Guaranteeing Correctness of Component Bindings in Dynamic Adaptive Systems based on Runtime Testing

Dirk Niebuhr, Andreas Rausch
TU Clausthal, Institute of Computer Science, Software Systems Engineering Research Group
Clausthal-Zellerfeld, Germany
{dirk.niebuhr, andreas.rausch}@tu-clausthal.de

ABSTRACT

Component-based software engineering has been successfully applied over the past years. Future system generations, like pervasive systems, are a vast array of decentralized, distributed, autonomic, heterogeneous, organically grown and continually evolving subsystems respectively components. Components may join or leave these systems during the whole system life-cycle – even during runtime. We depend more and more on these dynamic adaptive systems. Hence we have to guarantee their correctness although the systems are evolving during runtime. In this paper we will show that existing approaches cannot guarantee the correctness of component bindings in dynamic adaptive systems. To guarantee system correctness and to support binding of components during runtime we integrate runtime testing into our component infrastructure DAiSI.

Categories and Subject Descriptors

D.2.4 [**Software**]: SOFTWARE ENGINEERING—*Software/Program Verification*; D.2.11 [**Software**]: SOFTWARE ENGINEERING—*Software Architectures*; F.3.1 [**Theory of Computation**]: LOGICS AND MEANINGS OF PROGRAMS—*Specifying and Verifying and Reasoning about Programs*

General Terms

VERIFICATION, RELIABILITY, DESIGN

Keywords

Runtime Testing, Dependable Component Binding, Dependability, Component Based Software Engineering

1. INTRODUCTION

Component-based software engineering (CBSE) [20, 5] has been continuously improved and successfully applied over the past years changing the predominant development paradigm: Systems are no longer developed from scratch, but composed of existing, reusable software 'parts' - called software components. Wherever required additional glue code is developed to bind components that do not fit together exactly. Thereby CBSE promises to enable practical reuse of software components. A higher reuse rate leads to lower implementation costs, faster time-to-market, higher software quality, and last but not least to more flexible and adaptable software systems.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIPE'09, July 13–17, 2009, London, United Kingdom.
Copyright 2009 ACM 978-1-60558-648-9/09/07 ...$5.00.

1.1 Component-Based Pervasive Systems

Current research areas, like for instance ubiquitous computing, pervasive computing, or ultra-large scale systems [7] share a common future trend: Complex software systems are no longer considered to have well-defined boundaries. Instead future software systems consist of a vast array of distributed, decentralized, autonomous, interacting, cooperating, organically grown, heterogeneous, and continually evolving subsystems. Adaptation, self-x-properties, and autonomous computing are envisaged in order to respond to short-term changes of the system itself, the context, or a user's expectations. Furthermore, to cover the long-term evolution of systems becoming larger, more heterogeneous, and long-lived, pervasive systems must have the ability to continually evolve and thereby show up emerging behavior.

However, pervasive systems are only one example for systems that cannot be designed and built as a whole by a single vendor. From a component-based perspective pervasive systems consist of software components from various vendors with different component life cycles. Consequently, as pervasive systems are continually evolving, components may join or leave these systems during their whole life cycle - even during runtime. Therefore a common component infrastructure is required that supports runtime binding, composition and adaptation of component-based systems.

1.2 Towards Dependable Dynamic Adaptive Systems

Those component infrastructures for dynamic adaptive systems are already available - developed and provided by researchers and practitioners. Some component infrastructures, widely used in practice, already provide rudimentary support for dynamic adaptation, like OSGi [1], Java CAPS [19], or CORBA [18]. Others that provide more sophisticated support for dynamic adaptive systems can be found in research or pre-product component infrastructures, like MEF [15], ASG [8], or DAiSI [16].

While software systems, like pervasive systems, increasingly pervade our daily lives, dependability is no longer restricted to safety-critical applications, but rather becomes a cornerstone of the information society. We rely more and more on the correctness and availability of these pervasive systems. By looking at the past, we can find several software failures, like the automated baggage system at Denver airport [21]. Although these systems were built as a whole, severe failures occured. Some of them were explicitly caused by static software reuse like the Ariane 5 [6].

In pervasive systems we have to deal with an additional challenge: Those systems have no clear boundary as the concrete configuration variants in use are not predictable. Hence all possible variants can never be tested in advance during development time, since we do not necessarily know the components, which are bound together at runtime. This leads to an increased risk of software failures at runtime.

To sum up: On the one hand there is an increasing demand for component-based systems supporting dynamic adaptation, caused by joining or leaving components during runtime. On the other hand the issues of dependability become more critical. Hence, the existing component infrastructures for dynamic adaptive component-based systems have to be extended to guarantee system dependability although a system's components are joining and leaving it during runtime.

1.3 Successful Approaches from other Areas

Similar issues are already well known and considered in other systems, which are on a lower level and closer to hardware, like for instance in network infrastructure, home entertainment systems, or operating systems. These systems are built of components from different vendors. They are continually evolving during their whole life cycle. Components may join or leave a system during runtime. And most important: the components share a common infrastructure that provides techniques to guarantee system correctness with respect to dynamic adaptation of the overall system.

All of these approaches consist of two main concepts:

1. *Standardized and broadly accepted interfaces in the considered domain:* In the network domain physical connectors, like for instance RJ 45, as well as their pin configuration are standardized and well known by component vendors. A similar situation appears in operating systems: The interface for printer drivers is standardized. Third party vendors adhere to this interface specification to realize printer drivers that are plugged into an operating system during runtime.

Components implementing an interface specification may have errors no matter how precise and complete interfaces are specified. Moreover, as component vendors want to develop components with unique features, interface specifications must provide semantic underspecifications. In addition sample implementations are provided to give vendors a common understanding of the shared behavior of all components implementing the underspecified interface. However, a certain degree of freedom how to implement these standard interfaces remains. Hence, to guarantee the compatibility of resulting component implementation variations an additional concept is typically established:

2. *Runtime compatibility tests are performed before binding and using:* Consider again the network domain: Once you physically connect a PC to a network via RJ 45 the connecting switch initiates a first rudimentary compatibility test protocol before the PC is routed into the internet. An analogue situation appears in operating systems: Assume you are installing a printer. The installation routine asks you to print a test page to validate whether the driver works properly or not.

The main focus of the paper is to apply this approach – runtime compatibility tests before binding and using components – within a component-based infrastructure.

1.4 Contribution and Structure of the Paper

Available component infrastructures provide basic support for dynamic adaptation of component-based systems during runtime but do not handle the crucial dependability issues. However, to move component technologies from assisting standard system architecture development to supporting the development of pervasive systems we have to extend existing component infrastructures with basic technologies to guarantee the overall system dependability although the system is continually evolving during runtime.

In this paper we introduce a runtime testing extension to a component infrastructure to guarantee correctness of a system although components join and leave it during runtime.

The rest of the paper is structured as follows: In the next section we will introduce an application scenario from the emergency assistance domain. Based on this scenario we show in the next section that – although all components are verfied as correct with respect to a given domain interface specification – system correctness is not guaranteed if components can be bound to other components during runtime. In Section 4 we introduce our runtime testing approach. A short conclusion rounds up the paper.

2. APPLICATION EXAMPLE

Imagine a huge disaster like the one, which occurred during an airshow in Ramstein in 1988. Two planes collided in air and crashed down into the audience. In cases of such a disaster with many seriously injured casualties, medics need a quick overview of the whole situation. They do a triage [13], classifying casualties regarding the severity of their injury, in order to treat casualties with serious injuries first.

In our scenario, medics are supported by an IT system, providing them with an overview and enabling them to keep track of the physical situation of previously classified casualties. Each medic is equipped with a bunch of casualty units. Casualty units are microcontrollers storing data about specific casualties like their name, gender, or current position. Medics equip each casualty they discover with such a unit.

For seriously injured casualties biosensors like a pulse rate sensor or a blood pressure sensor can be attached to a casualty unit in order to automatically calculate their triage class. The biosensors enable medics and an incident command to monitor a casualty's physical condition without needing to be physically present at his place. So an incident command can quickly send a nearby medic to a casualty, whose situation became critical.

The described system is a typical ultra-large scale component-based system. It consists of a vast array of those casualty units, biosensors, medic units, and an incident command subsystem, which are bound during runtime. The overall system is evolving during runtime: new casualties

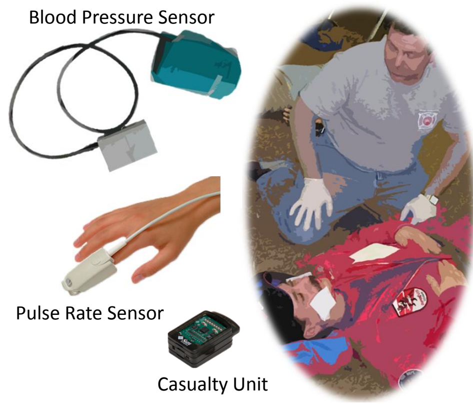

Figure 1: Overview about the different components within the application example

may be integrated via their casualty unit or casualty units may leave the system as casualties are evacuated. The components involved in the scenario are depicted in Figure 1.

3. COMPONENT-BASED IMPLEMENTATION

In this section we will show the domain architecture and interface specifications for a part of our application example. Components developed by two different vendors refer to this specification. We will describe, why binding their components at runtime causes problems.

3.1 Domain Architecture

Dynamic adaptive component-based systems like our application example, are based on standardized domain architectures and interface specifications. In our application example this domain architecture needs to contain **interface** BloodPressureSensorIf and **interface** PulseRateSensorIf for the biosensor components. The casualty unit uses these interfaces in order to provide its **interface** CasualtyUnitIf.

Since we want to build dependable systems from components developed by different vendors, the domain model does not only contain syntactical information like method signatures or datatypes. It also contains semantic specifications following the Design by Contract [14] approach. These specifications may be available in JML [12] or Spec# [3].

In our example **interface** PulseRateSensorIf may specify a method getPulseRate(), which must not return a negative value. The same postcondition may be specified for two methods contained in **interface** BloodPressureSensorIf: getSystolicBloodPressure() and getDiastolicBloodPressure(). Moreover an invariant may state the medical knowledge, that the systolic blood pressure must be greater or equal than the diastolic blood pressure.

More complex is the specification of **interface** CasualtyUnitIf. It contains methods dealing with personal data of the casualty like getGender(). Moreover it contains the specification of the getTriageClass() method. It returns the triage class of the casualty based on pulse rate and blood pressure.

Remarkable about this specification is that *TriageClass.Dead* should only be returned if both – pulse rate and blood pressure – are equal to zero. If only one is equal to zero, this may mean, that a sensor might have slipped off. Consequently *TriageClass.Unknown* should be returned. If biosensors are not available, the method should return the vital condition as it has been manually set by a medic.

Based on the domain architecture, different vendors can implement components for our application example.

3.1.1 Implementation by VendorA:

VendorA follows a straightforward approach. He has developed and shipped three components: a casualty unit, a blood pressure sensor, and a pulse rate sensor. VendorA's implementation of the casualty unit is directly derived from the specification. He simply implemented the getTriageClass() method exactly the way, as it has been specified in the domain architecture, considering the two relevant cases: If both sensor values are equal to zero *TriageClass.Dead* is returned whereas *TriageClass.Unknown* is returned if only a single sensor value is zero.

3.1.2 Implementation by VendorB

VendorB decides to develop only two components: a casualty unit as well as a combined biosensor integrating pulse rate sensor and blood pressure sensor. The combined sensor is built as a wrist cuff and therefore cannot slip off during usage. Consequently VendorB may add an invariant to describe this additional property of the combined sensor: (getPulseRate()==0) <==> (getSystolicBloodPressure()==0) <==> (getDiastolicBloodPressure()==0), so that these three values can only be zero at the same time.

Considering this sensor, VendorB also implements a casualty unit. It corresponds to the domain architecture with one small difference: Since the combined sensor guarantees that all sensor values can only be zero at the same time, VendorB implements the getTriageClass() method simpler: the triage class can always be considered as *TriageClass.Dead*, when the pulse rate is zero without considering the blood pressure in addition. This is not a violation of the domain specification if considered together with VendorB's biosensors as these sensors guarantee that it will never happen that only one sensor value is equal to zero. However it is an incorrect implementation of the interface specification from the domain architecture if considered on its own.

3.2 Verification in our Example

Both vendors are able to test and verify their solutions. Both will be successful. Even if vendors use static or dynamic checkers for JML, like ESC/Java2, LOOP, JACK, or JMLRAC [4], they will be able to prove that their component bindings do not violate the interface specifications. To sum up, each vendor solution is verified to be correct.

Consider a situation, where components from VendorA and VendorB are in use. As long as VendorB's casualty units are only connected to VendorB's combined sensors, everything will work fine. However we can imagine a situation, where such a combined sensor is out of battery. Now a medic, equipped with VendorA's components, comes along and connects a pulse rate sensor and a blood pressure sensor from VendorA to VendorB's casualty unit.

Figure 2 shows one possible resulting scenario. Assume the finger clip of the pulse sensor slips off. The pulse rate sensor of VendorA will return zero. As VendorB's casualty unit only considers the pulse rate for triage class calculation it's getTriageClass() method will return *TriageClass.Dead* erroneously. If the vital condition of the casualty in the

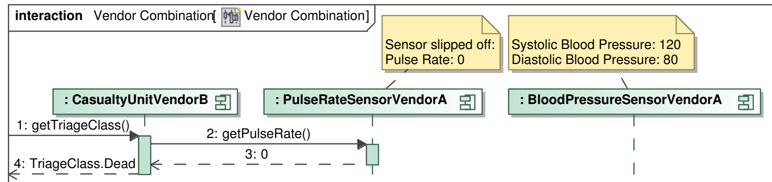

Figure 2: Binding VendorA's Sensors with VendorB's Casualty Unit

following becomes worse, the value of the blood pressure sensor will decrease. Unfortunately nobody will notice this – no medic will be sent to him as he is supposed to be dead.

Now assume that due to dependability reasons dynamic checkers like for instance the runtime assertion checker JMLRAC [4] are used during runtime. JMLRAC can be used to execute Java bytecode, which has been compiled by the JML runtime assertion checker compiler. This bytecode contains the specified pre- and postconditions as well as specified invariants. JMLRAC checks during execution, whether these conditions are violated and generates exceptions containing, which condition has been violated.

In this case we will get a runtime exception which results in a failure of the overall system. The exception states, that a postcondition of getTriageClass() has been violated.

As you can see, specifications in JML guiding a runtime assertion checker help us to detect incompatibilities within our example. However we are not satisfied with the results due to major drawbacks: We cannot detect incompatibilities in advance. We need to call a provided method in order to realize, that it does not satisfy the specified postconditions. Therefore we can only detect wrong behaviour at the time when it occurs. If you think of an in-car scenario, this corresponds to a situation, where we realize, that the inflateAirbag() method does not work as expected exactly in the moment, where we try to inflate the airbag due to a major accident. This does not really help us. Instead we need a mechanism to detect incompatibilities in advance.

4. OUR APPROACH TO DEPENDABLE DYNAMIC ADAPTIVE SYSTEMS

To adress these drawbacks, we provide an approach which enables us to automatically detect incompatibilities between components using runtime testing. However we can only take advantage of our approach, if we have a component infrastructure, which uses the test results to update the component binding. In the past, we developed the Dynamic Adaptive System Infrastructure DAiSI [16] aiming at dynamic adaptive systems. In the following we will introduce the underlying DAiSI component model before explaining our runtime testing approach in more detail.

4.1 DAiSI Component Model

The basic idea of the DAiSI component model is, that component developers can specify, which services their component *provides to other components* and which services it *requires from other components*. Therefore we have three notions in our component model: the basic notion of a *Component*, the notion of a *Component Service*, and the notion of a *Component Service Reference*. Component Services describe offered services of a component while Component Service References describe required services.

In order to support adaptation, the component model supports so called *Component Configurations*. Within a Component Configuration, a component can specify by Component Services References, which services it requires and on the other hand, which Component Services it can offer.

This enables DAiSI to automatically establish and adapt a component binding at runtime based on syntactical interface matching as described in [10]. However DAiSI does not guarantee that the established component binding is correct – we will describe how we can change this in this paper.

4.2 Binding Dynamic Adaptive Systems Based on Runtime Testing

We want to provide an approach to detect incompatibilities at runtime in advance. The basic idea in our approach is that service users (i.e. components which require a service) specify runtime-compliance test cases which are executed at runtime to evaluate the compatibility of a service provider.

Tests need to be executed by the component infrastructure before a Component Service is bound to a Component Service Reference (i.e. at binding-time). These tests decide, whether the behavior of a provided service corresponds to the behavior expected by a service user. If the test passes, the component infrastructure binds the service provider to the service user which defined the test case.

In our example, VendorB might specify a testcase for the (unknown) pulse rate sensor required by his casualty unit simply by querying the sensor and checking, whether the result is in an expected range.

After binding, the compatibility of service provider and service user needs to be monitored, since it can change over time as the internal state of a service provider or service user may change. If you look at our application example, it does not help us, if the tests at binding time pass: an incompatibility may suddenly come up, when the pulse sensor slips off. Thus we need to provide a mechanism enabling us to execute test cases triggered by state changes.

Our approach is, defining *equivalence classes* regarding the state of the service binding. By equivalence classes we understand state spaces of service users and service providers, where the same behaviour should apply. If we have these classes, runtime-compliance tests need to be executed, whenever the state changes in a way, that the equivalence class changes.

As our vision is, that service provider and service user do not know each other at development time, we cannot define shared Equivalence Classes for a binding. Therefore we need a methodology, how we can define these Equivalence Classes.

A service provider can define equivalence classes based on its implementation's control flow. A service user can define equivalence classes based on its expectations regarding the service provider. Both options alone do not help us: the equivalence classes defined for a service binding by a service

user can differ from the ones defined by a service provider. One reason for incompatibilities is that the understanding of a service binding and therefore the resulting equivalence class definitions differ. Therefore state spaces, where definitions of equivalence classes differ are especially important. They would be missed, if we defined equivalence classes only at one endpoint of the binding relation.

Instead our approach is, that equivalence classes for a service binding are defined by both: service user and service provider. This however leads us to the question, how they can be combined in order to derive a changed equivalence class for the service binding. Service provider and service user represent it independently simply as a number.

An equivalence class of a service binding is the tuple containing the service user's view and the service provider's view of the Equivalence Class. We call this tuple a *combined equivalence class*. Whenever the combined equivalence class changes towards an untested combination, a runtime-compliance test needs to be executed. As you can imagine, two components can calculate a combined equivalence class, before they are bound, which means that we can use the runtime-compliance test to decide about the compatibility at initial binding-time as well. The sequence for a runtime-compliance test is depicted in Figure 3.

For our example this means, that VendorA and VendorB define equivalence classes for all provided services and required services of their components. In the following we will only look at the Equivalence Class definitions of the incompatible components, namely VendorA's pulse rate sensor and blood pressure sensor as well as VendorB's casualty unit.

VendorA decides, that his sensors provide the same behavior regardless of the internal state. Therefore VendorA implements the getEquivalenceClass() method for both sensors in a trivial way, returning zero as Equivalence Class.

VendorB provides a sophisticated equivalence class definition for the pulse rate sensor required by his casualty unit. He can think of three different equivalence classes: the pulse rate may be *out of range* (below zero), *in range* (above zero), or *zero*, which may be due to a slipped off sensor, due to a malfunction, or due to a cardiac arrest of the casualty.

The equivalence class definition is depicted in Listing 1[1]. The equivalence class definition for the blood pressure sensor required by VendorB's casualty unit is left out intentionally, since it is not required to demonstrate our approach.

```
1  public int getEquivalenceClass_pulseRateSensor(){
2      int pulseRate = pulseRateSensor.getPulseRate();
3      if ((pulseRate<0)){
4          return -1;
5      } else {
6          if (pulseRate == 0){
7              return 0;
8          } else {
9              return 1;
10         }
11     }
12 }
```

Listing 1: Implementation of getEquivalenceClass() for the Required Pulse Rate Sensor.

Next to equivalence classes, VendorB needs to define test cases, which are executed by the component infrastructure, whenever the state of one of the involved components changes in a way, leading to an untested combined equivalence class. The execution of these test cases should determine, whether the two components are still compatible regarding the service binding. Therefore VendorB defines a method equivalenceClassTest_pulseRateSensor()[1].

In our case, VendorB states, that he is not compatible to the pulse rate sensor, if the pulse is out of range, therefore he returns false in this case. In all other cases, he queries pulse rate sensor and blood pressure sensor and checks, if exactly one of both is equal to zero. If this is the case, he returns that the test failed, otherwise the test passes[2].

```
1 public boolean equivalenceClassTest_pulseRateSensor(int
      providerEquivalenceClass, int
      userEquivalenceClass){
2     if (userEquivalenceClass == -1){
3         return false;
4     }
5     int bloodPressureSys = bloodPressureSensor.
          getSystolicBloodPressure();
6     int pulseRate = pulseRateSensor.getPulseRate();
7     boolean zeroPulseCorrespondsBloodPressure = !(
          pulseRate==0&&(bloodPressureSys>0));
8     return (zeroPulseCorrespondsBloodPressure);
9 }
```

Listing 2: Runtime-Compliance Testcase for the Required Pulse Rate Sensor.

In case of a slipped off sensor the test fails, since the pulse rate is zero while the blood pressure is still above zero. The component infrastructure removes the service binding and puts the casualty unit in the second best configuration, where the triage class needs to be manually set by a medic.

As you can see, we are now able to detect incompatibilities in advance. However there is still a major drawback of the runtime-testing approach: Since we are testing components of a system at runtime, we need to ensure that the test case execution has no side-effects on a running system. Our approach therefore integrates a so-called testing mode.

Before runtime test are executed, all involved components[3] are put into testing mode. Therefore these components know, that they cannot rely on the interaction with other components until this test mode is deactivated after test execution. This enables them to restore their state after test execution and to simulate some effects (consider the airbag example sketched before: to prevent that the airbag is inflated due to a test execution you need to substitute code inflating the airbag by simulation code in testing mode).

5. ACKNOWLEDGEMENTS

The approach presented here has been elaborated in cooperation with Cornel Klein, Jürgen Reichmann, and Reiner Schmid from Siemens AG. Together we filed a patent for it.

6. CONCLUSIONS AND FURTHER WORK

Reconfiguration, which means changing the service binding between components, is necessary for dynamic adaptive systems, since components may enter or leave a system at runtime. However proving the correctness of a service binding at runtime is not possible in general. Specifying interfaces in a domain architecture does not help either: There are several reasons, why vendors vary from an interface specification. Our approach is based on runtime-testing of components. This enables us to detect incompatibilities of pro-

[1] The postfix _pulseRateSensor in the method name specifies the associated required service.

[2] Note that the test case here is very simple in order to focus on the general principles of our approach. You can specify much more complicated test cases using our approach.

[3] A component is involved, if it is directly or transitively connected by service bindings to one of the components, which drives the test or which is currently under test

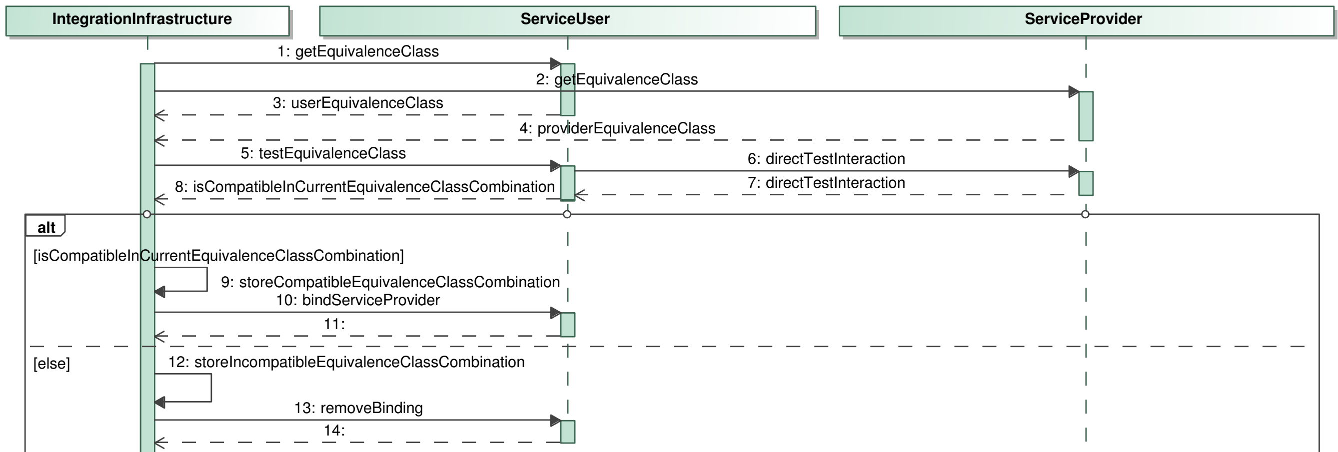

Figure 3: Sequence of the Runtime-Compliance Test in Our Approach.

vided and required services before they occur. Thus a component infrastructure can avoid semantically incompatible service bindings and chose valid bindings instead.

We developed dynamic adaptive systems from various domains based on DAiSI [2, 9, 11, 16] in the past. A proof-of-concept implementation of the runtime-testing approach has been integrated into our latest prototype of DAiSI. Based on the newly integrated DAiSI featuring runtime testing, we realized a dependable emergency assistance system, which was exhibited at CeBIT 2009 [17].

We did not take care yet about cyclic dependencies of components. Moreover we need to investigate test case generation, to enable component developers to provide a single specification of their components and assure good test cases while trading-off test case execution overhead. One next step could be, using runtime assertions generated from formal specification as test oracle for the runtime-compliance tests as it is the idea of JMLUnit. Another open question is, whether it is reasonable to test each equivalence class combination only once or whether to test it each time, when the equivalence class combination changes without considering previous test case results.

7. REFERENCES

[1] O. Alliance. *OSGi Service Platform Core Specification*, 2007.

[2] 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)*, Orange County, California, USA, Sept. 2006. Electronic Proceedings.

[3] M. Barnett, L. K. R. M., and W. Schulte. *The Spec# Programming System: An Overview*, volume 3362/2005 of *Lecture Notes in Computer Science*, pages 49–69. Springer, Berlin / Heidelberg, 2005.

[4] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Leino, and E. Poll. *An overview of JML tools and applications.* 2003.

[5] I. Crnkovic, M. Chaudron, and S. Larsson. Component-based development process and component lifecycle. *Software Engineering Advances, International Conference on*, 0:44, 2006.

[6] M. Dowson. The ariane 5 software failure. *SIGSOFT Softw. Eng. Notes*, 22(2):84, 1997.

[7] P. Feiler, R. P. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, L. Northrop, D. Schmidt, K. Sullivan, and K. Wallnau. Ultra-large-scale systems: The software challenge of the future. Technical report, The Carnegie Mellon Software Engineering Institute, 2006.

[8] K. Herrmann, K. Geihs, and G. Mühl. Ad hoc service grid - A self-organizing infrastructure for mobile commerce. In *Proceedings of the IFIP TC8 Working Conference on Mobile Information Systems (MOBIS 2004)*, pages 261–274, Oslo, Norway, 2004. Kluwer.

[9] T. Jaitner, M. Trapp, D. Niebuhr, and J. Koch. Indoor-simulation of team training in cycling. In E. Moritz and S. Haake, editors, *ISEA 2006*, pages 103–108, Munich, Germany, July 2006. Springer.

[10] H. Klus, D. Niebuhr, and A. Rausch. Towards a component model supporting proactive configuration of service-oriented systems. In *ICEBE '07: Proceedings of the IEEE International Conference on e-Business Engineering*, pages 600–603, Hong Kong, China, Oct. 2007. IEEE Computer Society.

[11] H. Klus, D. Niebuhr, and O. Weiß. Integrating sensor nodes into a middleware for ambient intelligence. In S. Schäfer, T. Elrad, and J. Weber-Jahnke, editors, *Proceedings of the Workshop on Building Software for Sensor Networks*, Portland, Oregon, USA, Oct. 2006. ACM. Electronic Proceedings.

[12] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of jml: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

[13] Manchester Triage Group. *Emergency Triage.* BMJ Books, 2005.

[14] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[15] Microsoft. Managed extensibility framework. http://www.codeplex.com/MEF.

[16] D. Niebuhr, H. Klus, M. Anastasopoulos, J. Koch, O. Weiß, and A. Rausch. DAiSI - Dynamic Adaptive System Infrastructure. Technical report, Fraunhofer Institut Experimentelles Software Engineering, IESE-Report No. 051.07/E, Jun 2007.

[17] D. Niebuhr, M. Schindler, and D. Herrling. Emergency assistance system – webpage of the cebit exhibit 2009. http://www2.in.tu-clausthal.de/~Rettungsassistenzsystem.

[18] Object Management Group. Common object request broker architecture. http://www.corba.org/, 2004.

[19] Sun Microsystems. *Sun Java Composite Application Platform Suite*, 2008.

[20] C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison Wesley Publishing Company, 2002.

[21] Thomas Huckle. Collection of software bugs. http://www5.in.tum.de/~huckle/bugse.html.