Dependable and Usage-Aware Service Binding

Holger Klus, Dirk Niebuhr, Andreas Rausch
Department of Informatics
Clausthal University of Technology
Clausthal-Zellerfeld, Germany
Email: {holger.klus, dirk.niebuhr, andreas.rausch}@tu-clausthal.de

***Abstract*—The internet is evolving from a global information network to an environment that offers services for all areas of life and business, such as virtual insurance, online banking, or entertainment. Such services are created frequently during runtime by service providers according to specific user needs and they operate in a network and service environment that provides unified access to virtualized resources. Since the number of available services increases rapidly, it is hard for a client to find appropriate services and to compose them to useful systems. Additionally, clients may use the services in changing situations and the set of appropriate services and their binding should change accordingly to meet the users' needs in every situation.**

In this article, we present an approach which enables the automatic context-aware binding of services during runtime to so called dynamic adaptive systems. For this purpose, we introduce an approach for checking semantical compatibility [1] of services followed by an integrated approach for usage-aware service binding. Finally, we present our infrastructure DAiSI which provides an integrated implementation of both aspects. DAiSI has been applied to an application prototype which is also presented in this article and which validates the applicability of the presented solutions.

***Keywords*-Service orchestration, service binding, dynamic adaptive systems, runtime testing, context awareness, runtime adaptation, user interaction**

I. Introduction

Software-based systems pervade our daily life – at work as well as at home. Public administration or enterprise organization can hardly be managed without software-based systems. We come across devices executing software in nearly every household. Increasing size, penetration and features of software-based systems have brought us to a point, where software-based systems are the most complex systems engineered by mankind.

Current research areas, like ubiquitous computing, pervasive computing, or ultra-large scale systems, want to enable the engineering of future software-based systems by sharing a common trend: Complex software-based systems are no longer considered to have well-defined boundaries. Instead future software-based systems are composed of a large number of distributed, decentralized, autonomous, interacting, cooperating, organically grown, heterogeneous, and continually evolving services. Adaptation, self-x-properties, and autonomous computing are envisaged in order to respond to short-term changes of these service-based system itself, the context, or a user's expectation. Furthermore, to cover the long-term evolution of service-based systems becoming larger, more heterogeneous, and long-lived, service-based systems so called dynamic adaptive service-based systems must have the ability to continually evolve and grow, even in situations unknown during development time.

Since the number of available services increases rapidly, the requirements of a service user – independent of the service user seen as human end-user or as another service requesting for services – regarding other services get even more and even harder to manage. As a result, a service user has to solve similar problems regarding service discovery, like the cumbersome and painful task to find information in the Internet with different web search engines and other, more or less sophisticated tools currently available. It is still the user (human or machine) who has to be the active part searching and browsing the World Wide Web like looking for a needle in a haystack. Similar problems arise in the Internet of Services: The service user has to be the active part searching in the Internet of Services for the best services. Once he has found service candidates, it's again up to him to bind and orchestrate these services to the intended dynamic adaptive service-based system. The following questions arise regarding this process:

- What services provide (in a dependable manner) the service user's required service functionality and quality?
- What services fit best to the actual service user's context?
- What services fit best to the actual service user's usage intention?
- What is required by the selected services?
- How can the service binding be established for a dependable and usage-aware orchestration?

The aforementioned characteristics of a dynamic adaptive service-based system are the reason that classical software engineering approaches need to be enhanced. In traditional software engineering many aspects like software architecture, testing, component configuration were considered during design time. In dynamic adaptive service-based systems several aspects, like for instance the service selection, binding and orchestration to the intended dynamic adaptive service-based system, can only be processed during runtime considering context information and service users' intentions.

In the following we will introduce our means to enable a (semi-)automatic binding of dynamic adaptive service-based systems. Thus we will first introduce the characteristics of dynamic adaptive service-based systems regarding service bindings. We will continue by giving an example of a dynamic adaptive service-based system in section III which is used to illustrate our methods of establishing a service binding in dynamic adaptive service-based systems. Section IV introduces our approach to achieve dependable, usage-aware service bindings by applying runtime-testing and considering the context during service binding. A conclusion rounds up the paper.

[1] Patent pending. Patent Nr. 10 2008 050 843.8, 8.10.2008

 ISBN: 978-1-61208-156-4

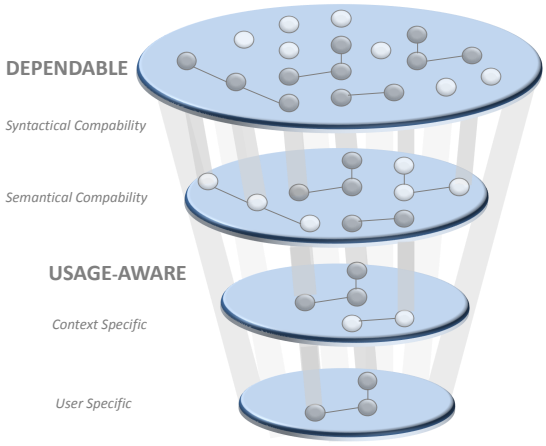

Figure 1. Dependable And Usage-Aware Service Binding

Figure 2. Sequence of Events During an Disaster

II. Dynamic Adaptive Service-Based Systems from a Bird's Eye View

The Internet of Services offers a frequently dynamically changing set of services for all areas of life and business. To fulfill the actual user's[2] needs with respect to it's context the proper dynamic adaptive service-based system has to be created from services contained in the cloud of services. Technical infrastructures for such a cloud of services are already available, like for instance OSGi [1] or SOPERA [2].

Using these infrastructures one has to manually define, select, bind and orchestrate required services respectively one has to implement a predefined set of possible configurations. Assume – due to continual evolution and growth of number of existing services in the cloud – a new, yet not considered, service appears, which would perfectly fit as an additional configuration option into the defined set of possible service binding and orchestration configurations. As long as the manually defined respectively implemented set of possible configurations will not be re-defined respectively re-implemented this new service will not be taken into consideration during selection, binding and orchestration of the services in the cloud to the desired dynamic adaptive service-based system.

For that reason, more sophisticated infrastructures have been developed recently in research and practice, like for instance Microsoft Extensibility Framework (MEF) [3], ASG [4], KER-META [5], or DAiSI [6]. These infrastructures provide support for the top level shown in Figure 1: Infrastructures like for instance MEF or DAiSI introspect the provided and required interfaces of existing services and service users in the cloud and try to automatically bind those that match on the syntactical level.

Obviously, to provide dependable dynamic adaptive service-based systems, guaranteeing syntactical compatibility in service binding is not enough. Therefore one also has to guarantee the semantical compatibility of a service binding more precisely a compatible behavior of a service user and its required service to be bound. Hence as shown in Figure 1 in the next level – semantical compatibility – only those required and provided services will be bound that have a compatible syntax and behavior.

As a result we can guarantee the dependability of a service binding although it will be established automatically during runtime. However we have to go further. It is not enough to establish only correct bindings: we also have to take care that only those service bindings that match with the actual context and an actual user's intention will be established. For that reasons on the next level, the context specific level, only bindings between those services are allowed that match with the actual context. And finally, to provide dependable and usage-aware service bindings, the last level selects those bindings from the remaining possible bindings, which match the current user's intention.

By applying this four level approach to reduce possible service bindings finally only dependable and usage-aware bindings are established. In the next section we will show these four levels within a concrete application scenario. We use it to illustrate the infrastructure-related challenges that we face, to enable dynamic adaptive service-based system with all characteristics described above.

III. Scenario

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 the number of casualties exceeds the number of medics by far. Thus medics need to get a quick overview of the whole situation before treating individuals. Therefore they do a quick triage [7], classifying the casualties regarding the severity of their injury, in order to treat casualties with serious injuries first. The sequence of events in case of such a disaster is depicted in Figure 2.

In our scenario, medics are supported by an IT system, enabling them to get a quick overview of the overall situation and to keep track of the physical situation of previously classified casualties. Medics come along with a medic unit and a bunch of casualty units. While a medic unit is a smartphone in our scenario, a casualty unit consists of vital data sensors. Each medic unit runs a medic application and each casualty unit runs a casualty service.

Whenever medics discover a casualty in field, they equip him with a casualty unit and thus the according casualty service of this casualty unit is started. The medic enters data regarding the casualty like his identifier, name, gender, or current position on a medic application. The data entered by the medic is stored by the casualty service.

[2]Note, users can be human users as well as other services requesting services from this service cloud.

 ISBN: 978-1-61208-156-4

Biosensors like pulse rate sensors or blood pressure sensors are part of a casualty unit in order to keep the triage class of the associated casualty up to date if his condition changes over time. Thus a casualty service executed on a casualty unit enables medics to capture and monitor a casualty's physical condition without needing to be physically present at his place. Within our example implementation, we used small sensor nodes [8] to realize casualty units.

After finishing the triage process, medics can use their medic applications to locate nearby casualties (respectively their services) or those, which need help most urgently. Information about these casualties is displayed by the medic application. This information is retrieved from the casualty services.

Figure 3 shows an excerpt of the domain architecture used for the emergency management system in our scenario. You can see, that casualty services provide a service interface CasualtySIf, which is used by medic applications. This interface provides access to the vital data sensors and enables service users to read and update the triage class of the associated casualty.

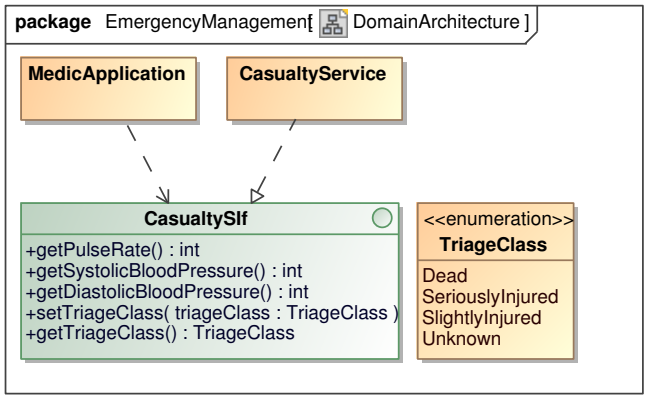

Figure 3. A Domain Architecture for Emergency Management Systems

We distinguish between two kinds of medics. One group of medics is responsible for the evacuation of slightly injured casualties while the other group takes care of seriously injured casualties. Casualties belonging to the aforementioned groups are displayed on the smartphones of the according medics.

Given that only casualties in the direct vicinity of a medic should be taken into consideration, only those are displayed on a medics' smartphone. Consequently medic get a filtered view of the situation.

This system is a typical dynamic adaptive service-based system. It consists of a vast array of services which are bound to service users at runtime: casualty services and medic applications. All services respectively service users can be provided by different vendors. Thus not each medic applications is compatible with each casualty service; To avoid malfunctions only compatible medic applications and casualty services may be bound to each other.

The overall system is evolving during runtime as new casualties may be integrated via their casualty service as well as casualty services may leave the system as casualties are transported to hospitals for further treatment. Next to this, medics acting as service users may enter or leave the system at any time.

We use a map depicted in Figure 4 to give you an overview of the situation which will be considered in the following. The map depicts all casualties, medics and their locations.

The map is divided into three areas: Area 1 is assigned to medic M1, area 2 is assigned to M2 and consequently area 3 is assigned to medic M3. The casualties in our example are classified into two triage classes: slightly injured casualties and seriously injured casualties. Slightly injured casualties are colored in blue, like casualty e, whereas seriously injured casualties are colored in red, like a, b, and c. According to the triage class, medics are responsible for a subset of these casualties. To keep it simple in our example all medics are responsible for seriously injured casualties.

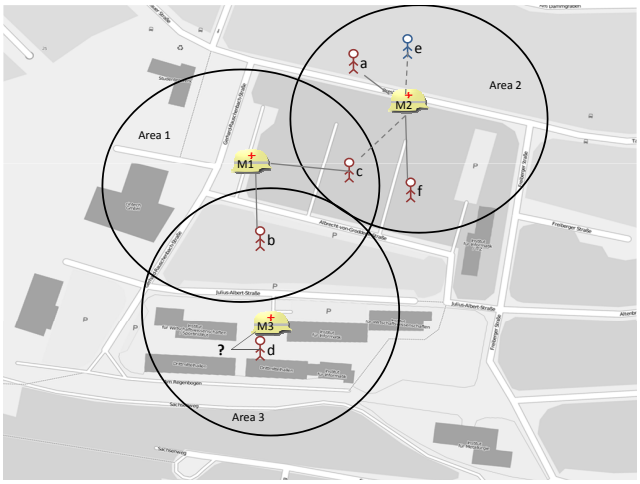

Figure 4. A Situation from our Application Scenario

Area 1 is allocated to M1; consequently he is responsible for casualty b and casualty c.

A closer look at area 2 points out, that medic M2 has the option to treat either casualty a or casualty f. Casualties c and e are not bound to his medic application. Casualty c is not bound to his medic application as he is semantically incompatible to M2. Casualty e is slightly injured and therefore not bound to his medic application as M2 is responsible for seriously injured casualties only.

Area 3 shows the scope of responsibility of M3. A characteristic of M3 needs to be considered: M3 is a medic, who has been at the disaster location by coincidence. He got slightly injured and now has to decide if he either wants to be treated as a casualty d or if he wants to help casualties as medic M3. The latter results in the fact that b is displayed on M3's medic application.

Summing up, casualty services can be bound to medic applications in different ways. This article describes the mechanisms to derive meaningful bindings. These mechanisms can be used to bind dependable and context aware systems at runtime.

IV. Challenges and Approaches in Dynamic Adaptive Service-Based Systems

In the following we describe the challenges associated with dynamic adaptive service-based systems. We illustrate these challenges by the means of our scenario introduced before and depict possible approaches.

Thereby we focus on challenges associated with providing an infrastructure for dynamic adaptive service-based systems which is capable of automatic system configuration.

 ISBN: 978-1-61208-156-4

A. Semantical Compatibility

During the scenario description we already identified several dimensions that may be considered to restrict the set of possible system configurations towards a reasonable size. The first dimension is the syntactic and semantic check.

1) Challenge: Dynamic adaptive service-based systems have to handle the fact that service providers as well as service users from various vendors may enter and leave the system at runtime. To allow service users to be bound to service providers at runtime, common domain architectures containing service interface specifications are provided. This causes the problem that even if all service providers adhere to this domain architecture, the system correctness cannot be guaranteed if service bindings are established which have not been tested in advance. This has two main causes:

1) A service provider implements a service interface in an incorrect way (*Incorrect service*).
2) A service user has implicit assumptions about an service exceeding the specification from the domain architecture (*Underspecification*).

Both causes are relevant in practice. On the one hand we may face incorrect services at any time, as we have an open system. On the other hand we need semantic underspecification in a domain architecture to enable vendors to provide service providers and service users with unique features.

Each service user may be tested respectively verified in combination with a specific service provider or even with various service providers. Once a service user is bound to a new service provider his assumptions might no longer hold resulting in an incorrect system configuration. Hence, the dependability of the resulting system can no longer be guaranteed, resulting in potential system failures. It is thus necessary to make additional requirements of service users explicit in a way enabling us to evaluate, whether a specific service provider fulfills them at runtime.

2) Approaches: The basic assumption of our work is, that the correctness of all possible system configurations of dynamic adaptive service-based systems cannot be verified respectively tested in advance. However, correctness of a specific binding between a service user and a service provider with respect to the service interface can be verified in advance. Hence, we still have to detect and prevent incorrect service bindings at runtime if we want to establish bindings which have not been tested in advance.

Our approach is the first step towards *dependable* dynamic adaptive service-based systems – since it is a testing approach we cannot *guarantee* the correctness of the system. Therefore it needs to be supplemented with a verification mechanism capable of proving a restricted set of safety-critical properties of the system at runtime.

The main question addressed by our approach is, whether a service provider is compatible to a service user. There are several approaches to provide (limited) statements regarding the correctness of service bindings:

- Enriching the domain architecture and its service interface specifications towards a complete specification. Therefore not only the service interfaces but also the required environment has to be specified in a semantically sound manner. This would mean, that the domain architecture contains a specification, which implies a single specific realization and leaves no space for unique features provided by single vendors. Consequently, there is no need for additional verification of the correct service binding during runtime as no different variants are possible. Hence, this is not a practical approach.
- If service user respectively service provider specify – not on the domain architecture but on the vendor specific implementation level – their required respectively provided service, they are compatible in general, if the specification of the provided service implies the specification of the required service [9]. As a consequence we have to prove the refinement relation between two specifications to ensure their compatibility. It is well known that for a specification technique with a high expressiveness (e.g. first order logic, turing machines) this can not be automatized [10], [11].
  If we limit the expressiveness of the used formalism (e.g. finite state machines, regular expressions, some forms of temporal logic) this relationship can automatically be proven [12], [13]. Following this approach, correctness cannot be guaranteed as we cannot specify some aspects – usually the critical ones – and therefore cannot verify them anymore.
- Using pre- and postconditions of the domain architecture to generate assertions, which are evaluated at runtime. This enables us to shut down the system, whenever an incompatibility of service bindings occurs. However we would like to be able to recognize these incompatibilities in advance and find an alternate service binding, enabling a seamlessly continued execution of the system.
- Checking the correctness of service bindings by bisimulation [14]. In this case we need to compare the states of two simulated system executions *for every system execution step*: one system is containing the service user and performs changes to its system state as specified in the required service specification, the other one is containing the service provider and performs changes to its system state as specified in the provided service specification. Beside the obvious massive performance problems in case of a recognized incompatibility the only solution is again shutting down the whole system.

Since none of the previously mentioned approaches is applicable at runtime very well, we propose integrating a runtime testing approach into a service orchestration infrastructure. Whenever a service user and a service provider should be bound together, test cases are executed in advance, which check, whether they match. Due to changing states of a service user / provider, additional tests need to be executed at system runtime as well. We need to show, that these test cases executed during reconfiguration are good enough to expose mismatches of service bindings. Whenever a test case fails, the blamable service binding can be deactivated and the remaining system can still be executed.

In the following we will consider service bindings between medic applications and casualty services from our scenario introduced before. In this scenario the casualty services act as service providers implementing the service interface CasualtySIf.

As already discussed in the previous sections, dynamic adaptive service-based systems, like the emergency assistance system in our application example, are based on a standardized domain architecture containing interface specifications as shown in Figure 3, For the application example this domain architecture needs to contain **interface** CasualtySIf. Medics use this interface to provide their functionality.

Since we want to build a dependable system binding service users and service providers developed by different vendors, the domain model may not only contain syntactical information like method signatures or datatypes occurring in the interfaces. It also may contain semantic specifications following the ‘Design by

Contract' [15] approach. To specify pre- and postconditions and invariants we may use mature specification techniques like the Java Modeling Language (JML) [16].

The specifications for **interface** CasualtySIf thus may be as follows. It specifies a method getPulseRate(), which must not return a negative value, indicated by the following postcondition: */*@ *ensures* (\ *result* >= 0)@*/*[3]

The same postcondition is specified for the two methods dealing with blood pressure: 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: */*@ *public invariant getSystolicBloodPressure() >= getDiastolicBloodPressure(); @*/.

Next to this, a medic application may use **interface** CasualtySIf to set the triage class by executing the method **public void** setTriageClass(TriageClass tc). This enables medic applications in our scenario to calculate the triage class based on the vital data queried from the casualty and set it directly at the casualty. Thus in the following a context-aware medic application can show for example only casualties which are seriously injured.

Imagine medic M2 from Figure 4 is equipped with a medic application, which calculates the triage class in a way, that it sets the triage class to Dead whenever a pulse rate of zero is returned by the casualty. It does not consider the blood pressure in addition. This is correct due to the medic application's implicit assumption that a blood pressure of zero is implied in this case as well. This assumption might have been derived directly from the service interface specification, if this is specified as follows: (getPulseRate()==0) <==> (getSystolicBloodPressure() ==0) <==> (getDiastolicBloodPressure()==0).

However the case might occur, that this assumption will not hold at runtime, since

- the vendor of the casualty service has implemented the service interface incorrectly (*Incorrect Service*)
- the domain architecture does not contain the invariant specified before and thus enables different implementations by different vendors (*Underspecification*).

In practice binding M2's medic application to a casualty, where this assumption does not hold might lead to misclassifications. Let's consider that casualty c's fingerclip measuring the pulse rate slips off. In this case casualty c would be classified as dead by M2's medic application although he still has a blood pressure greater than zero. This would lead to a situation, where no medic is sent to this casualty anymore. Thus it is crucial to detect this incompatibility between medic applications and casualties.

Now assume that due to dependability reasons dynamic checkers like the runtime assertion checker JMLRAC [17] are used during runtime. JMLRAC can be used to execute Java bytecode, which has been compiled by the JML runtime assertion checker compiler jmlc. This bytecode contains specified pre- and postconditions as well as invariants. JMLRAC checks during execution, whether these conditions are satisfied. If any condition is violated, it generates an exception containing, which condition has been violated. In the situation depicted above an exception will state, that an invariant of the casualty service has been violated: the blood pressure is not zero although the pulse rate is zero.

[3] *result* is a JML notation referring to the return value of the specified method.

If we consider medic M1, equipped with a medic application which considers blood pressure as well during triage class calculation, we will recognize another shortcoming. The medic application of M1 sets the triage class to Dead whenever pulse rate as well as blood pressure are equal to zero. This means, that no problem would appear, if this application interacts with the casualty service – even if the fingerclip slips off. Anyhow JMLRAC would state that the invariant of the casualty service has been violated although this would not cause any problems for a binding to this specific medic application.

As you can see, specifications in JML guiding a runtime assertion checker help us to detect the incompatibilities within our example. However we are not satisfied with the results due to major drawbacks: We may detect incompatibilities which are not relevant for the binding and we cannot detect incompatibilities in advance. Service users need to call a method of a service provider in order to realize, that it does not satisfy their assumptions. Therefore we can only detect wrong behavior 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 we expected exactly when we try to inflate the airbag due to a major accident. Thus it does not really help us to establish dependable service bindings. Instead we need to take all possible means to detect incompatibilities in advance.

To address these drawbacks, we provide an approach which enables us to automatically establish service bindings in a system using runtime tests to detect incompatibilities among service users and service providers. However we can only take advantage of such an approach, if we have a service orchestration infrastructure, which uses the test results during service binding. In the past, we developed the Dynamic Adaptive System Infrastructure DAiSI [6] aiming at dynamic adaptive service-based systems.

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

Tests need to be executed by DAiSI 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, the vendor of a medic application might specify such a test case for required casualty services. The test case simply queries the sensor and checks, whether the results are in the expected range. If the tests pass, the integration infrastructure can bind a tested service to its specific service user.

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

Our approach here is, defining equivalence classes regarding the state of the service binding. By equivalence classes we understand state spaces of service user and service provider, where the same behavior should apply. If we have these classes, runtime-compliance tests need to be executed each time, when the state changes in a way, that the equivalence class changes as well.

The service provider can define equivalence classes based on the control flow from its implementation. The service user can

define equivalence classes based on its requirements regarding a service provider. Both options alone do not help us: equivalence classes defined for a service binding by a service user can be very different from the ones defined by a service provider.

One reason for incompatibilities that we want to detect is, that the understanding of the service binding and therefore the resulting equivalence class definitions differ. Therefore areas, where the definitions of 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 service binding are defined by both: service users and service providers. This however leads us to the question, how they can be combined in order to derive a changed equivalence class for a service binding. Our approach is very simple: The service provider as well as the service user represent the equivalence class for a service independently simply as a number (e.g. an enumeration of different behavior expectations regarding the service binding).

The equivalence class for the service binding now 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 a combined equivalence class changes towards an untested combination, runtime-compliance tests need to be executed, since this means, that the behavior of the provided service or the expectations of the service user have changed. The service user needs to associate a specific test case with each of its equivalence classes. The specific test case associated with the current service user's view of the equivalence class is executed, if the combined equivalence class changes.

Looking at our approach from a more abstract point of view, the system executes a sequence as depicted in Figure 5 to determine, whether a specific service provider behaves as expected by a specific service user.

3) Evaluation: For our example this means, that the vendor of the medic application and the vendor of the casualty, define equivalence classes for the used respectively provided service.

The vendor of the casualty may for example decide, that its service provides the same behavior regardless of the internal state. Therefore he implements the getEquivalenceClass () method for both sensors in a trivial way: The provider's view of the equivalence class is 0 all the time (cf. Listing 1).

```
1  public int getEquivalenceClass(){
2    return 0;
3  }
```

Listing 1. A Trivial Implementation of getEquivalenceClass() for a Casualty.

The vendor of M2's medic application associates the equivalence classes of the required casualty service directly with the triage class it calculates for the casualty. The getter for the equivalence classes is depicted in Listing 2. The postfix _casualty in the method name specifies the required service, associated with this equivalence class definition.

```
1  public int getEquivalenceClass_casualty
       (){
2    return calculateTriageClass(casualty);
3  }
```

Listing 2. Implementation of getEquivalenceClass() of the Medic Application for Required Casualties.

Next to equivalence classes, the vendor of M2's medic application needs to define test cases, which are executed by the orchestration infrastructure, whenever the state of service provider or service user changes in a way, leading to an untested combined equivalence class. The execution of these test cases should determine, whether service user and service provider are compatible regarding this service binding in the current state space. Therefore the medic application contains a test case, which could be specified in a specification language like UML Testing Profile [18], [19] or TTCN-3 [20], [21] as depicted in Listing 3.

This test case states, that the medic application in general is not compatible to casualties, when the pulse is out of range or when the triage class is calculated to dead even though one of the sensor values is different from zero. In these cases the test fails, whereas it passes in all other cases[4].

```
1  testcase IsCasualtyCompatible() runs on
       Medic {
2    casualty.call (getPulseRate());
3    casualty.getreply (getPulseRate:{} value
         ?) -> value pulseRate {}
4    casualty.call (getSystolicBloodPressure
       ());
5    casualty.getreply (getPulseRate:{} value
         ?) -> value systolicBloodPressure
       {}
6    casualty.call (getDiastolicBloodPressure
       ());
7    casualty.getreply (getPulseRate:{} value
         ?) -> value diastolicBloodPressure
       {}
8    alt {
9      [] getEquivalenceClass_casualty() ==
         TriageClass.Dead {
10       if (pulseRate != 0 ||
           systolicBloodPressure != 0 ||
           diastolicBloodPressure != 0) {
11         setverdict (fail);
12       } else {
13         if (0<pulseRate<300 && 0<
             systolicBloodPressure<300 &&
             0<diastolicBloodPressure<300)
             {
14           setverdict (pass);
15         } else {
16           setverdict (fail);
17         }
18       }
19     }
20     [] getEquivalenceClass_casualty() ==
         TriageClass.SeriouslyInjured {
21       [...]
22     }
23     [...]
24   }
25 }
```

Listing 3. Testcase for Casualties as Defined by the Medic Application in TTCN-3.

If you now take a look at the situation from our example, we may initially face the situation, that pulse rate as well as blood pressure are measured by the sensors. In this case, the combined

[4]Note, that the test case in our example is very simple in order to focus on the general principles of our approach. Of course we can specify much more complicated test cases using our approach.

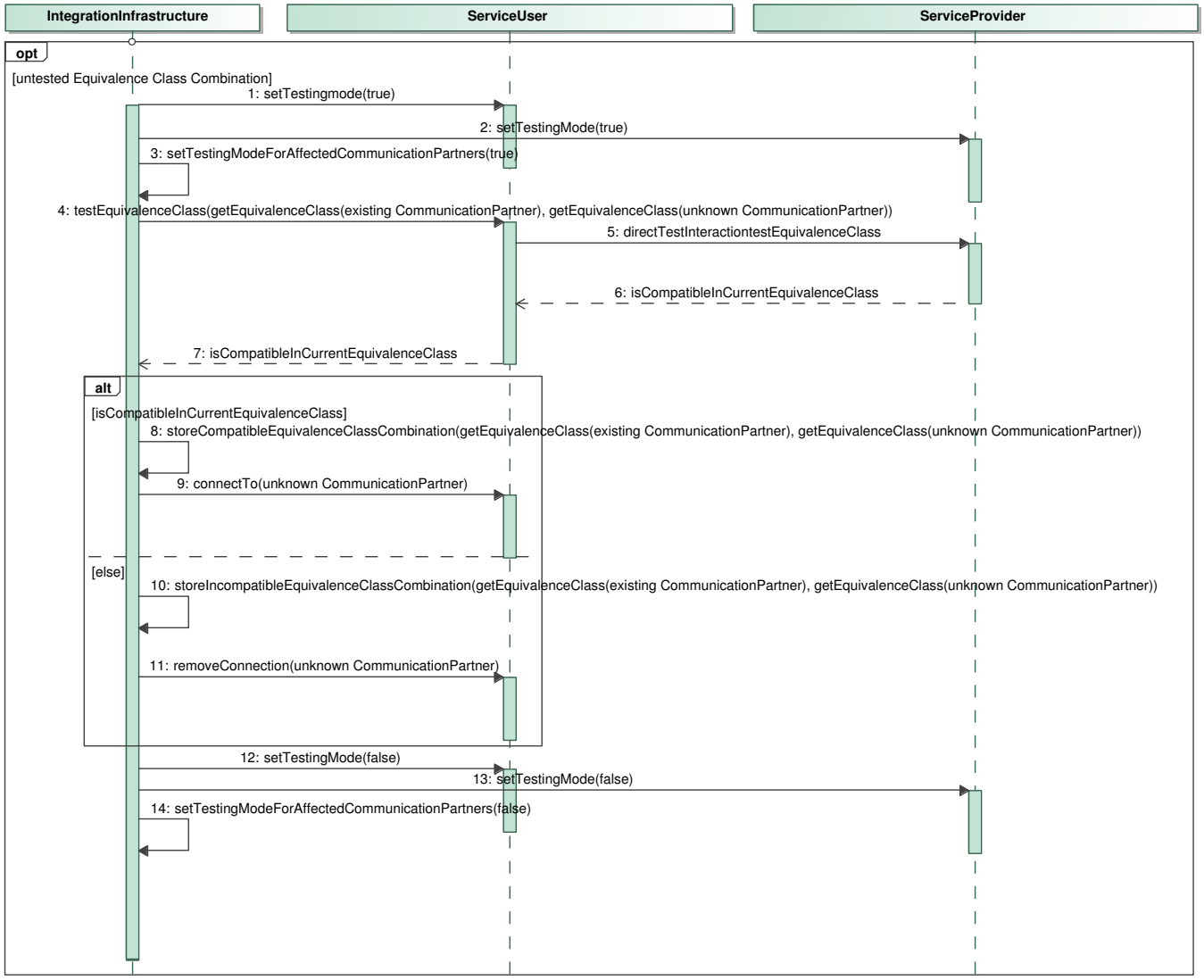

Figure 5. Sequence of the Runtime-Compliance Test in Our Approach.

equivalence class *(Provider: 0, User: 1)* will be calculated at runtime for the service binding between medic application and casualty. The test case defined above will query the sensor values and will pass. In case the fingerclip of the pulse rate sensor slips off, the combined equivalence class changes towards *(Provider: 0, User: 0)* triggering another runtime compliance test. The test case execution fails, since the triage class is calculated to dead although the blood pressure is still above zero. Therefore the orchestration infrastructure will remove the service binding between M2's medic application and the casualty service of casualty c – there might also exist a fallback mode, where the triage class can be manually set by M2.

The test case of M1's medic application would not fail, since it does not require that pulse rate and blood pressure need to be equal to zero at the same time. Thus the orchestration infrastructure would maintain the service binding between M1's medic application and the casualty service of casualty c.

As described above, we are able to detect incompatibilities in advance. However there is still a major drawback of the runtime-testing approach: Since we are testing the interaction between service users and service providers at runtime, we need to ensure that test case execution has no side-effects on our running system. Our approach therefore integrates a so-called testing mode.

Before runtime tests are executed, all involved service providers and service users[5] are notified and can transition into testing mode. Service users respectively providers in testing mode know, that they cannot rely on the interaction with other service providers until this mode is deactivated after test execution. This enables service users respectively providers to restore their state after test execution and to simulate effects (consider the airbag example sketched before: you do not want, that the airbag is inflated due to a test execution, therefore the code inflating the airbag must be substituted by simulation code in

[5]A service user respectively provider is involved, if it is directly or transitively connected by service bindings to the service user respectively provider, which drives the test or which is currently under test

 ISBN: 978-1-61208-156-4

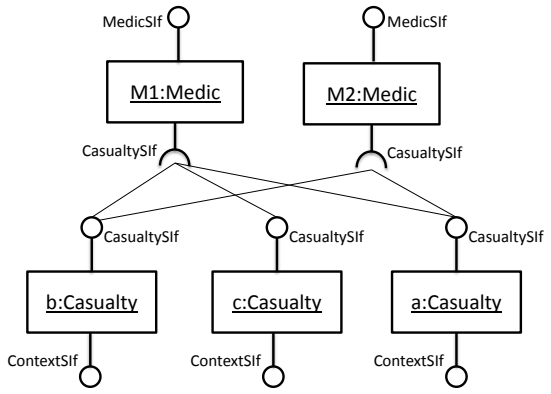

Figure 6. Considered Situation After the Check of Syntactical and Semantical Compatibility.

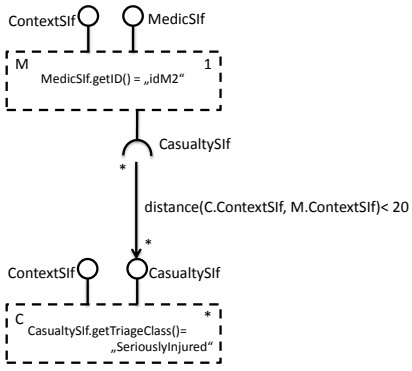

Figure 7. Two Placeholders Defining the Two Required Sets of Services.

testing mode).

Moreover we need to consider, that test-cases may be erroneous. This risk can be limited, if developers of service users use the same test-cases already during development to test mock-ups representing the third-party service providers. This enables the identification of erroneous test-cases at development time.

The runtime testing approach has been implemented within our DAiSI orchestration infrastructure. It enables us to detect incompatibilities and remove incorrect bindings at runtime. The scenario has been exhibited at CeBIT 2009. If you are interested in the application example, you can find a more detailed description at the exhibit's webpage [22]. A more detailed description of our approach and its reference implementation within DAiSI can be found in [23].

B. Context Awareness

In the section before we introduced a method for checking the syntactical and semantical compatibility of services. Another challenge is to use this information to bind those compatible services to useful *context-aware applications*. One solution for the aforementioned challenge is described in this section.

1) Challenge: After restricting the set of services and bindings with respect to syntactical and semantical compatibility, context information has to be considered, in order to bind the services to useful applications. A cutout of the situation after eliminating incompatible services is depicted in Figure 6.

We consider three service providers (the casualties a, b and c), and two service users (the medics M1 and M2) as shown in Figure 4. The depicted bindings between service providers and service users mean that they are syntactically and semantically compatible. Thus, the shown situation is a result of our runtime testing approach introduced in the previous section.

This binding is not useful in all applications respectively in all situations. A useful application in our scenario is to show only those casualties in the medic application which are located close to a medic and which are seriously injured. In Figure 4 you can see that casualty b is located out of range of medic M2 and that casualty e is only slightly injured. Therefore, the only valid binding regarding context awareness is to bind M2 to casualty a, as a is seriously injured and within range of M2. To constraint the service bindings on application level is the challenge we focus on in the following.

2) Approach: As indicated before, additional information about services is needed in order to refine the service binding. To get this information, we use the methods defined in the according service interfaces shown in Figure 3. Additionally we define a service interface called ContextSIf depicted in Listing 4 containing context information relevant for service binding. In our example this interface contains positioning information.

```
1  package de.cadaisi.repository.service;
2
3  import java.awt.Point;
4
5  public interface ContextSIf {
6    public Point getPosition();
7  }
```

Listing 4. The ContextSIf service interface.

We will now have a look at how to choose relevant services for a specific application considering this information first. Later we will introduce how the distance between medic and casualties can be taken into account, too. First we want to define that only those casualties should be connected to the medic which are injured seriously. To do this, we first define two sets of services, one which realizes the medic functionality, and the other realizing the casualty functionality. Figure 7 shows a graphical representation of the two sets of services.

Each dashed rectangle represents a set of services with common properties. We call these rectangles *placeholder*. During runtime these placeholders will be filled with services which meet specific constraints. In the upper-left corner of the rectangles an identifier for each placeholder is given, and in the upper-right corner the required cardinality.

The set of appropriate services is defined by several constraints. First it is given, that a service in placeholder M has to implement the MedicSIf and ContextSIf service interfaces and that it should require the CasualtySIf service interface. Furthermore, a call of the method MedicSIf.getID() should return a value equal to 'idM2'. In that way we pick a specific medic (M2) for the application. For the placeholder C we define similar requirements. A service placed here has to implement the CasualtySIf and ContextSIf service interface and the triage class has to be 'SeriouslyInjured'. Thus, only casualties remain which are seriously injured.

Because properties of services may change over time, the services filling the placeholders also change periodically. Our infrastructure is able to observe these properties during runtime and updates the sets of services accordingly. We call a placeholder activatable, if the number of required services, defined by the cardinality in the upper-right corner of each placeholder, is available.

 ISBN: 978-1-61208-156-4

Having defined these two sets of services, their bindings have to be defined next. In Figure 7 you can see a binding between M and C. In this case we define that each service in M can be bound to an arbitrary number of services in C and that each service in C can be bound to an arbitrary number of services in M. Because we already restricted the set M to exactly one service, the resulting application consists of one medic which is bound to an arbitrary number of casualties.

Furthermore, the binding defines one additional restriction, that is, the maximum distance between services in M and C. We call a binding activatable, if all defined restrictions are fulfilled by a binding. Our middleware considers these bindings as first class entities. It observes their properties and compares them with the according requirements during runtime.

The resulting application is runnable, if all defined placeholders and bindings are activatable. Again, our infrastructure automatically establishes the required bindings as soon as the user starts the application and the application is runnable. In the next section we describe our realization of this infrastructure in more detail.

3) Evaluation: Our infrastructure is able to handle the restrictions described before while considering the syntactic and semantic compatibility of the services. To do this, our developed infrastructure reads an XML based application descriptor where all constraints can be specified by the application developer. Such a descriptor looks like depicted in Listing 5.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <application>
4    <placeholder name="M" cardinality="1">
5      <provides name="MedicSIf">
6        <constraint property="getID()" value
             ="idM2" comparator="de.cadaisi.
             Comparator.equal" />
7      </provides>
8      <provides name="ContextSIf" />
9      <requires name="CasualtySIf" />
10   </placeholder>
11
12   <placeholder name="C" type="Casualty">
13     <provides name="CasualtySIf">
14       <constraint property="getTriageClass
             ()" value="SeriouslyInjured"
             comparator="de.cadaisi.
             Comparator.equal"/>
15     </provides>
16     <provides name="ContextSIf" />
17   </placeholder>
18
19   <binding service="CasualtySIf">
20     <source name="M" cardinality="*" />
21     <target name="C" cardinality="*" />
22     <constraint property="de.cadaisi.
           Context.getDistance(M, C)" value="
           20" comparator="de.cadaisi.
           Comparator.lessThan" />
23   </binding>
24 </application>
```

Listing 5. An Application Descriptor for the Emergency Management System.

In this XML file, two placeholders are defined which represent the placeholders described in the previous section. One requirement for services to fill placeholder M is that they have to implement CasualtySIf. Furthermore, a constraint is given which says that only services can fill this placeholder that return 'idM2' if calling the getID() method. A constraint consists of three parts: a property, a comparator, and a comparative value. Our infrastructure will call this method periodically on services of consideration and check whether the return value meets the condition.

We implemented some predefined comparators in our middleware, but the set of comparators can be enhanced by application developers in order to provide comparison of more complex objects. Finally, the required cardinality for this placeholder to become activatable is given. The placeholder C is defined similarly to M.

Additionally, one binding between these two placeholders is defined within the application descriptor. Here, first the service interface is given which should be bound. The source and target tag define the service user and the service provider, respectively. And finally, it is defined that the distance between service user and service provider should not be larger than 20. Our infrastructure calls the given method getDistance() and compares the return value with the given value, again using the predefined comparator class.

V. Conclusions

Binding service users to service providers in dynamic adaptive service-based systems at runtime is a hard task. Due to the diversity of service users and services, we usually have several options, how we may bind a specific dynamic adaptive service-based system from them. Dependability and specific usage situations help us, to filter meaningful bindings from these options. Thus we can derive meaningful service bindings by applying some simple mechanisms. As we did not encounter performance problems within our application examples, we did not evaluate the performance impact of our approach, yet.

We discussed two aspects of service bindings: semantical compatibility and context awareness. To identify semantical incompatible service bindings, we proposed a runtime testing approach. It is based on test cases defined by a service user.

The semantical compatibility of service bindings may change during runtime as the internal state of a service provider respectively service user. Since runtime testing only gives a snapshot of semantical compatibility at the time of test case execution, we propose to define equivalence classes for a service binding. An equivalence class states, that equivalent behavior of the associated service is provided respectively expected.

These equivalence classes are defined by the service users as well as at the service providers side. An orchestration infrastructure can monitor the equivalence classes and repeat test execution when one of those equivalence classes changes at runtime. Thus we can identify service bindings which are semantical incompatible and prevent that they are established at system runtime by our orchestration infrastructure.

The second aspect we discussed in this article was the context-aware service binding. Based on the identified syntactical and semantical compatible service bindings, a useful application has to be built which is able to react on context changes. To do this, we introduced the placeholder concept, where each placeholder defines a set of services based on common characteristics. These characteristics are on the one hand provided and required service interfaces. On the other hand we showed how we make use of information provided by service interfaces in order to refine the according set of services. Additionally, possible bindings between services have been restricted also using information provided by service interfaces.

 ISBN: 978-1-61208-156-4

Based on our existing infrastructure DAiSI [6] we realized an orchestration infrastructure, which is capable of deriving and establishing valid and meaningful service bindings regarding dependability [23] and context awareness. Using DAiSI we can establish service bindings in dynamic adaptive service-based systems automatically at runtime. Thus a vendor of a service-oriented application does not need to deal with service discovery and binding anymore, since this task is performed by our infrastructure.

An open issue regarding service binding is the involvement of the user. Despite considering dependability and context awareness, our orchestration infrastructure may determine multiple valid service bindings. Our infrastructure cannot reason about the quality of these bindings any further and thus will establish any of these bindings. However the user might be able to chose a service binding based on his specific goals. One of these situations occurs in our scenario; the medic M3 is injured and has to decide if he either wants to be displayed as a casualty or as a medic. Since the infrastructure cannot decide this, it makes a user-driven decision indispensable. This user-integration into the binding process is only at a conceptual stage at the moment.

ACKNOWLEDGMENT

The runtime testing approach for dependable service bindings 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.

This work was partly funded by the NTH School for IT Ecosystems. NTH (Niedersächsische Technische Hochschule) is a joint university consisting of Technische Universität Braunschweig, Technische Universität Clausthal, and Leibniz Universität Hannover. Furthermore, the work was partly funded by OPEN (Open Pervasive Environments for migratory iNteractive services), an VII Framework EU STREP project.

Many thanks to the reviewers for their helpful comments enabling us to improve this paper.

REFERENCES

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

[2] SOPERA, "Sopera enterprise service bus," http://www.sopera.com [Online; accessed 17-November-2009].

[3] Microsoft, "Managed extensibility framework," http://www.codeplex.com/MEF [Online; accessed 17-November-2009].

[4] 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)*. Oslo, Norway: Kluwer, Sep. 2004, pp. 261–274.

[5] Triskell Project, "Kermeta," http://www.kermeta.org [Online; accessed 17-November-2009].

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

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

[8] R. B. Smith, B. Horan, J. Daniels, and D. Cleal, "Programming the world with sun spots," in *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. New York, NY, USA: ACM, 2006, pp. 706–707.

[9] A. Rausch, "Software evolution in componentware using requirements/assurances contracts," in *ICSE 22, Proceedings of the 22th International Conference on Software Engineering*, Jun 2000.

[10] A. M. Turing, "On computable numbers, with an application to the entscheidungsproblem," in *Proceedings of the London Mathematical Society*, 1936, pp. 230–265.

[11] A. Church, "An unsolvable problem of elementary number theory," *American Journal of Mathematics*, vol. 58, pp. 345–363, 1936.

[12] B. Zimmerova, P. Vařeková, N. Beneš, I. Černá, L. Brim, and J. Sochor, "Component-interaction automata approach (coin)," pp. 146–176, 2008.

[13] A. Both and W. Zimmermann, "Automatic protocol conformance checking of recursive and parallel component-based systems," in *CBSE*, 2008, pp. 163–179.

[14] E. Estévez and P. R. Fillottrani, "Bisimulation for component-based development," *Journal of Computer Science & Technology*, vol. 1, no. 6, May 2002.

[15] B. Meyer, "Applying "design by contract"," *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[16] G. T. Leavens, A. L. Baker, and C. Ruby, "Preliminary design of jml: a behavioral interface specification language for java," *SIGSOFT Softw. Eng. Notes*, vol. 31, no. 3, pp. 1–38, 2006.

[17] 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.

[18] UML Testing Profile Webpage, http://tinyurl.com/yhxrhlp [Online; accessed 29-November-2007].

[19] P. Baker, Z. R. Dai, J. Grabowski, O. Haugen, I. Schieferdecker, and C. Williams, *Model-Driven Testing: Using the UML Testing Profile*, 1st ed. Springer, Berlin, 2007. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-72563-3

[20] Testing & Test Control Notation Webpage, http://www.ttcn-3.org/ [Online; accessed 29-November-2007].

[21] C. Willcock, T. Dei, S. Tobies, S. Schulz, S. Keil, and F. Engler, *An Introduction to TTCN-3.*, 1st ed. Wiley & Sons, Apr. 2005.

[22] D. Niebuhr, M. Schindler, and D. Herrling, "Emergency assistance system – webpage of the cebit exhibit 2009," http://www2.in.tu-clausthal.de/%7ERettungsassistenzsystem/en [Online; accessed 09-February-2009].

[23] D. Niebuhr, "Dependable Dynamic Adaptive Systems – Approach, Model, and Infrastructure," Ph.D. dissertation, Technische Universität Clausthal, 2010.