DAiSI—A Component Model and Decentralized Configuration Mechanism for Dynamic Adaptive Systems

Holger Klus
Technische Universität Clausthal
Clausthal-Zellerfeld, Germany
holger.klus@tu-clausthal.de

Andreas Rausch
Technische Universität Clausthal
Clausthal-Zellerfeld, Germany
andreas.rausch@tu-clausthal.de

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

Keywords-dynamic adaptive systems; component model; component composition; adaptation; componentware; component container; 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 now 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 value. The challenge is therefore to provide software systems that are robust in the presence of increasing challenges such as change and complexity [2].

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

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

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

Self-managed systems are those capable of adapting to the current context as required though self-configuration, self-healing, self-monitoring, self-tuning, and so on. These are also referred to as self-x, autonomic systems. We call them dynamic adaptive systems.

Providing dynamic adaptive systems is a great challenge in software engineering [2]. 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 2. In our research group we have also developed a framework for dynamic adaptive (and distributed) systems, called DAiSI

 ISBN: 978-1-61208-341-4

(Dynamic Adaptive System Infrastructure). The first version of DAiSI was implemented and published in 2006/07 [15], [10], [14] , [11]. Based on the DAiSI framework a couple of dynamic adaptive systems (research and industrial demonstrators) were developed and evaluated within the following domains: assisted sport training systems [3], emergency management systems [7], [9], assisted living systems for elderly people [8], [10], intelligent beer dispensing systems [5], [6], and airport baggage management system [12], [13], [11]. All of these systems were exhibited at CeBIT, such as [4]. Some of them were successfully transformed into products, for instance [5] and [6].

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

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

The rest of the paper is structured as follows: After a short description of the related work we provide an overview of the DAiSI framework. In the following three subsections we will introduce DAiSI's main essential: a domain model, an adaptive component model, and a decentralized dynamic configuration mechanism. Then we describe a small sample application to illustrate the decentralized dynamic configuration mechanism of the adaptive components. 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. Components are the units of deployment and integration. During design time components may be added or removed from a system [16].

However, dynamic changes, e.g. adding or removing components from a system during run time is not direct support. Service-oriented approaches promise a more flexible approach for dynamic changes. Service users query for services within a service directory. Once they have found the corresponding service they can dynamically connect themselves to the service [17].

Unfortunately in service-oriented approaches the components are responsible for the dynamic adaptive behavior. They have to query for the proper services, verify that the services fit the ones they are looking for and connect themselves to the corresponding services. For that reason a couple of frameworks have been developed. Those frameworks support the component configuration during run time and thereby form dynamic adaptive systems. CONIC and REX provide a description technique to describe an initial system configuration and system adaptions during run time [18], [19].

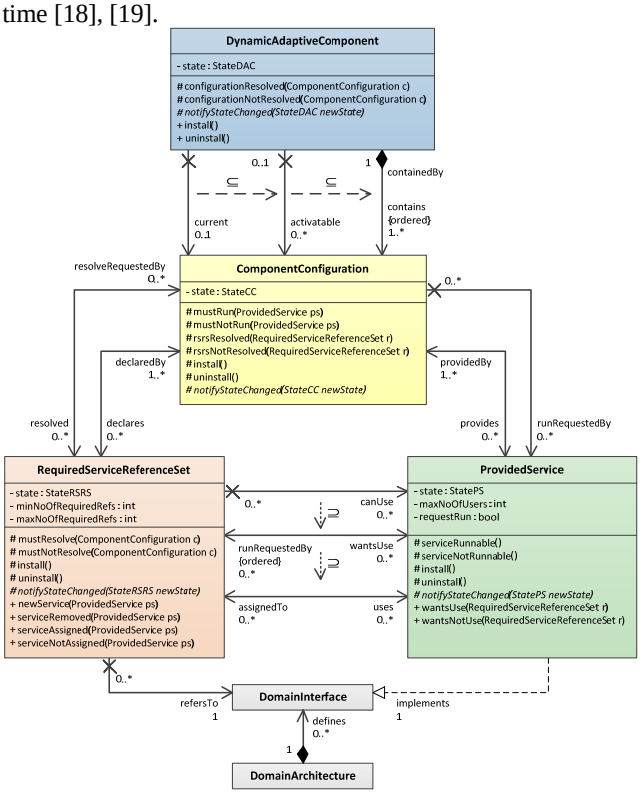


Figure 1. Core elements of the DAiSI framework.

Current frameworks such as ProAdapt [20] and Config.NETServices [21] have a more generic adaption and configuration mechanism. Components that were not known during the design-time of the system, are added and removed from the dynamic adaptive system during run time. Therefore a generic component configuration mechanism is provided by the framework. As with our first version of the DAiSI framework, these frameworks are based on a centralized configuration mechanism. Moreover the underlying component model is restricted—for instance the exclusive usage of services cannot be described.

III. DAISI – DYNAMIC ADAPTIVE SYSTEM INFRASTRUCTURE

Our approach for self-organizing systems is based on a specific framework called DAiSI [15], [10], [14] , [11]. DAiSI consists of three main parts or elements: a domain model, an adaptive component model, and a decentralized dynamic configuration mechanism. All three will be introduced at a glance in the following section. The three elements and their relationship to each other are depicted in Figure 1 using a UML class diagram. Note, a complete description of the DAiSI framework can be found in [22].

 ISBN: 978-1-61208-341-4

A. Domain Model

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

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

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

Domain interfaces contain syntactical information like method signatures or datatypes occuring in the interfaces. In addition they may also contain a behavioral specification of the interface following the design by contract approach, for instance using pre- and postconditions and invariants to describe the functional behavior of a domain interface [9].

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

Using simple domain interface descriptions the correctness of the binding can only be guaranteed on a syntactical level. Once the domain interface descriptions contain additional information about the functional behavior, the correctness of the binding can also be guaranteed on the behavioral level. Therefore we have developed a sophistic approach based on run-time testing. Further information of DAiSI's solution to guarantee functional correctness of dynamic adaptive systems during run time can be found in [9], [23].

B. Adaptive Component Model

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

1) Dynamic Adaptive components

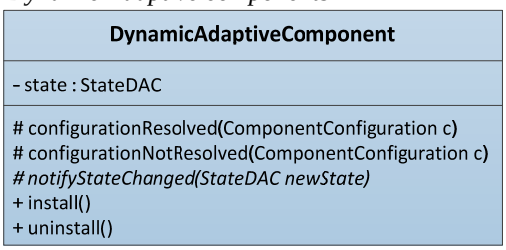


Figure 2. DynamicAdaptiveComponent class.

Each component instance within the system is represented by an instance of the class DynamicAdaptiveComponent, see Figure 2. By calling the install or uninstall methods, a component is, respectively, published or removed from the system. If install is called, all other parts of that component are informed by calling the trigger install. The framework then starts trying to resolve dependencies on other components in order to run ProvidedServices and provide them to other components within the system. Each DynamicAdaptiveComponent realizes a state machine, as shown in Figure 3 whose current state is stored in a variable called state.

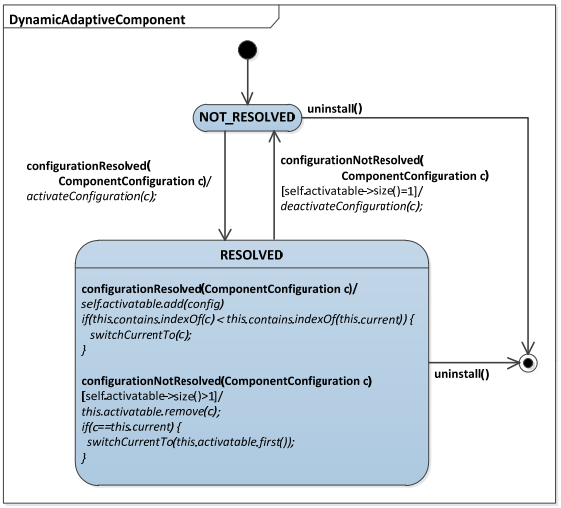


Figure 3. State machine - DynamicAdaptiveComponent class.

Two states are distinguished for *DynamicAdaptiveComponent*, namely RESOLVED and NOT_RESOLVED. In the beginning a component is in the NOT_RESOLVED state. If, for a single *ComponentConfiguration*, all dependencies to services of other components are resolved, the trigger *configurationResolved* of *DynamicAdaptiveComponent* is called and the state machine switches to state RESOLVED.

Every time a state transition takes place, the abstract method, *notifyStateChanged*, is called. A component developer can override this method in order to react to certain state transitions, e.g. by showing or fading out a graphical user interface.

2) Component Configuration

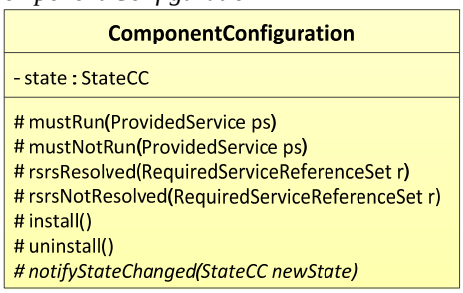


Figure 4. ComponentConfiguration class.

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

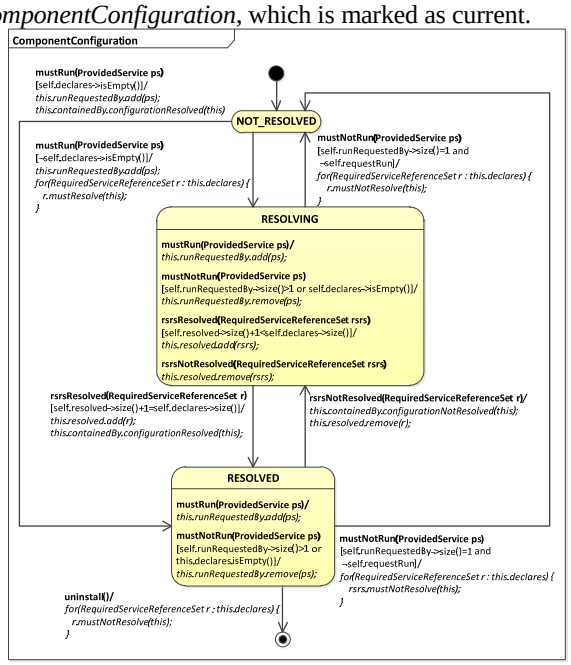


Figure 5. State machine - ComponentConfiguration class.

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

3) Provided Service

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

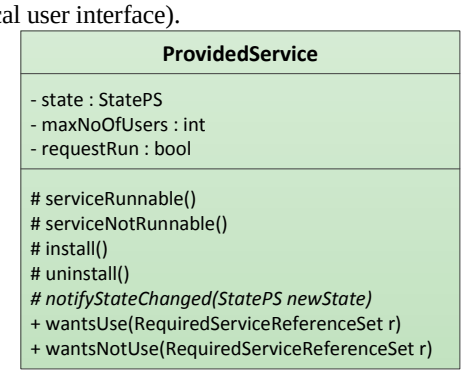


Figure 6. ProvidedService class.

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

marked as *activatable*. However, a service is in RUNNING state if it is connected to a *ComponentConfiguration* which is marked as *current*. If a *ComponentConfiguration* becomes current, all connected *ProvidedServices* are informed by calling the *serviceRunnable* trigger.

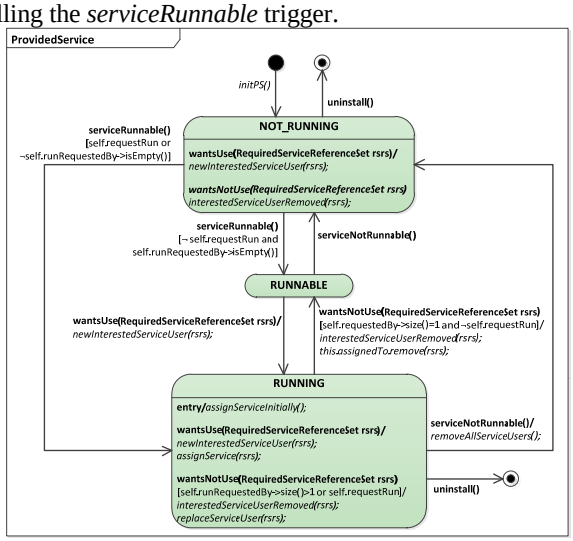


Figure 7. State machine - ProvidedService class.

4) Required Service Reference Set

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

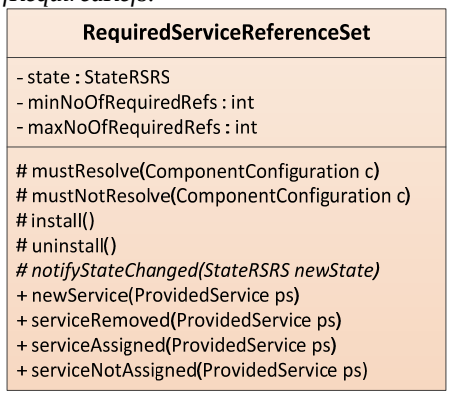


Figure 8. RequiredServiceReferenceSet class.

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

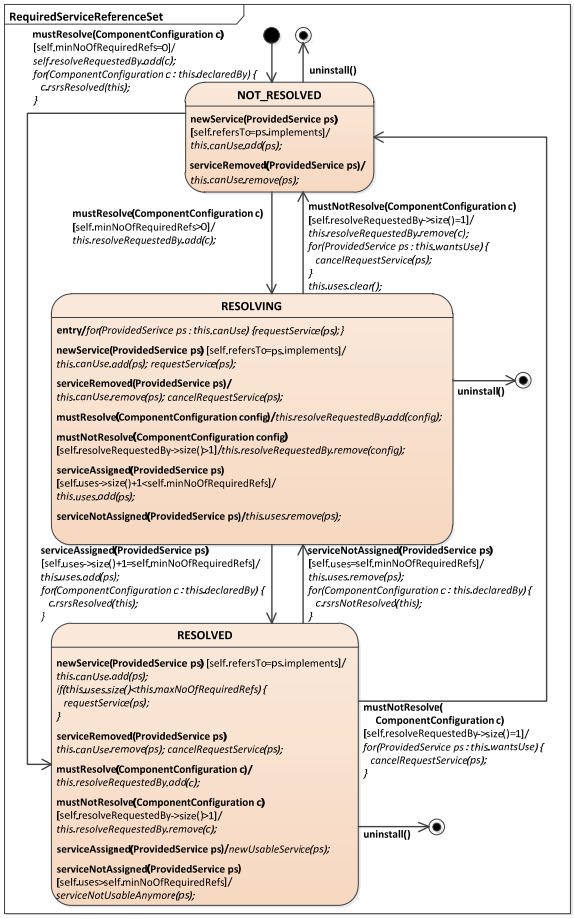


Figure 9. State machine - RequiredServiceReferenceSet class.

5) Notation for DAiSI Components

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

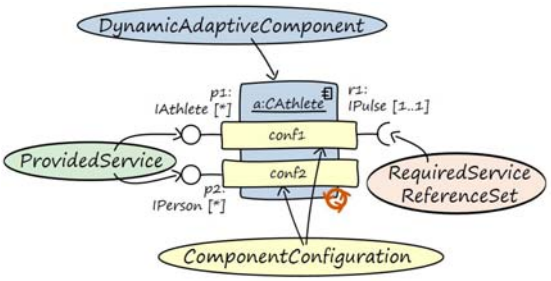


Figure 10. Notation for DAiSI components.

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

C. Decentralized Dynamic Configuration Mechanism

There exist three types of relations between *RequiredServiceReferenceSets* and *ProvidedServices*, represented by the associations *canUse*, *wantsUse* and uses. The set of services that implement the domain interface referred by the *RequiredServiceReferenceSet* is represented by *canUse*. Note, this only guarantees a syntactically correct binding. In [9] and [23] we have shown how this approach can be extended to guarantee functional-behaviorally correct binding as well during run time using a run-time testing approach.

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

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

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

1) Local Configuration Mechanism

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

Figure 11. CTrainer component.

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

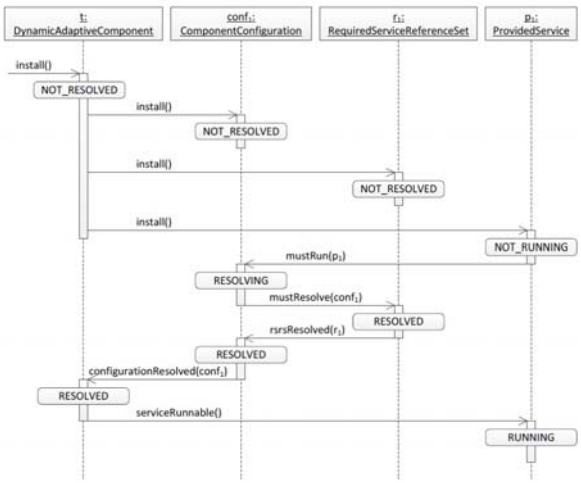


Figure 12. Local configuration mechanism component.

2) Inter-Component Configuration Mechanism

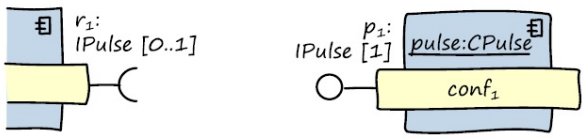


Figure 13. CAthlete and CPulse components.

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

 ISBN: 978-1-61208-341-4

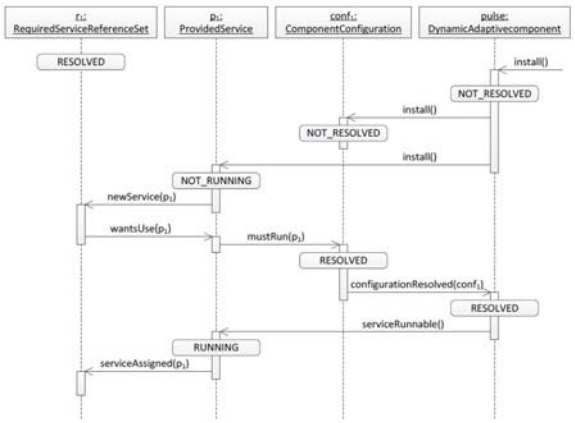


Figure 14. Inter-component configuration mechanism.

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

IV. SAMPLE APPLICATION – SMART BIATHLON TRAINING SYSTEM

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

A. Domain Model

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

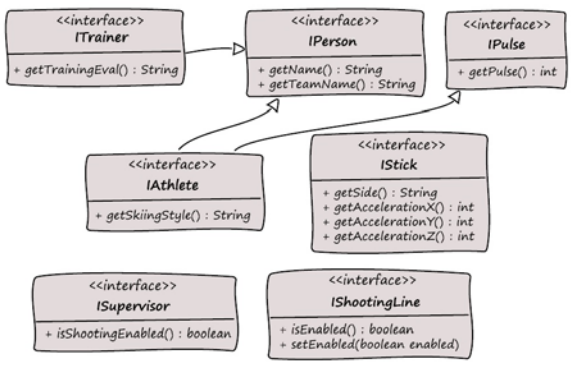


Figure 15. Domain model - "Smart Biathlon Training System".

B. Available Components

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

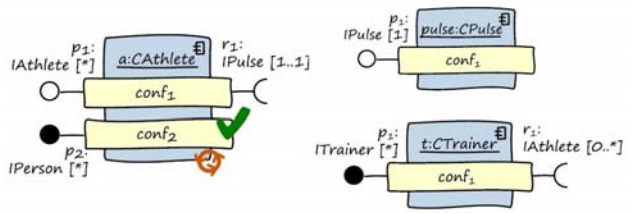


Figure 16. Adaptive components: CPulse, CAthlete, CTrainer.

The *CPulse* component provides an exclusive usable service *IPulse* and requires no other services from the dynamic adaptive system. The *CAthlete* component provides two services: *IPerson* and *IAthlete*. In $conf_2$ it provides the service, *IPerson*, which has the flag, *requestRun*, and requires no service from the environment. In conf1 it provides the service, *IAthlete*, but therefore requires a service, *IPulse*. And finally the *CTrainer* component may supervise an arbitrary number of athletes and thus provides a corresponding number of *ITrainer* interfaces to the real trainer, supporting him with the online training information of the supervised athletes.

C. Decentralized Dynamic Configuration Mechanism

Assume the following situation in the dynamic adaptive system. The component, *CPulse*, is activated and the component, *CAthlete*, is activated, see Figure 17 As the *requestRun* flag of the provided service of $conf_2$ is set and no additional service references are needed, this configuration is activated and the service is provided within the dynamic adaptive system.

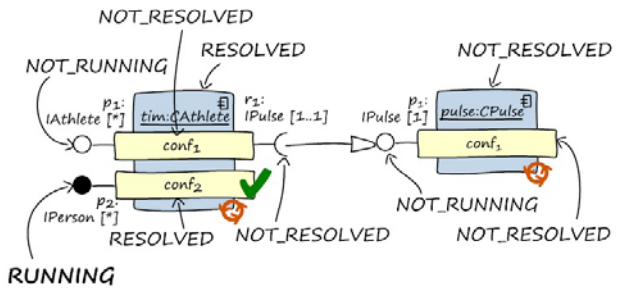
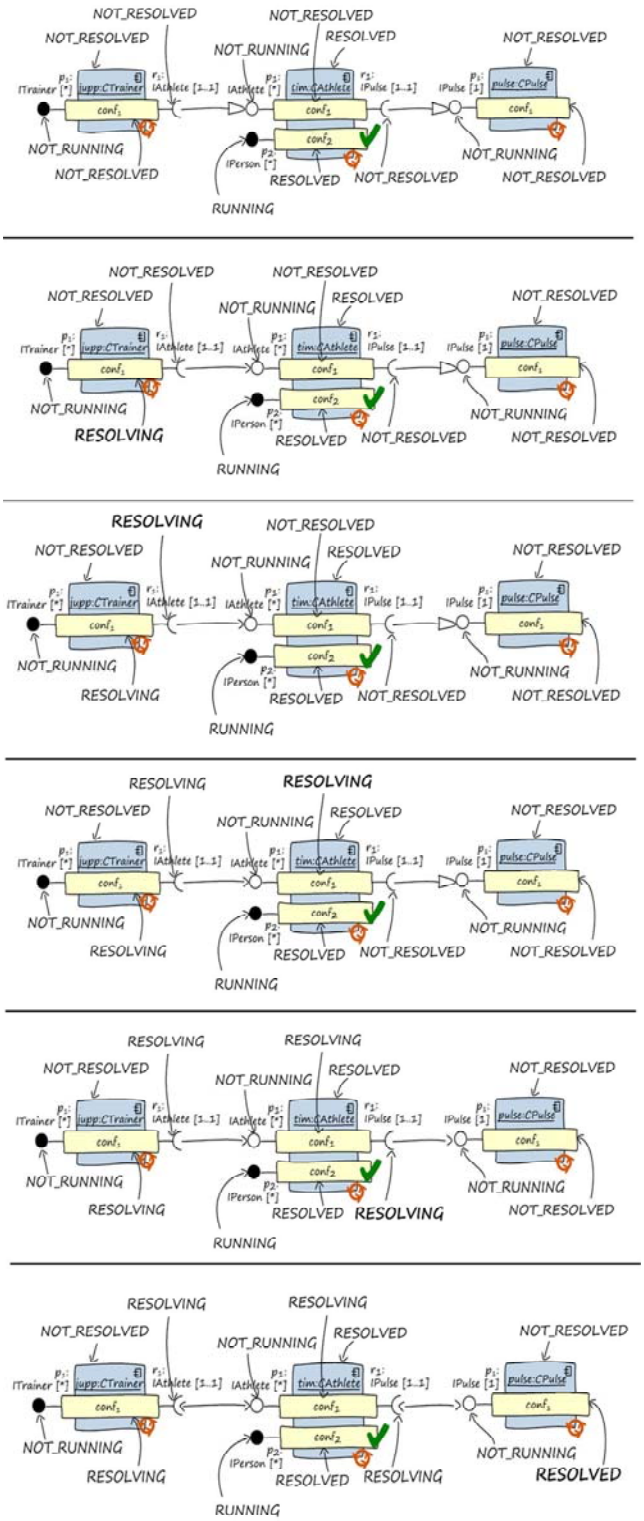


Figure 17. Initial situation in the Dynamic Adaptive System.

For the higher configuration, $conf_1$, *CAthlete* requires a reference to a service of type *IPulse*. The *CPulse* component is able to provide this service. As the provided service, *IAthlete*, of configuration conf1 of component *CAthlete* is not requested by any other component and has not set the *requestRun* flag, this higher configuration is not activated.

Figure 18 shows the following situation: A component, *CTrainer*, has been activated and integrated into our dynamic adaptive system. In the following the decentralized dynamic configuration mechanism is shown. Based on the interaction between the state machines of the adaptive components the dynamic adaptive system is reconfigured and the component is dynamically integrated into the system.

The configuration strategy is then as follows. Each service with *requestRun* flag set—in Figure 18 the new service *ITrainer* of the *CTrainer* component—resolves the required services transitively from the root to the leaf. Once all required services are resolved these services are activated (RUNNING) from the leaf to the root. If not all required services were resolvable, the resolved services are set back to NOT_RESOLVED. This allows other services to resolve these services.



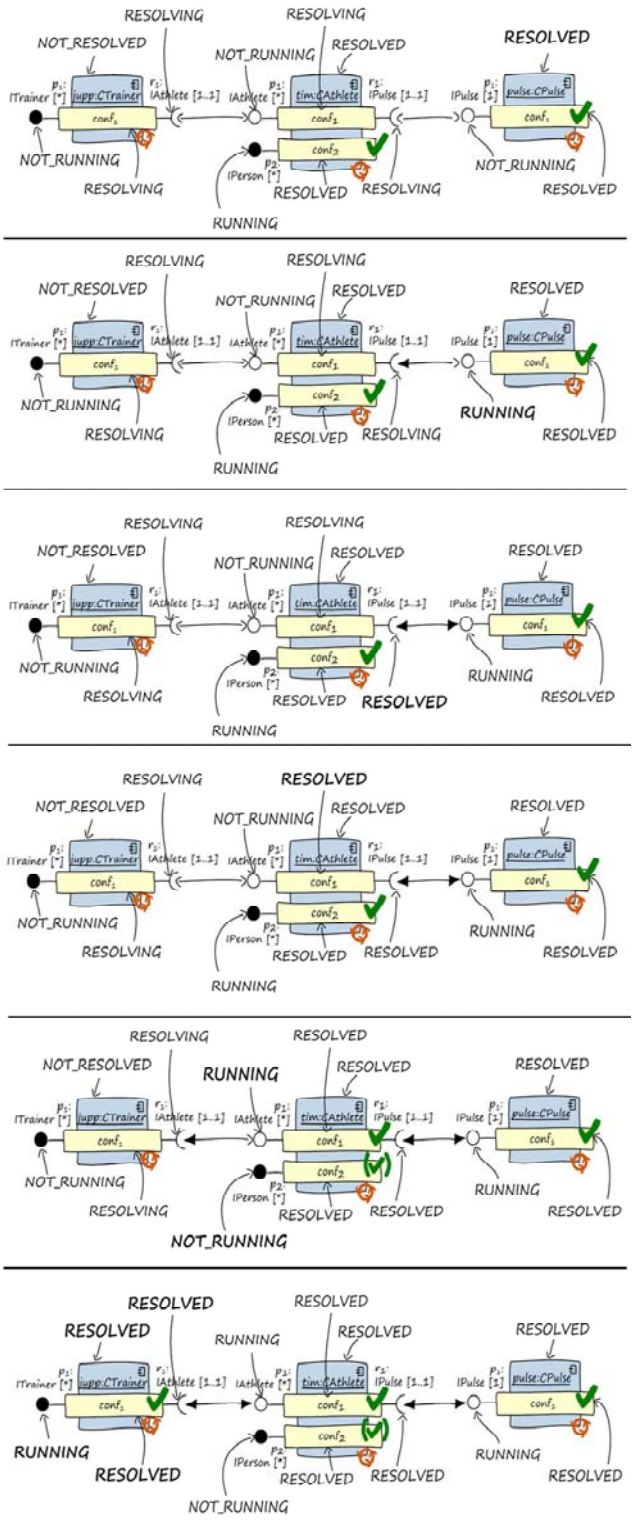


Figure 18. Step-by-Step decentralized dynamic configuration of the Smart Biathlon Training System.

V. CONCLUSION

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

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

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

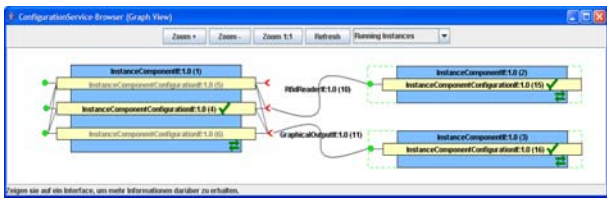


Figure 19. DAiSI Dynamic Adaptive System Monitor.

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

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

In this paper we have shown DAiSI’s new component model supporting service cardinalities and the new decentralized dynamic configuration mechanism. A first dynamic adaptive system has been successfully implemented in the assisted sports training domain.

Consequently, further systems will be realized based on the new DAiSI version. Additional research is required to establish concepts to provide a proper balance between controllability of the system’s applications and the autonomy of the system components participating in these applications.

 ISBN: 978-1-61208-341-4

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.

[2] J. Kramer and J. Magee. A rigorous architectural approach to adaptive software engineering. Journal of Computer Science and Technology, 24(2):183{188, 2009}.

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

[4] Emergency assistance system, Webpage of the cebit exhibit 2009, http://www2.in.tu-clausthal.de/~Rettungsassistenzsystem/, accessed 2014

[5] Intelligent beer dispensing system, Webpage of the cebit exhibit 2010", http://www2.in.tu-clausthal.de/~smartschank/systembeschreibung.php, Online; accessed 2014

[6] DIRMEIER SmartSchank, Intelligent Beer Dispensing System, DIRMEIER GmbH, http://www.dirmeier.de/DIRMEIER-0-0-0-1-1-1.htm, Online; accessed 2014

[7] A. Rausch, D. Niebuhr, M. Schindler, and D. Herrling. Emergency Management System. In Proceedings of the International Conference on Pervasive Services 2009 (ICSP 2009), 2009.

[8] Bilateral German-Hungarian Collaboration Project on Ambient Intelligent Systems. http://www.belami-project.hu/~micaz/belamiproject/history/part1. Online; accessed 2014.

[9] 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.

[10] H. Klus, D. Niebuhr, and A. Rausch. A Component Model for Dynamic Adaptive Systems. In Proceedings of the International Workshop on Engineering of software services for pervasive environments (ESSPE 2007), 2007.

[11] H. Klus, D. Niebuhr, and A. Rausch. Dependable and Usage-Aware Service Binding. In Proceedings of the third International Conference on Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE 2011), 2011.

[12] A. Rausch and D. Niebuhr. ECas News Journal, DemSy—A Scenario for an Integrated Demonstrator in a Smart City. 2010.

[13] C. Deiters, M. Köster, S. Lange, S. Lützel, B. Mokbel, C. Mumme, and D. Niebuhr, NTH computer science report, DemSy—A Scenario for an Integrated Demonstrator in a SmartCity. 2010.

[14] D. Niebuhr, H. Klus, M. Anastasopoulos, J. Koch, O. Weiß, and A. Rausch. DAiSI—Dynamic Adaptive System Infrastructure. Technical Report Fraunhofer IESE, 2007.

[15] M. Anastasopoulos, H. Klus, J. Koch, D. Niebuhr, and E. Werkman. DoAmI—A Middleware Platform facilitating (Re-)configuration in Ubiquitous Systems. In Proceedings of the Workshop on System Support for Ubiquitous Computing (UbiSys). 2006.

[16] C. Szyperski. Component Software. Addison Wesley Publishing Company. 2002.

[17] 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, S. 3–12.

[18] J. Magee, J. Kramer, and M. Sloman. Constructing Distributed Systems in Conic. In: IEEE Transactions on Software Engineering 15 (1989), Nr. 6, S. 663–675

[19] J. Kramer. Configuration Programming: A Framework for the Development of Distributable Systems. In: Proceedings of IEEE International Conference on Computer Systems and Software Engineering (COMPEURO 90). 8-10 May 1990, Tel-Aviv, Israel: IEEE Computer Society Press, 1990. ISBN 0818620412, S. 374–384

[20] R. R. Aschoff, and A. Zisman. Proactive adaptation of service composition. In: H. A. Müller, L. Baresi (Hrsg.): 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, S. 1–10

[21] A. Rasche, 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, S. 164–171

[22] H. Klus. Anwendungsarchitektur-konforme Konfiguration selbstorganisierender Softwaresysteme, Ph.D. Thesis, Technische Universität Clausthal, 2013.

[23] D. Niebuhr. Dependable Dynamic Adaptive Systems: Approach, Model, and Infrastructure. Clausthal-Zellerfeld, Technische Universität Clausthal, Institut für Informatik. Dissertation. 2010

 ISBN: 978-1-61208-341-4