Guaranteeing Correctness of Component Bindings in Dynamic Adaptive Systems

Dirk Niebuhr and Andreas Rausch
Software Systems Engineering, Institute for Computer Science, Clausthal University of Technology
Clausthal-Zellerfeld, Germany
{dirk.niebuhr, andreas.rausch}@tu-clausthal.de

Abstract—Component-based software engineering has been continuously improved and successfully applied over the past years. Future systems, like ultra-large scale systems, are a vast array of decentralized, distributed, autonomic, heterogeneous, organically grown and continually evolving subsystems resp. components. Components join or leave these systems during the whole system life-cycle – even during runtime. Since we depend on these dynamic adaptive systems we have to guarantee their correctness although they evolve during runtime. To guarantee system correctness and to support binding of components during runtime we claim to integrate specific concepts into a component infrastructure. In this paper we will introduce a technique for runtime testing.

Index Terms—Runtime Testing, Dependable Component Binding, Dependability, Component Based Software Engineering, Dynamic Adaptive Systems, Correctness

I. INTRODUCTION

Component-based software engineering (CBSE) [1], [2] 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 components. 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 more flexible and adaptable software systems.

Current research areas, like for instance ubiquitous computing, pervasive computing, or ultra-large scale systems [3], share a common future trend: Complex software systems are no longer considered to have well-defined boundaries. Instead future software systems - so called dynamic adaptive 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 the user’s expectations. Furthermore, to cover the long-term evolution of systems becoming larger, more heterogeneous, and long-lived, dynamic adaptive systems must have the ability to continually evolve and thereby show up an emerging behavior.

Dynamic adaptive systems cannot be designed and built as a whole by a single vendor. Dynamic adaptive systems as well as their subsystems consist of software components from various vendors with different component life cycles. Consequently components may join or leave these systems during their whole life cycle – even during runtime.

While dynamic adaptive systems increasingly pervade our daily lives, dependability is no longer restricted to safety-critical applications. By looking at the past, we can find severe software failures although these systems were built as a whole, like the automated baggage system at Denver airport [4] or Ariane 5 [5]. In dynamic adaptive systems we have to deal with an additional challenge: Those systems have no clear boundary resp. extension as the configuration variants in use are not predictable. Hence all possible variants can never be tested in advance during development time. This increases the risk of software failures.

Therefore a common component infrastructure is required that supports runtime binding, composition and adaptation of dependable dynamic adaptive systems.

A. Related Work

Component infrastructures, widely used in practice, already provide rudimentary support for dynamic adaptation, like for instance OSGi [6], Java CAPS [7], or CORBA [8]. More sophisticated support for dynamic adaptive systems can be found in research and pre-product component infrastructures, like MEF [9], ASG [10], or DAI$SI$ [11].

Within the IST PLASTIC project [12] the PLASTIC platform has been developed which is intended to provide dependability by validation of functional properties at runtime by monitoring component interaction [13]. In contrast to our approach the PLASTIC platform is only capable of detecting incompatibilities after they occurred.

Our assumption is, that the correctness of all possible system configurations of dynamic adaptive systems cannot be verified at development time. However, component correctness with respect to the interfaces they implement can be verified in advance within a given environment. Hence, we still have to verify the correctness of component bindings during runtime if a component is used within a new environment. This is a first important step towards guaranteeing the correctness of a system during runtime.

There are several approaches which provide (limited) statements regarding the correctness of a component binding like Enriching the domain architecture and its interface specifications towards a complete specification, Proving the refinement relation between provided and expected behaviour [14], [15],
[16], [17], or Bisimulation. They are shorty discussed regarding their applicability to achieve dependable dynamic adaptive systems in [18].

B. Contribution and Structure of the Paper

In this paper we will introduce a runtime testing-based approach for dependable dynamic adaptive component-based systems. The rest of the paper is structured as follows: In Section II we will introduce an application scenario from the emergency management domain. It will be used to introduce our approach to guaranteeing system correctness in Section III. A short conclusion rounds up the paper.

II. 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 a huge amount of seriously injured casualties, medics need to get a quick overview of the whole situation. The medics do a quick triage [19], classifying the 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. Each medic is equipped with a bunch of casualty units. A casualty unit stores data regarding a specific casualty like the casualty’s name, gender, or current position. Medics equip each casualty they discover with such a unit.

For seriously injured casualties a biosensor, consisting of a pulse rate sensor and a blood pressure sensor, can be attached to a casualty unit in order to keep the triage class of a casualty up to date if his condition changes over time.

This system is a typical dynamic adaptive component-based system. It consists of a vast array of components, which are bound during runtime. The overall system is evolving during runtime as new casualties may be integrated via casualty units as well as casualties may leave the system.

A domain architecture for Emergency Management Systems contains a specification of the interface of a casualty unit. It may contain a getTriageClass() method, which we use to illustrate our approach. This method returns the triage class of a casualty based on his pulse rate and blood pressure. The domain architecture specifies, that TriageClass.Dead should only be returned, whenever pulse rate and blood pressure are equal to zero. If only one of these values is zero a sensor might have slipped off. Consequently TriageClass.Unknown should be returned.

VendorA has developed and shipped two components: a casualty unit and a biosensor. VendorA’s implementation of the casualty unit is directly derived from the specification. He implemented the getTriageClass() method, 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. VendorA’s biosensor has two physical sensors – a wrist cuff measuring the blood pressure and a fingerclip measuring the pulse rate.

VendorB has developed and shipped two components as well: a casualty unit as well as a biosensor measuring pulse rate and blood pressure sensor by a single wrist-cuff.

Having this sensor in mind, VendorB also implements a biosensor, we face problems when a fingerclip slips off, as a casualty is classified as dead and therefore no medic will be sent to him anymore. Although VendorB specifies an additional requirement for biosensors, that pulse rate and blood pressure are zero only at the same time, we can detect misbehaviour only at the time it occurs as shown in [18].

III. OUR APPROACH TO DEPENDABLE DYNAMIC ADAPTIVE SYSTEMS

To adress these drawbacks, we provide an approach which enables us to automatically establish a component binding in a system using runtime tests to detect incompatible bindings. However we can only take advantage of such an approach, if we have a component infrastructure, which uses the results from our approach during component binding. In the past, we developed the Dynamic Adaptive System Infrastructure DAiSI [11] aiming at dynamic adaptive systems. The basic idea of the underlying DAiSI component model is, that component developers should be able to specify, which services their component provides to other components and which services it requires from other components.

As motivated before, we want to detect incompatibilities at runtime in advance. This is done by runtime testing in our approach: service users specify test cases which are executed on a service provider at runtime.

We need to execute tests before a service user is bound to a service provider (i.e. at binding-time). These tests decide, whether the behavior of a service provider corresponds to the expected behavior defined by a service user in terms of a test case. In our example VendorB specifies a test case for the biosensor required by his casualty unit. The test case queries the sensor and check, whether the result are in a range expected by VendorB. If it passes, the service provider can be bound to the specific service user.

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

\footnote{This is no violation of the domain specification, as his biosensor implementation guarantees that a single biosensor value will never be zero.}
Our approach here is, defining Equivalence Classes regarding the state of a Binding. By Equivalence Classes we understand state spaces of service user and service provider, where the same behaviour should apply. Runtime-compliance, tests need to be executed each time, when a state change implies a changed Equivalence Class.

As our vision is, that service provider and service user do not know each other at development time, we do not want to define shared Equivalence Classes. Instead we want to have a methodology, of combining them from Equivalence Classes defined separately by service user and service provider.

Service providers can define Equivalence Classes based on the control flow from their implementation. Service users can define Equivalence Classes based on their expectations regarding a required service. Both options alone do not help: Equivalence Classes defined by a service user can be very different from those defined by a service provider.

One reason for incompatibilities that we want to detect is, that the understanding of the binding and therefore the resulting Equivalence Classes differ. Therefore areas, where the Equivalence Classes differ are especially important. They could be missed, if we only define Equivalence Classes at one endpoint of the binding relation.

Instead our approach is, that Equivalence Classes for a binding are defined by both: service user and service provider. This leads us to the question, how they can be combined in order to derive a changed Equivalence Class for a binding. Our approach is very simple: The service provider as well as the service user represent Equivalence Classes of the binding independently simply as a number.

The Equivalence Class for a binding is the tuple containing the service user’s view of the Equivalence Class and the service provider’s view of the Equivalence Class. We call this tuple a Combined Equivalence Class. Whenever the Combined Equivalence Class changes the behavior of the provided service or the expectations of the service user has changed. Therefore a runtime-compliance test needs to be executed then. The service user has associated test cases for his Equivalence Classes, which are executed, when the Combined Equivalence Class changes.

For our example this means, that VendorA and VendorB define Equivalence Classes for all provided and required services of their components. In the following we will look at the Equivalence Class definitions of VendorA’s biosensor as well as VendorB’s casualty unit.

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

VendorB provides a different Equivalence Class definition for the biosensor required by his casualty unit. He defines three Equivalence Classes: one where a sensor value is out of range (<0), one where a sensor value is zero, which may be due to a slipped off sensor, due to a malfunction, or due to a cardiac arrest of the casualty. The third Equivalence Class is the remaining range (both values >0).

These Equivalence Classes are depicted in Listing 1. The postfix _bioSensor in the method name specifies the required service, associated with this Equivalence Class definition.

```java
public int getEquivalenceClass_BioSensor()
    { int pulseRate = bioSensor.getPulseRate();
      BloodPressure bp = bioSensor.getBloodPressure();
      if (pulseRate <0||bp.equals(new BloodPressure(0,0))){
        return -1;
      } else {
        if (pulseRate == 0||bp.equals(new BloodPressure(0,0))){
          return 0;
        }
      }
      return 1;
    }
```

Listing 1. Implementation of getEquivalenceClass() for the Biosensor Required by VendorB’s Casualty Unit.

Next to the Equivalence Classes, VendorB defines test cases in a method equivalenceClassTest(), which is executed, whenever the state of a biosensor or casualty unit changes in a way, leading to an untested Combined Equivalence Class. The execution of these test cases should determine, whether the biosensor is still compatible.

In our case, VendorB’s casualty unit is not compatible to a biosensor in general, when a sensor value is out of range, therefore false is returned in this case. If the Equivalence Class indicates, that a sensor value is zero, the test case queries the biosensor and checks, whether blood pressure and pulse rate are equal to zero due to a cardiac arrest. If this is not the case, the test fails, otherwise the test passes².

```java
public boolean equivalenceClassTest_BioSensor(int userEquivalenceClass){
    if (userEquivalenceClass == -1){
      return false;
    }
    BloodPressure bp = bioSensor.getBloodPressure();
    int pulseRate = bioSensor.getPulseRate();
    return (pulseRate==0&&bp.equals(new BloodPressure(0,0)));
  }
```

Listing 2. Runtime-Compliance Testcase for the Biosensor as Defined by VendorB’s Casualty Unit.

In our example, we may initially face the situation, that pulse rate as well as blood pressure are measured correctly by the biosensor. In this case, the Combined Equivalence Class (Provider: 0, User: 1) will be calculated at runtime for the binding between biosensor and casualty unit. In this case, the test case defined above will pass. If the fingerclip slips off, the Combined Equivalence Class changes towards (Provider: 0, User: 0) triggering another runtime-compliance test. The test case execution fails, since the pulse rate is zero while the blood pressure is still above zero. Therefore we will remove the binding and may put the casualty unit into a configuration, where the triage class needs to be manually set by a medic.

Thus we are 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 our running system. Our approach therefore integrates a so-called testing mode.

Before runtime test are executed, all involved components³ are put into testing mode. Therefore these components know,

²Note, that the test case in our example is very simple in order to focus on the general principle of our approach.

³A component is involved, if it is directly or transitively connected to the component, which drives the test or which is currently under test.
that they cannot rely on the interaction with other components until this test mode is deactivated after test execution. This enables components to restore their state after test execution and to simulate effects.

IV. 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.

V. CONCLUSIONS AND FURTHER WORK

Reconfiguration, which means changing component bindings, is necessary for dynamic adaptive systems, since components may enter or leave a system at runtime. However proving the correctness of a binding at runtime is not possible without simplifications. Specifying interfaces in a domain architecture does not help either: there are several reasons, why vendors vary from such a standard interface. Our approach is based on testing components at binding-time as well as during runtime. This enables us to recognize incompatibilities of components before they occur at runtime. Thus we can mark bindings of incompatible components as invalid and chose a valid binding instead.

A proof-of-concept implementation of the approach has been integrated into our latest prototype of DAiSI. In the past we developed dynamic adaptive systems from various domains based on DAiSI [20], [21], [22], [11]. Based on the newly integrated DAiSI featuring runtime testing, we realized a dependable emergency assistance system, which was exhibited at CeBIT 2009 [23].

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 the 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 for example.

REFERENCES
