same DAiSI component as Figure 3, 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 *LeftStickRole* interface role refines the *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 *getSide():String* and the return value would be compared against the string “left”. The interface role *RightStickRole* can be implemented accordingly.

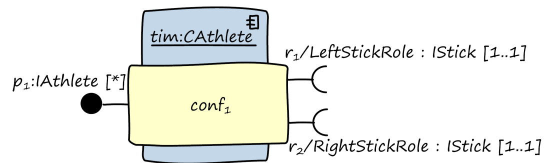

Figure 4. A DAiSI component for a biathlon athlete with interface roles.

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 [6] we described a way to re-evaluate the semantic compatibility of services whenever return values change equivalence classes.

V. Conclusion

This paper presented an extended version of the DAiSI framework. The key aspect that the developer does not need to implement the adaption behavior himself, has been prevailed. While the system configuration, more precisely the component wiring, in older versions of DAiSI and other dynamic adaptive systems was only considering syntactic and semantic compatibility, the newest findings enable the developer to specify interface roles. These open the possibility to consider return values of services during system configuration, which was not possible before. We implemented the framework in Java and a slightly limited version in C++. Components of both framework implementations can be linked together, because of an underlying CORBA layer.

Interface roles are obviously just the first step towards application architecture conform system configuration. In the near future, we will extend the approach to support an architecture description that allows the specification of constraints that are not component local, as the interface roles are.

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-391-9

REFERENCES

[1] L. Northrop, P. Feiler, R. P. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, D. Schmidt, K. Sullivan, and K. Wallnau, "Ultra-Large-Scale Systems - The Software Challenge of the Future," Software Engineering Institute, Carnegie Mellon, Tech. Rep., June 2006. [Online]. Available: http://www.sei.cmu.edu/uls/downloads.html

[2] 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.

[3] C. Szyperski, Component Software: Beyond Object-Oriented Programming, 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[4] D. Niebuhr, C. Peper, and A. Rausch, "Towards a development approach for dynamic-integrative systems," in Proceedings of the Workshop for Building Software for Pervasive Computing, 19th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), Nov 2004. [Online]. Available: http://sse-world.de/index.php/download_file/view_inline/157/

[5] H. Klus, D. Niebuhr, and A. Rausch, "A component model for dynamic adaptive systems," in Proceedings of the International Workshop on Engineering of software services for pervasive environments (ESSPE 2007), A. L. Wolf, Ed. Dubrovnik, Croatia: ACM, sep 2007, pp. 21–28, electronic Proceedings. [Online]. Available: http://sse-world.de/index.php/download_file/view_inline/79/

[6] D. Niebuhr, Dependable Dynamic Adaptive Systems. Verlag Dr. Hut, 2010. [Online]. Available: http://www.dr.hut-verlag.de/9783868536706.html

[7] M. P. Papazoglou, "Service-oriented computing: Concepts, characteristics and directions," in Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on. IEEE, 2003, pp. 3–12.

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

[9] J. Kramer, "Configuration programming-a framework for the development of distributable systems," in CompEuro'90. Proceedings of the 1990 IEEE International Conference on Computer Systems and Software Engineering. IEEE, 1990, pp. 374–384.

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

[11] I. Warren and I. Sommerville, "Dynamic configuration abstraction," in Software EngineeringESEC'95. Springer, 1995, pp. 173–190.

[12] R. R. Aschoff and A. Zisman, "Proactive adaptation of service composition," in Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012 ICSE Workshop on. IEEE, 2012, pp. 1–10.

[13] A. Rasche and A. Polze, "Configurable services for mobile users," in Object-Oriented Real-Time Dependable Systems, 2002.(WORDS 2002). Proceedings of the Seventh International Workshop on. IEEE, 2002, pp. 163–170.

[14] ——, "Configuration and dynamic reconfiguration of component-based applications with microsoft. net," in Object-Oriented Real-Time Distributed Computing, 2003. Sixth IEEE International Symposium on. IEEE, 2003, pp. 164–171.

[15] 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. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-30475-3_52

[16] T. Haselwanter, P. Kotinurmi, M. Moran, T. Vitvar, and M. Zaremba, "Wsmx: A semantic service oriented middleware for," in B2B Integration, International Conference on Service-Oriented Computing. Springer, 2006, pp. 4–7.

[17] OMG, OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1, Object Management Group Std., Rev. 2.4.1, August 2011. [Online]. Available: http://www.omg.org/spec/UML/2.4.1

[18] H. Klus and A. Rausch, "Daisi - a component model and decentralized configuration mechanism for dynamic adaptive systems," in ADAPTIVE 2014, The Sixth International Conference on Adaptive and Self-Adaptive Systems and Applications, 2014, pp. 27–36.