Dirk Niebuhr

Dependable Dynamic Adaptive Systems

Approach, Model, and Infrastructure

SSE-Dissertation 1

Software
Systems
Engineering

Institut für Informatik
Lehrstuhl von Prof. Dr. Andreas Rausch

TU Clausthal
Institut für Informatik
Software Systems Engineering

Dependable Dynamic Adaptive Systems — Approach, Model, and Infrastructure

Dissertation zur Erlangung des Grades eines Doktors der Naturwissenschaften

vorgelegt von
Dirk Niebuhr
aus Celle

genehmigt von der
Fakultät für Mathematik / Informatik und Maschinenbau
der Technischen Universität Clausthal

Tag der mündlichen Prüfung
09.08.2010

Vorsitzender der Promotionskommission
Prof. Dr. Jürgen Dix

Hauptberichterstatter
Prof. Dr. Andreas Rausch

Berichterstatter
Prof. Dr. Jörg P. Müller

SSE-Dissertation 1, 2010

Für Sabine

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Clausthal-Zellerfeld, am 22. Oktober 2010

Dirk Niebuhr

Contents

Dankbarkeit macht das Leben erst reich.

Dietrich Bonhoeffer

Vorwort

Diese Ausarbeitung stellt das mit Abstand umfangreichste Werk meines bisherigen wissenschaftlichen Werdegangs dar. Ohne die Unterstützung von zahlreichen Personen in meinem Umfeld hätte ich nicht die hierzu erforderliche Kraft aufbringen können. Daher gilt mein tiefster Dank all denen, die mich bei der Fertigstellung der Ausarbeitung unterstützt haben.

Ich bitte insbesondere diejenigen um Verständnis, die hier nicht namentlich genannt werden – eine solche Aufzählung wäre niemals wirklich vollständig, weshalb ich mich darauf beschränkt habe, einige wenige hervorzuheben.

In erster Linie gilt mein Dank meiner Frau Sabine, der ich diese Arbeit widme. Durch ihre schier unerschütterliche Geduld und das Ertragen meiner Launen in sämtlichen Phasen der Ausarbeitung hat sie den wesentlichen Anteil zu der vorliegenden Arbeit beigetragen. Insbesondere auch das Ermutigen zu Arbeitstagen außerhalb des Clausthaler Umfelds in Orlando, Trier oder Nienhagen haben letztlich maßgeblich zum Erfolg beigetragen. Darüber hinaus hat sie durch Reviews und durch die Hilfe bei der Umsetzung von Abbildungen nach meinen Vorstellungen – was mit Sicherheit nicht immer leicht war – zum Gelingen der Arbeit beigetragen.

Sabine: Das wir alle diese Arbeit jetzt in den Händen halten können, ist Dein Verdienst – Deine Unterstützung und Deine Liebe haben mir die dafür erforderliche Kraft gegeben. Es gibt keine Worte auf dieser Welt, welche die Dankbarkeit ausdrücken, die ich Dir gegenüber empfinde. Danke, dass Du für mich da bist!

Darüber hinaus danke ich meinen Eltern, die mir meine universitäre Ausbildung ermöglicht haben und somit die Grundsteine für diese Arbeit gelegt haben. Zusätzlich haben sie mir – ebenso wie meine Schwiegereltern – die Möglichkeit gegeben, die Arbeit außerhalb der eigenen Wohnung fortzusetzen und mir dabei viel Verständnis und Unterstützung entgegengebracht.

Ich danke meinen „Doktorvater“ Andreas Rausch, sowie seinem „Stellvertreter“ Jörg Müller. Beide haben in zahlreichen Diskussionen den Ansatz hinterfragt und dadurch schrittweise zu einer immer weiteren Verbesserung beigetragen. Andreas Rausch danke ich darüber hinaus für das hervorragende Arbeitsumfeld, was er geschaffen hat. Es ist mir ein Vergnügen bei ihm zu arbeiten und ich freue mich darüber, dass wir diese Zusammenarbeit auch in Zukunft fortsetzen.

Besonderer Dank gilt auch meinen Kollegen der Abteilung für Software Systems

Engineering – insbesondere denjenigen, mit denen ich in den vergangenen Jahren publiziert habe. Hervorheben möchte ich an dieser Stelle Holger Klus, der es wirklich lang mit mir in einem Büro ausgehalten hat und der gemeinsam mit mir zu den „Gründungsvätern“ der Lehrstuhlsäule Flexible Softwarearchitekturen gehört. Ich hoffe, dass wir auch in Zukunft unsere erfolgreiche Zusammenarbeit in der einen oder anderen Form fortsetzen können.

Neben wissenschaftlichen Diskussionen habe ich am Lehrstuhl stets auch fachfremde „Küchendiskussionen“ über Krokodillängen, Holzflugzeuge oder ähnlich absurde Themen genossen (Danke, Christian!). Für unterhaltsame Stunden im Rahmen zahlreicher Brettspiel-/ oder Tanzabende danke ich Hella Schäfer und Thomas Ternité. Beide haben darüber hinaus gemeinsam mit Thomas Bravin, Holger Klus, Sandra Lange, meiner Frau Sabine und Björn Schindler meine Zeit während der Promotion auch musikalisch versüßt.

Thomas Bravin und Annett Panterodt haben mir stets mit Rat und Tat zur Seite gestanden, und mir so die tägliche Arbeit erleichtert – vielen Dank für die stets schnelle Hilfe und für die Geduld die ihr mir gegenüber aufgebracht habt!

Aus meiner „Kaiserslauterer Zeit“ danke ich insbesondere Marcus Trapp, der mir durch zahlreiche Review-Anmerkungen enorm weitergeholfen hat – Diskussionen mit ihm, sowie seine Ideen haben die Ausarbeitung substantiell verbessert. Darüber hinaus, danke ich dem restlichen BelAml Team. Im Kontext dieses Projekt konnte ich den Grundstein der Implementierung der DAiSI legen. Ich drücke Euch – sofern ihr fußballbegeistert seid – die Daumen, dass der FCK sich in der 1. Bundesliga festsetzen kann.

Im Rahmen meiner Arbeit habe ich zahlreiche CeBIT-Auftritte begleitet. Mein besonderer Dank gilt hierbei den HiWis, die jeden dieser Auftritte zu einem Erfolg gemacht haben. Besonders hervorheben möchte ich an dieser Stelle vier HiWis, die mir aufgrund ihres außergewöhnlichen Engagements besonders in Erinnerung geblieben sind: Markus Heintz, Dirk Herrling, Mirco Schindler und Tim Schumann.

Zu guter Letzt danke ich den Partnern des RESIST Projekts bei Siemens: Cornel Klein, Jürgen Reichmann und Reiner Schmid. Ihr habt mich auf die grundlegende Idee der Verhaltensäquivalenzklassen gebracht und diese gemeinsam mit mir zur Patentanmeldung ausgearbeitet.

Kurzfassung

Heutige komponentenbasierte Systeme sind geprägt von hoher Dynamik. Begünstigt durch die zunehmende Mobilität von Geräten, auf denen Softwarekomponenten ausgeführt werden, treten diese Komponenten Systemen zur Laufzeit bei oder verlassen sie. Daher darf die Komponentenverschaltung innerhalb eines Systems nicht statisch, zur Entwicklungszeit festgelegt sein, sondern muss dynamisch anpassbar sein.

Die dynamische Anpassung der Systemverschaltung wirft Fragen bezüglich der Verlässlichkeit des resultierenden dynamisch adaptiven Systems auf. Da die zu integrierenden Komponenten dem System nicht vorab bekannt sein müssen, ist die Adressierung dieser Fragen komplex. Insbesondere ist es nicht möglich, im allgemeinen Fall zu entscheiden, ob eine Systemverschaltung semantisch kompatibel ist.

Diese Arbeit diskutiert Ansätze, wie verlässliche dynamisch adaptive Systeme, die zur Laufzeit aus unbekannten Komponenten verschaltet werden, ermöglicht werden können. Diese Ansätze basieren auf einem formalen Modell für dynamisch adaptive Systeme. Eine gangbare Lösung zur Etablierung von verlässlichen dynamisch adaptiven Systemen basierend auf Laufzeittests wird vorgestellt.

Auch wenn eine formale Verifikation stärkere Aussagen macht als dieser Ansatz, ist dieser im Gegensatz zur formalen Verifikation auch zur Laufzeit eines verlässlichen dynamisch adaptiven Systems anwendbar. Er erlaubt Laufzeitaussagen darüber, ob eine Systemverschaltung semantisch kompatibel ist, *bevor* eine inkompatible Verschaltung zu Problemen in der Ausführung führt.

In der Ausarbeitung ist der Ansatz anhand eines Anwendungsbeispiel aus der Katastrophenmanagementdomäne dargestellt. Die Infrastruktur DAiSI zur automatischen Verschaltung von verlässlichen dynamisch adaptiven Systemen wurde im Rahmen der Arbeit realisiert. Basierend auf DAiSI wurde das Anwendungsbeispiel, in Form von Einzelkomponenten, die von DAiSI zur Laufzeit zu einem Gesamtsystem zusammengesetzt werden, implementiert.

Aus Komponentenentwicklersicht erfordert der Ansatz nur geringe Anpassungen von vorhandenen Komponenten und schränkt die Implementierung nur minimal ein. Diese Anpassungen, wie beispielsweise das Entwickeln von Testfällen für jede verwendete unbekannte Komponente, führen jedoch zu einer höheren Verlässlichkeit des zur Laufzeit verschalteten Systems.

Abstract

Today's Component-based systems tend to be more and more dynamic. Due to the increased mobility of devices hosting Components, Components are entering respectively leaving a system at runtime. Therefore, a system's Component Binding, which is part of the System Configuration, needs to be changeable at runtime.

This dynamic adaptation of the System Configuration imposes Dependability issues, which are hard to address, since Components which need to interact in a system are not necessarily known to the system up-front. Therefore, it is not possible to determine a semantically compatible System Configuration in general.

In this thesis we will discuss approaches how to achieve Dependable Dynamic Adaptive Systems which are bound from unknown Dependable Dynamic Adaptive Components at runtime. These approaches are based on a formal model for Dependable Dynamic Adaptive Systems. We come up with a feasible solution to establish Dependable Dynamic Adaptive Systems, which is based on runtime testing.

Although this approach does not provide statements about the system, that are comparable with statements derived from formal verification, it is applicable in Dependable Dynamic Adaptive Systems at runtime of a system. It provides runtime-statements, whether a Component Binding is incompatible, *before* such an incompatible binding can lead to problems during system execution.

We will motivate and describe our approach at an application example from the emergency management domain. The infrastructure DAiSI – **D**ependable Dynamic **A**dapt**i**ve **S**ystem **I**nfrastructure – has been realized within this thesis. Based on DAiSI we implemented the application example as single Dependable Dynamic Adaptive Components. DAiSI binds these Components together at runtime resulting in the Dependable Dynamic Adaptive System described in the application example.

From a Component vendor's point of view, our approach only requires minimal changes to existing Components and makes only few restrictions regarding Component implementations. These few changes – like writing test cases for each used unknown Component – however, result in a higher Dependability of the overall system established at runtime.

Your task is not to foresee the future, but to enable it.

Antoine de Saint Exupéry

1 Introduction

To produce systems out of *IT Components*, Component-based development approaches have been elaborated and successfully applied over the past years. These approaches are changing the predominant development paradigm: Systems are no longer redeveloped from scratch, but composed of existing Components [Szy02, BRSV00].

Systems are composed of existing Components.

Nowadays, these IT Components are used within an organically grown, heterogeneous, and dynamic IT environment. They are expected to collaborate on their own without an explicit design of a system containing them. Thus, the System Configuration – containing Components as well as their Component Bindings – of these systems is not static anymore. It is neither explicitly defined at development time nor at deployment time. Instead these systems are expected to change their System Configuration over time as new Components enter the system at runtime.

The System Configuration changes at runtime.

Ultra large scale systems [FGG+06] or IT Ecosystems [HKNR08] are only two examples for future's system generations which are expected to change their System Configuration over time. These systems cannot be designed and built by a single Component vendor anymore. We will sum up these system generations in this thesis under the notion *Dynamic Adaptive Systems*.

By Dynamic Adaptive Systems we understand IT systems built from *Dynamic Adaptive Components*, which act as Service Partners meaning that they provide Services to other Dynamic Adaptive Components respectively use Services provided by other Dynamic Adaptive Components. Dynamic Adaptive Components are characterized by their continuous adaptation to a System Context or a User Context. You can think of a speech recognition Component, which adapts its speech recognition

Dynamic Adaptive Systems = Components + Bindings + Adaptation.

algorithm to the current physical stress of a speaking user (*adaptation to the User Context* [KR06]) or a printer Component, which can provide a copy-function after adaptation, whenever a scanner Component is present in the system (*adaptation to the System Context*).

A Dynamic Adaptive System consists of such Dynamic Adaptive Components connected via Component Bindings. A Dynamic Adaptive System is characterized by its openness towards entering Dynamic Adaptive Components. This results in a continuous adaptation of the *System Configuration* during runtime of the system due to changes in the System Context (newly entering respectively leaving Dynamic Adaptive Components) or in the User Context (a new user starts interacting with the system or a user changes his[1] interaction style ,for example, due to a new task). The System Configuration in a Dynamic Adaptive System needs to be established and updated at runtime since the context may change at any time during runtime.

Infrastructure support is necessary.

Due to this continuous adaptation of the System Configuration, support by a System Infrastructure capable of automatically establishing or updating a System Configuration is necessary for these systems. A supporting System Infrastructure enables developers of Dynamic Adaptive Components to focus on application features of the Component instead of having to deal with configuration related issues like Service discovery and binding as well.

Different vendors agree on a Domain Architecture, specifying Service Interfaces.

Dynamic Adaptive Systems are expected to be bound from Dynamic Adaptive Components developed by different vendors that only agree on a Domain Architecture instead of a complete System Configuration and its specific Dynamic Adaptive Components. The Domain Architecture describes Service Interfaces which specify interfaces between Dynamic Adaptive Components in applications of a specific domain. A Service Interface consist of a syntactical specification describing method names, parameters, return types, or attributes. In addition a semantical specification describing the behavior of methods or containing protocols constraining the allowed order of method calls may complement this syntactical specification.

These Service Interface specifications from the Domain Architecture enable vendors to develop Components which interact with third-party Dynamic Adaptive Components in Dynamic Adaptive Systems. They do not need to know these third-party Dynamic Adaptive Components in advance anymore – instead they can refer to Service Interfaces specified in a Domain Architecture. However, if vendors do not define Component Bindings explicitly anymore but only implicitly by dependencies defined in terms of provided and required Service Interfaces, a System Configuration is not defined at development time anymore.

In addition there is no single deployment time of the Dynamic Adaptive System, since Dynamic Adaptive Components should be able to enter a Dynamic Adaptive

[1] This is not meant to be gender-specific. Wherever you read “he” within this thesis, “she / he” is meant.

System after the initial deployment of the system resulting in a changed System Configuration. Due to these facts the integration of Dynamic Adaptive Components into a Dynamic Adaptive System by updating the System Configuration needs to be performed at system runtime.

A user of a Dynamic Adaptive System expects Dynamic Adaptive Components to collaborate autonomously with each other and provide a real added value to him. On the other hand, as Dynamic Adaptive Systems are prominent in many system visions of the future, we will soon get in touch with Dynamic Adaptive Systems in everyday life. Thus, we depend more and more on these organically grown Dynamic Adaptive Systems. Hence their Dependability gains in importance even though these systems have not been developed and tested in advance as motivated before.

We depend on Dynamic Adaptive Systems – thus they need to be *dependable*.

Therefore, the need for *Dependable Dynamic Adaptive Systems* comes up. They have cope with dynamically entering and leaving Dynamic Adaptive Components during runtime just like any Dynamic Adaptive System would. In addition they need to detect and avoid possible resulting semantically incompatible System Configurations during runtime. In order to achieve this, Dynamic Adaptive Systems require means to detect *semantical incompatibilities* between Components.

We need to detect semantical incompatibilities.

Whenever a Dynamic Adaptive Component has specified expectations regarding a required Service, which contradict the specific behavior of a syntactically compatible Service provided by another Dynamic Adaptive Component, we state that these two Dynamic Adaptive Components have a semantical incompatibility regarding this Service. Reasons, why these semantical incompatibilities occur in practice have been investigated in the context of architectural mismatch for systems which are built from existing parts at design-time [GAO95]. As a main reason they identified, that those parts have implicit assumptions regarding their environment that don't match the actual environment or conflict with the assumptions of other parts.

False assumptions cause semantical incompatibilities.

If we look at Dynamic Adaptive Systems, we want to be able to change the Component Binding at runtime while ruling out semantically incompatible System Configurations. One way of getting there would be to specify these additional assumptions in a Domain Architecture.

However, this does not help alone: vendors want to develop Components with unique features – semantical underspecification of the Domain Architecture helps us to achieve this. Underspecification of the Domain Architecture has the effect [Bro93, SS06], that we may face Components, which have additional assumptions regarding their environment, that are not specified in the Domain Architecture. This leads to semantical incompatibilities as other Dynamic Adaptive Components may not fulfill these additional requirements. Another reason for semantical incompatibilities is, that we cannot assume, that all Dynamic Adaptive Components adhere to the Domain Architecture in terms of a formally correct implementation of a Service

Domain Architectures are underspecified.

Interface [LTW+06].

Semantical incompatibilities need to be detectable at runtime.

If we allow, that Dynamic Adaptive Components may define additional requirements regarding required Services, we need to find a specification, which we can evaluate at runtime. Otherwise we would not be able to benefit from the specifications, as we need to derive a valid System Configuration at runtime in Dynamic Adaptive Systems.

Several research areas try to expose semantical incompatibilities. They address this by different approaches, which can be classified into design-time approaches and runtime approaches.

Design-time approaches like model checking or other design-time verification techniques try to expose semantical incompatibilities before Component execution by proving specific properties. Runtime approaches like runtime-testing require, that the evaluated Components are executed [Lig02].

Runtime testing enables us to detect semantical incompatibilities.

The approach presented here is a runtime approach using runtime-tests to expose and prevent semantical incompatibilities. Specific about our approach is the concept of Combined Behavior Equivalence Classes which are used in order to determine *which* Compliance Test Cases are executed *when* during system runtime.

In the following we sketch several future system visions prominent today. These system visions share the commonality that they focus on Dynamic Adaptive Systems. Since there are so many of these system visions, we are convinced, that Dynamic Adaptive Systems are the system generation of the future. Therefore, it is necessary to take means to achieve Dependable Dynamic Adaptive Systems.

1.1 Dynamic Adaptive System Visions

Today's system visions are Dynamic Adaptive Systems.

Nowadays several visions from industry as well as from research deal with Dynamic Adaptive Systems. All these visions have in common, that they assume, that it is not possible anymore to design and develop these systems as a whole, instead they need to evolve over time. Among them is the vision of Organic Computing, which is characterized by the idea, that mechanics from biology can be used for the establishment of Dynamic Adaptive Systems [fIGiVI03, BMMS+06]. By applying principles from biology these systems should provide so called *self-x* properties like self-configuration or self-healing.

Ultra Large Scale Systems [FGG+06] on the other hand deal with future systems, which exceed nowadays systems by far in terms of lines of code, amount of processed data, and by the number of involved Components. Establishing these systems, therefore, provides several challenges for the research community related to scalability of today's engineering approaches. In the following we will introduce three visions – Ubiquitous Computing, Pervasive Computing, and Ambient Intelligence – in more detail. All these visions deal with Dynamic Adaptive Systems,

which emphasizes, that Dynamic Adaptive Systems consisting of Dynamic Adaptive Components are relevant for future systems.

The vision of *Pervasive Computing* has been motivated mainly by industrial interests. The Pervasive Computing vision tries to cover the whole life with (mainly already existing) IT Components and use this coverage for creating economical benefit. These Components may be immobile like a home server or mobile like a personal digital assistant or a smartphone. Thus, we are talking of Dynamic Adaptive Systems – with Components entering or leaving a system – in Pervasive Computing as well.

The Pervasive Computing vision is highly correlated with privacy and security issues: on the one hand users of pervasive systems do not want that all their personal information is stored and processed electronically (privacy), on the other hand they want to make sure, that only authorized people can access their data (security) [Ken06].

If we want to achieve that systems following these visions are widespread in everyday life, it is indispensable that we take care about Dependability of Dynamic Adaptive Systems.

1.1.1 Ubiquitous Computing

The term *Ubiquitous Computing* was initially established by Mark Weiser in his essay "The Computer for the 21st Century" in 1991 [Wei91]. The vision of ubiquitous computing is shared among the American research community. By ubiquitous computing Mark Weiser refers to the *third era of computing*: the ubiquitous computing-era specified by the relation "one person, multiple computers". In this era the technology disappears for the users – it is becoming ubiquitous. This third era is also called "*Third Paradigm" Computing* by Alan C. Kay and follows the PC-era (one person, one computer) and the mainframe-era (multiple persons, one computer). The new era is sometimes called "post-PC-era" [Mat07, Pre99] as well, since it follows the PC-era. All three eras and the progress between them over the years are depicted in Figure 1.1 by showing the trend in total revenue related to selling hardware belonging to the three categories.

The post-PC-era is characterized by computing devices, which are embedded almost everywhere in a small and inexpensive way, like RFID tags in clothes for product identification, cups using sensors to monitor liquid consumption, or even microchips implanted under the skin for secure authentication or payment by touch. These distributed devices are expected to form larger systems by connecting over a (preferably wireless) network.

From todays point of view you can see, that several parts of this vision of ubiquity are already slowly becoming true:

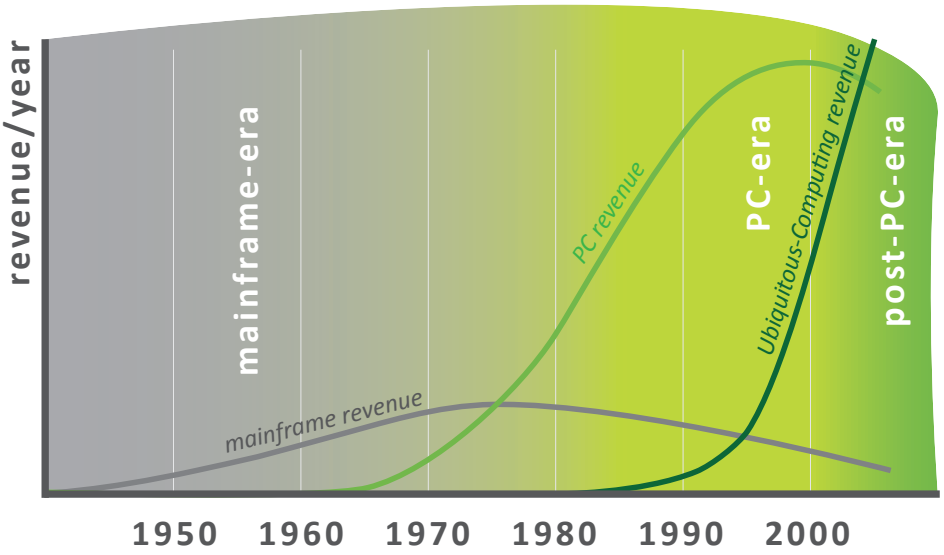


Figure 1.1: A Shift in the Computing Paradigm according to [Cor 6].

- RFID tags are already wide spread in high-priced products - this is currently limited to high-priced products due to the costs of these tags. As tag prices are currently steadily decreasing, heading towards a price of 5 ¢ per Tag [SS03] they are becoming more and more common in everyday products as well.

  In Rheinberg, Germany the Metro Group Future Store Initiative established an experimental store featuring RFID technology the so called Future Store [Bla04]. The whole product chain was covered by RFID-Tags. In 2008 they moved to a larger store in Tönisvorst for further evaluation. They also offer a mobile shopping assistant – a cell-phone based application [Ini09] guiding customers to products.

- Companies like Digital Angel [Ang10] sell microchips for identification of people as well as for access control, which need to be implanted. On the other hand, the movement of criminals may already be restricted by means like electronic shackles to overcome the problem of filled prisons.

- Modern cellphones provide mobile internet access to everybody. The Apple iPhone – the first cellphone featuring an affordable internet flatrate – gained a global market share of about 17 percent within a single year [Gar09] and, therefore, stressed the demand for mobile devices featuring an internet con-

nection. This emphasizes, that nowadays we are tending to be "connected" to a network wherever we are located.

Therefore, we recognize a progress towards the Ubiquitous Computing vision at the moment – we get in touch with devices carrying computing power in almost every situation of our everyday life without really recognizing them. The vision of Ubiquitous Computing requires the interaction paradigm to evolve, since the user should interact with an Ubiquitous Computer in a very natural way. Instead of interacting with a graphical user interface in a classical way he should interact with a Ubiquitous Computer itself by gestures or direct movement of Ubiquitous Computing devices.

Ubiquitous Computing is approaching.

We realize this change in the interaction paradigm slowly as motion sensors are integrated in many devices today like game controllers (Nintendo Wii) or mobile devices like the Apple iPhone or the iPod. They allow us to interact with a system simply by moving the interaction device.

It is obvious, that Ubiquitous Computing does not deal with a single computing device but with a bunch of devices featuring Dynamic Adaptive Components, which are interconnected to build a Ubiquitous Computing system. Since these devices may be mobile and may enter or leave a system at runtime, Ubiquitous Computing systems are Dynamic Adaptive Systems in the sense of this thesis.

1.1.2 Ambient Intelligence

Ambient Intelligence (AmI) describes a vision from European research [AE06]. It tries to combine the visions of Ubiquitous Computing and Pervasive Computing in order to establish an environment, where computation power and network access are available anywhere, embedded nearly invisible in everyday life and providing easy means for interaction. The Ambient Intelligence vision focuses on *intelligence*, that should be created by connecting several "dumb" Components. Establishing AmI systems should provide the user with a real added value compared to the sum of the values provided by each integrated Component. AmI systems try to achieve this goal by focusing on developing software and it's architecture.

First Ambient Intelligence systems have been established as prototypes [NSRD08, dRA04] and some Components from these systems like the Philips Ambient Light Technology [otN09] product line have evolved into series-production in recent years. Most of the prototypes cover the field of home automation where a flat is equipped with an Ambient Intelligence System which enables its inhabitants to query the contents of the refrigerator remotely or monitor its contents to detect spoilt food [KNR07], for example. Other features of these smart homes are mood recognizing lightning controls or multimedia applications like a visual mailbox integrated into the house.

Within systems from the Assisted Living domain sensors are attached to everyday devices in order to keep elderly people longer in their own home. Within the BelAmI project [fESE09] an elderly person's flat has been established as the Assisted Living Lab [NBKL06]. Within this flat elderly persons are supported by an Assisted Living system monitoring the fluid consumption of the elderly person which is measured by an intelligent cup [BGS01], for example.

These features may enrich the live of their inhabitants or may even enable inhabitants to live their life there. AmI systems are flexible regarding the specific Components involved in a system and Components may enter or leave these systems at runtime. Thus, AmI systems are Dynamic Adaptive Systems according to our definition as well.

Next to those AmI systems sketched before, AmI system prototypes have been developed within other so called *Assistance-domains* like *Assisted Training* [JTNK06] or *Assisted Working* [GSQ07].

In the following we will look at some application domains of Dynamic Adaptive Systems in more detail to motivate our approach for Dependable Dynamic Adaptive Systems.

1.2 Motivation

Nowadays, there are several trends towards Dynamic Adaptive Systems. Dynamic Adaptive Systems face the problem, that Dynamic Adaptive Components need to interact with heterogeneous and changing Service Partners (represented by third party Dynamic Adaptive Components) at runtime. As already motivated in Section 1.1 you can find these systems in several domains. We will introduce small examples from different domains in the following to illustrate our meaning of Dynamic Adaptive Systems. We will use these examples, to stress why we think, that Dependability is crucial for Dynamic Adaptive Systems.

One example for a Dynamic Adaptive System is a home entertainment system. Home entertainment systems in modern households include various devices like television sets, AV-Receivers, DVD-Recorders, cell phones, game consoles as well as associated remote controls. These devices share the commonality, that they can only use their full potential, if they interact with each other. Moreover you can recognize an increasing trend towards wireless communication among these devices. For home entertainment systems in an average household, it is typical, that they consist of devices manufactured by different vendors. Moreover they evolve over time, as devices are exchanged with new generation devices (e.g. a DVD-player is replaced by a BluRay-player) or new device types are added to the home entertainment system (e.g. a video projector is installed in the living room).

Home entertainment systems today can be connected at runtime, since their

interfaces are specified precisely. In the past these interfaces were mainly defined by analog signals. However, recent times show a trend towards standardized digital (software) interfaces like the High Definition Multimedia Interface HDMI between the Components. Unfortunately, as a result of this trend, we can recognize numerous situations, where the connection between home entertainment devices fails due to the increased complexity of the standardized interfaces. It is not natural anymore, that a BluRay player transmits its high definition video image to an LCD screen after connecting them [Tos09].

Semantical incompatibilities show up in practice.

Leaving the home entertainment domain, an interaction between different devices improves the value of the overall system for the user as well. If you think of a user shopping for groceries it would be a huge benefit, if his cell phone could display a shopping list based on the contents of his refrigerator. Connecting the cell phone to the shopping mall could enable it to display, where the next item on his list can be found, like it is realized in the mobile shopping assistant in the Metro Future Store [Ini09].

Integrating cell phones would increase the capabilities of a system in several other domains as well. For example, many modern cell phones are capable of GPS based turn-by-turn navigation on their own as well as of querying traffic information via mobile internet access. By integrating cell phones into a car system, these services could improve the car system by large. On the one hand, you can think of outputting route information and traffic jam warnings using the car stereo or offering a handsfree set supporting radio mute during phone calls. On the other hand you could offer turn-by-turn navigation, even if you are in areas where you do not receive a GPS signal like, for example, a long tunnel, since the navigation application could use the speed or the steering angle measured by the car telemetry.

Next to this, integrating personal communication devices like cell phones may offer value-added services in public buildings. At an airport you can think of checking in for a flight directly on your cell phone instead of having to enqueue with several other passengers at the check-in counter. After check-in you could be guided to your gate on your smartphone by using the infrastructure of the airport for indoor-navigation. Within the airplane, integrating your luggage with the system of the airplane could guarantee, that your luggage is really stored within the luggage compartment of the airplane. If you think of public offices, integrating personal communication devices could replace the registration at a front desk by a registration using your cell phone.

If you look at the previous examples, you can recognize, that all of them include the integration of various Dynamic Adaptive Components. A remaining question is, how this integration should be performed. The integration of Dynamic Adaptive Components into a Dynamic Adaptive System may be an operation performed by a system administrator or they may be seamlessly integrated without further user

interaction. The integration of these Dynamic Adaptive Components into a Dynamic Adaptive System should include as little user interaction as possible.This seamless integration is indispensable, as a Dynamic Adaptive System is supposed to integrate movable devices like smartphones, which may enter or leave a system at any time. You do not want an administrator to manually update the System Configuration each time a smartphone is entering the Dynamic Adaptive System.

Integrating cell phones into a system, however, is challenging, since they have a very short life cycle. While systems in a car or in a public building usually last for a long period, cell phones are replaced much more frequently. According to a study from Telephia, about 60 percent of all European cell phone users replace their phone within 2 years by a new one. 27 percent of the users even replace it within the first year [Tel06].

Due to the different lifecycles of the Components integrated in Dynamic Adaptive Systems we cannot assume, that the Dynamic Adaptive Components know their Service Partners at development time. For reasons of cost, an end user will only in rare cases exchange all Dynamic Adaptive Components involved in a Dynamic Adaptive System at a single point in time by a new generation. Even if he would like to exchange all Dynamic Adaptive Components, he might not be able to do it, as the operator of the airport cannot exchange the cellphones of all passengers.

Standards are required in Dynamic Adaptive Systems.

Therefore, Dynamic Adaptive Components like cell phones can not know their Service Partners at development time. Instead they can only interact with other Service Partners by using standardized interfaces (e.g. the Bluetooth specification [SIG07] – to be more precise: its Hands-Free-Set profile [SIG05] in case of cell phones).

Integrating Dynamic Adaptive Components into a Dynamic Adaptive System at runtime can have several negative side-effects, that need to be avoided. For example, no user would tolerate, that integrating his iPod into a system of his car would lead to a malfunction of his seat control or even cause the airbags in his cockpit to inflate. Although the speed information measured by the car is used by a user's cell phone, he would still want that the car displays the speed on its electronic speedometer.

Software systems are buggy – even if they have been developed as a whole.

By looking at the past, we can find several software failures, like the automated baggage system at Denver airport [Huc10]. Although these systems were built as a whole, severe failures occurred. Some of them were explicitly caused by reusing software Components like the Ariane 5 [Dow97] as programmers had a different understanding of the interface.

Dynamic Adaptive Systems bear even higher risks.

In Dynamic Adaptive Systems the risk that these incompatibilities will occur is even higher as in the examples for software failures before: vendors of Dynamic Adaptive Components only know a Service Partner in terms of his Service Interface instead of having the Service Partner available for testing during development. Therefore, they may not be able to recognize incompatibilities during development-

time.

Therefore, we need to guarantee at runtime, that integrating a Dynamic Adaptive Component does not have negative side-effects on existing functionality of a Dynamic Adaptive System. This is called *Dependable Integration* in this thesis. By performing Dependable Integration we are moving from Dynamic Adaptive Systems towards Dependable Dynamic Adaptive Systems. Dependable Integration characterizes the process, how a Dependable Dynamic Adaptive System establishes a System Configuration binding only Dependable Dynamic Adaptive Components which are free of semantical incompatibilities.

A runtime mechanism for Dependability is required.

Providing Dependable Integration is a hard task. Dynamic Adaptive Components are developed at different point in time and – in addition – by different vendors. Due to these facts, the specific System Configuration of a Dynamic Adaptive System cannot be tested or even proven to be free of semantical incompatibilities in advance. In this thesis, we assume, that there may be Dynamic Adaptive Components, which offer Services referring to Service Interfaces although they do not implement them correctly with respect to the Service Interface specification from the Domain Architecture.

This is a realistic assumption, since there are so many different vendors of Dynamic Adaptive Components that we can hardly assume, that all Dynamic Adaptive Components have been formally verified during development. Even if we would assume, that only correct implementations are available in the system drawbacks would occur: Vendors want to provide Dynamic Adaptive Components which offer semantically more specific behavior than specified in the Domain Architecture to distinguish from Components provided by third-parties. These specific semantics can only be exploited by other Dynamic Adaptive Components, if these have the possibility to specify, what requirements they have in addition to the specification from the Domain Architecture. In our approach, they could specify these additional requirements by Compliance Test Cases.

Formal verification alone does not help.

This assumption increases the demand for a runtime mechanisms establishing a Dependable System Configuration. Development mechanisms cannot provide a solution on their own as Service Partners for which a Compatibility needs to be proven are potentially unknown at development time.

Service Partners are not known upfront.

In this thesis we try to achieve Dependable Integration by changing the way, how Dynamic Adaptive Components specify their provided and required Services. They add Behavior Equivalence Classes defining state spaces, where a provided Service *behaves equivalently* respectively where a required Service *is expected to behave equivalently*. This enables us to provide a mechanism based on runtime-testing, which is capable to decide, whether two Components are semantically compatible regarding a specific Service Binding.

Behavior Equivalence Classes and Compliance Test Cases enable us to detect semantical incompatibilities.

For our previous example of the BluRay player and the LCD screen our approach would consider the BluRay Player as a Service Provider and the LCD screen as a

Service User. The LCD screen could define, that it expects a high resolution image, when a BluRay player plays a BluRay, whereas it expects a low resolution image, when it plays a DVD. In this case the state spaces *plays a BluRay* respectively *plays a DVD* are the Behavior Equivalence Classes from the Service User's point of view. The LCD screen could define a testcase for the Behavior Equivalence Class *plays a BluRay* testing, whether the video image stream is formatted as expected.

The BluRay player could define that it provides different behavior depending on whether it is playing a video or whether it is currently paused or stopped. In this case, the state spaces *plays a video*, *is paused*, and *is stopped* would be the Behavior Equivalence Classes from the Service Provider's point of view.

At runtime, different combinations of these Behavior Equivalence Classes can occur, like {plays a BluRay, plays a video} or {plays a DVD, is paused}. For each combination the test cases defined by the Service User are executed to detect incompatibilities. A Service User could define fallback strategies to enable graceful degradation in case of a failed test.

In our example, the LCD screen would recognize, that the BluRay player outputs a high resolution video image, which is not formatted as expected. Thus, it cannot use this Service provided by the BluRay player. As a consequence the LCD screen could use the low resolution video image, provided by the BluRay player for downwards compatibility.

Semantical Compatibility changes at runtime.

Our approach can not only decide about the compatibility once (prior to Service Binding) but instead monitor the compatibility as the states of the Dynamic Adaptive Components change during system runtime. However, due to this additional requirement regarding specification, we will call Dynamic Adaptive Components, which specify their provided and required Services in that way, *Dependable Dynamic Adaptive Components* within this thesis.

1.3 Goals of the Thesis

The main goal of this thesis is, that we want to provide runtime support for Dependable Dynamic Adaptive Systems. Therefore, we need to achieve several subgoals:

System Model

- We need to gain a deep understanding of Dependable Dynamic Adaptive Systems and the triggers, which threaten their Dependability. Thus, we want to elaborate a system model as a formal basis, which on the one hand enables us to describe the dynamics of these systems like the changing System Configuration, and on the other hand enables us to provide a mechanism, which maintains the Dependability of the resulting Dependable Dynamic Adaptive System.

- We need to provide a methodology, how we can decide, whether an Un-

known Service Partner[2] and a Requesting Service Partner[3] are semantically compatible regarding an Service offered by the Unknown Service Partner.

Methodology

This methodology needs to take into account, that Dependable Dynamic Adaptive Components are developed by different vendors at different points in time. Therefore, a methodology including a proof of semantical Compatibility *during development-time* is hardly applicable for this Compatibility decision, on its own. A runtime mechanism, deciding about semantical Compatibility, is needed.

- We need to provide a proof of concept implementation of a Dependable System Infrastructure, which is capable of establishing and updating a Dependable System Configuration of a Dependable Dynamic Adaptive System containing Dependable Dynamic Adaptive Components, which are compliant to our system model. This System Infrastructure needs to use the previously mentioned runtime mechanism during establishing respectively updating the System Configuration to avoid, that semantically incompatible Dependable Dynamic Adaptive Components are bound together.

Sample Implementation

- We need to provide a guideline for developers, how their Dynamic Adaptive Components need to be designed in order to benefit from our methodology. Therefore, we will provide an demonstrative application scenario and will provide example Dependable Dynamic Adaptive Components appearing in this scenario.

Development Guidelines

These goals are targeted within the thesis at hand in order to enable, that we can build Dependable Dynamic Adaptive Systems. In the following you will find a reader's guide, describing how the thesis is structured and how the different sections of the thesis can be mapped to general building blocks of a software systems engineering methodology.

---

[2] A Dynamic Adaptive Component acts as an *Unknown Service Partner* for other Dynamic Adaptive Components (Requesting Service Partners) when it is bound to them since its Current Configuration offers Services, which they requested.

It is unknown in terms of the implementation of the associated Service Interfaces, as Requesting Service Partners need to ensure, that the behavior of the implementation provided by the *Unknown Service Partner* is semantically compatible with the behavior they expected when they declared the Service Reference.

[3] A Dynamic Adaptive Component acts as a *Requesting Service Partner* when it declares Service References which may be bound to Services provided by other Dynamic Adaptive Components in their Current Configurations.

A Dependable Dynamic Adaptive Component acting as a *Requesting Service Partner* in addition needs to be able to decide, whether the specific Dependable Service to which its Dependable Service Reference may be bound by a Dependable Service Binding fulfills its expectations regarding the behavior.

1.4 Reader's Guide

The thesis you are currently reading is intended to be read as a whole. However, we tried to make it as modular as possible giving the references to the previous parts explicitly whenever required. Nevertheless there are at least two chapters which are "mandatory to read" from our point of view.

We strongly suggest, that you read Chapter 3 as it describes an application example, which is used throughout the thesis to depict the presented concepts. In addition you need to read Chapter 6 as it describes the main contribution of this thesis – our approach to Dependable Dynamic Adaptive Systems by applying runtime-testing. To gain a better understanding you may need to look up the references to our formal system model given during the explanation of our approach in Chapters 4 and 5.

Capital letters = Glossary term.

Several terms in this thesis are written in capital letters – like "Dependable Dynamic Adaptive Component". This way of expression specifies, that these are terms, which are specifically defined in this thesis. Whenever you are not sure, whether you precisely understand what is meant by a specific term, you can look them up in the glossary – it contains all of these terms in alphabetical order. In addition – if you are reading the digital PDF version of the thesis – these terms are hyperlinked, enabling you to directly jump to the definition in the glossary by simply clicking at them.

However, the glossary only provides a summary – no additional information about these terms is included. If you are reading the thesis as a whole you, therefore, only need to look up terms, in cases when you are not sure what was meant by a term anymore and don't want to jump back to the previous explanation in the thesis.

Next to this, you will stumble across some words written in *Italics*. This way of expression just stresses these words in the specific sentence, there is no additional semantics of these *Italics*.

In the following we will describe the structure of the thesis.

1.4.1 Structure of the Thesis

In this thesis you will find our approach of achieving Dependable Dynamic Adaptive Systems by runtime testing.

In Chapter 2 you will find the current state of the art regarding technology, which is required in order to establish Dynamic Adaptive Systems in general. It describes the state of the art regarding Dependability and regarding Dynamic Adaptive Systems. This chapter summarizes work, which is related to our approach.

Chapter 3 provides an application example from the emergency management domain. This application example describes a Dynamic Adaptive System, which is

used during the thesis to illustrate the introduced concept in more detail. We will show for this application example, why it is necessary to talk about Dependability when we want to bind such a Dynamic Adaptive System from Dynamic Adaptive Components at runtime. The application example has been prototypically implemented using our approach. We exhibited it at CeBIT 2009 [NSH09].

In Chapter 4 you can find a structural model for Dependable Dynamic Adaptive Systems introducing terms like system, Component, configuration or service interface, which describe how a system respectively a Component is structured. We capture how changes in the structure of a system respectively the structure of its Components influence Dependability by introducing so-called structural reconfiguration triggers.

In contrary to the structural part, the following Chapter 5 focuses on the behavioral part of the model. We capture how changes of the behavior of a system respectively the behavior of its Components influence Dependability by introducing so-called behavioral reconfiguration triggers.

In Chapter 6 we will explain our sample implementation of our infrastructure for Dependable Dynamic Adaptive Systems DAiSI. We will describe, how DAiSI realizes our formal system model and especially how it implements the reconfiguration triggers and the check of semantical Compatibility by runtime testing, which is the main contribution of this thesis.

A short summary discussing further work will round up the thesis.

In addition, the appendix contains a complete specification of our application example as well as its realization built as Dependable Dynamic Adaptive Components running on top of DAiSI.

In the following we will explain, how the contributions of this thesis can be mapped to basic building blocks of a Software System Engineering Methodology.

1.4.2 The Thesis in the Context of a Software Systems Engineering Methodology

Within the Software Systems Engineering research group at Clausthal University of Technology we sketched a software systems engineering methodology [Rau05b]. This methodology consists of several building blocks, which need to provide solutions in order to enable us to build Dependable Dynamic Adaptive Systems. These building blocks of a software systems engineering methodology are depicted in Figure 1.2.

In the following we will describe our contributions to these building blocks, which are necessary to enable Dependable Dynamic Adaptive Systems.

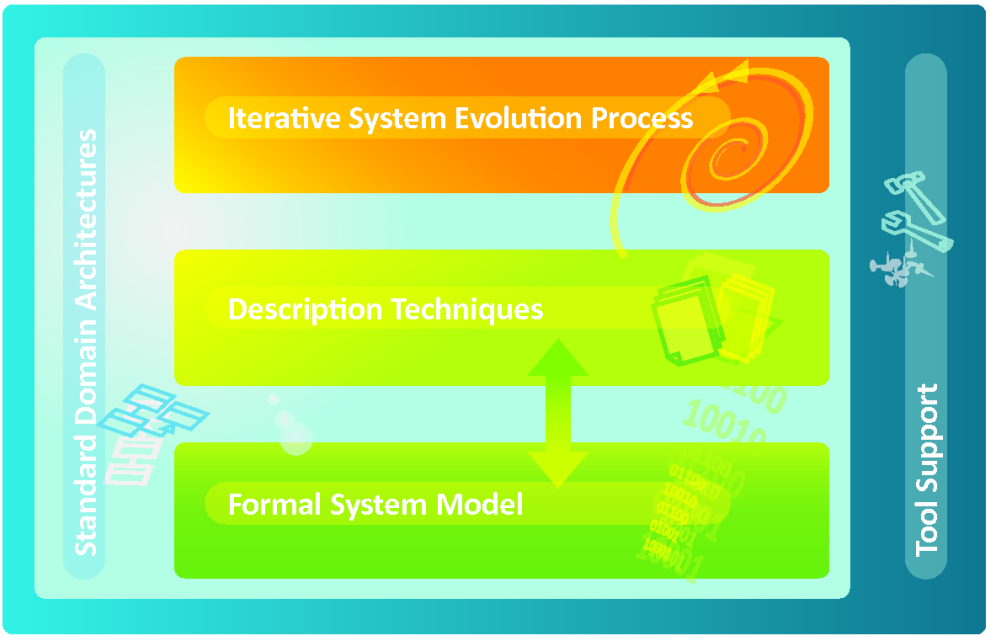


Figure 1.2: Building Blocks of a Software Systems Engineering Methodology.

Formal System Model

System model = foundation for approach and System Infrastructure.

A well-defined conceptual framework of Dependable Dynamic Adaptive Systems is required as a reliable foundation. It consists of a mathematical *formal system model* which is used to unambiguously express the basic notions and their relations, like for instance Dependable Dynamic Adaptive Components and integration of those within a Dependable Dynamic Adaptive System.

Within the thesis, Chapter 4 and Chapter 5 build the formal model for Dependable Dynamic Adaptive Systems. They are our contribution to the building block *formal system model*.

Description Techniques

Description techniques for Dependable Dynamic Adaptive Systems are required to describe the systems as well as the Components. The used description techniques have to be formally founded using the formal system model as the descriptions will be interpreted during runtime, like for instance to verify required guarantees during Dependable Integration.

UML is used throughout this thesis.

The building block *description techniques* is tackled during the chapters on the formal system model (Chapter 4 and Chapter 5). Within this thesis we use the graphical notation of the Unified Modeling Language (UML) and standards like

TTCN-3 [Ins07] or the UML Testing Profile [Gro07] to describe Dependable Dynamic Adaptive Systems by easily readable diagrams. We do not come up with our own description techniques.

Iterative System Evolution Process

To develop Dependable Dynamic Adaptive Systems neither a pure top-down nor a pure bottom-up development approach is sufficient. Usually, an *iterative system evolution process* is more appropriate. Thereby not only the development activities are organized, but also the complete life-cycle of the Dependable Dynamic Adaptive Systems is captured, like for instance analysis about the impacts of changes to a system's architecture. For Dependable Dynamic Adaptive Systems more and more steps of this process need to be shifted from design-time to runtime.

A shift of activities from design-time towards runtime.

Especially verification can hardly be performed at development time, as the interaction partners are not necessarily known at development time in Dependable Dynamic Adaptive Systems. Chapter 6 of this thesis addresses these verification means, which are needed at runtime. These runtime verification means are our contribution to the building block *iterative system evolution process*.

In addition, Chapter 7 contains a discussion of the impact, our approach may have on a software development process for Dependable Dynamic Adaptive Systems.

Standard Domain Architectures

Dependable Dynamic Adaptive Systems are open for entering and leaving Dependable Dynamic Adaptive Components, which were not known at the development-time of other parts of the system. Such an open and dependable system can only be realized based on a shared and open *standard Domain Architecture* designed from domain requirements. This Domain Architecture serves as a common blueprint for all valid System Configurations.

In this thesis we assume, that vendors providing Dependable Dynamic Adaptive Components for a specific domain have agreed on a standard Domain Architecture. We show at an example, how such a Domain Architecture can look like for the example in Chapter 3. This example captures syntactical as well as semantical aspects of the Service Interfaces of the domain.

Next to this, there is no noteworthy contribution to the *standard domain architecture* building block. As Domain Architectures are typically underspecified, since vendors want to provide Components with unique selling points, we concentrate on behavior, that is not specified in a Domain Architecture. We want to be able to check the semantical Compatibility of Components regarding this behavior, which has not been specified during standardization, as well.

Tool Support

All aspects should be empowered by *tool support*. The formal system model is realized by a Dependable System Infrastructure for Dependable Dynamic Adaptive Systems.

Our Dependable System Infrastructure DAiSI is a tool, which can be used to establish Dependable System Configurations within Dependable Dynamic Adaptive Systems automatically at runtime. DAiSI also contains tools monitor and administer a Dependable Dynamic Adaptive System. Thus, the description of DAiSI's implementation in Chapter 6 is our contribution to the *tool support* building block.

In the following we will introduce the state of the art regarding Dependability and Dynamic Adaptive Systems. We will distinguish our approach from related work.

Anything that is in the world when you're born is normal and ordinary and is just a natural part of the way the world works. Anything that's invented between when you're fifteen and thirty-five is new and exciting and revolutionary and you can probably get a career in it. Anything invented after you're thirty-five is against the natural order of things.

Douglas Adams

2 State of the Art

In the following we will give an overview of the state of the art from two perspectives: first of all, we will focus on state of the art dealing with Dynamic Adaptive Systems. This includes paradigms like Component-based systems or service oriented architecture.

The second perspective focuses on Dependability. We will discuss, how Dependability is defined, and how it can be achieved by different methodologies.

2.1 Dynamic Adaptive Systems

Dynamic Adaptive Systems in the sense of this thesis are characterized by the fact, that they consist of a set of (software) parts. These systems are dynamic, meaning that Components may enter or leave the system at any time during runtime. Consequently the binding between these parts needs to adapt during runtime. The parts together with their bindings form the System Configuration.

Dynamicity = Components enter or leave a system at any time.

Summed up the primary characteristic of a Dynamic Adaptive System in the sense of this thesis is a System Configuration which dynamically changes at runtime. The System Configuration must not be statically defined in advance during development but needs to be established dynamically during runtime by a System Infrastructure.

Dynamic Adaptive Systems are adaptive, meaning that they (respectively their parts) may behave differently according to the current context of the system. Thus, they adapt their behavior to fully benefit from entering Components, pro-

Adaptation = Different behavior according to System Context.

vide graceful degradation in case of leaving Components, or to adapt to a user's current needs.

According to the underlying system paradigm, the parts of a Dynamic Adaptive System are called Components or services. We will investigate the well-known paradigms of Component-based systems and service oriented architecture in the following. We will discuss, how they can enable a dynamically changing System Configuration.

2.1.1 Component-Based Systems

The basic idea behind Component-based software engineering is, that systems can be composed of off-the-shelf third-party-Components. Therefore, it is first of all necessary to define, what we mean when talking about a Component.

Component Definition

Component = independent deployment + subject to third-party composition.

Several attempts were made to define a Component in the past [BDH+98]. We will try to focus on the commonalities of these definitions in the following. A Component is the fundamental building block [Gro04a] of Component-based systems. Thus, C*omponents* represent the *parts* of a System Configuration in Component-based systems. According to [Szy02] a Component is a *unit of independent deployment* and *subject to third-party composition.*

Since it is independently deployed it should not have external deployment relationships towards other Components. This means, that it needs to be deployable on its own without requiring additional Components. If a Component has external relationships, these need to be made explicit, enabling third-parties to provide the required deployment environment for this Component.

No access to source code.

As a Component is subject to third-party composition, a Component vendor needs to consider the use case, that Components provided by him are composed to a system by third-parties, which do not have access to the source-code of the Component. Therefore, public interfaces of a Component need to be precisely specified and this specification needs to be available to the public.

Interface specification = syntax (mandatory) + semantics (optional).

The interface specification needs to contain at least a syntactical specification describing method signatures and exchanged datatypes. To give third-parties a more precise understanding of capabilities of a Component, its Component vendor can complement this interface specification with a semantical specification. The semantical specification may specify the behavior of interface methods as well as protocols defining the order of method calls.

Components need to be explicitly designed for reuse in order to benefit from the ideas of Component-based software engineering.

Component Reuse

By reusing Components in several systems, the overall quality of a Component should increase due to the many users, which may report faulty behavior to the Component vendor.

In order to enable reuse, it needs to be considered during development of Components. The interfaces need to be general enough allowing application in a large set of systems. On the other hand, they need to be specific enough to be applicable in systems of third-parties. Standardized domain architectures for applications of a specific domain may help during this trade-off between general and specific interfaces, since they define a set of interfaces, which may be used or implemented by Components targeting a specific domain. However, even standardization of domain architectures bears the risk of multiple competing standards covering the same domain [Szy02].

Trade-off between specific and general interfaces.

Component markets are an approach to facilitate Component-based software engineering. They provide a marketplace, where Component vendors sell their Components to third-parties or they acquire COTS (Commercial off-the-shelf) Components to build own Components or compose whole systems. However, these markets still don't fit the market participants' needs very well. Amongst others, Component vendors want to be able to discover, evaluate, and select COTS Components, analyze their cost and value and manage the overall acquisition process [US04].

If you take a closer look at the idea of Component markets, you can recognize, that they mainly are intended to sell COTS Components to vendors, which compose a whole (sub-)system from these COTS Components. This enables vendors to bind systems from COTS Components– optionally by adding self-developed glue-code or own Components. As a result, they get a composed (sub-)system which they can verify before selling it to customers.

In Dynamic Adaptive Systems, however, there is no single vendor, which may integrate and verify the system. Instead Dynamic Adaptive Components sold by different vendors are integrated into a running system at the moment, a customers starts them up by turning on a device hosting these Components. This is a very important difference and a huge threat for Dependability, since vendors cannot verify a (sub-)system as they cannot foresee each System Configuration occurring at runtime.

No system vendor in Dynamic Adaptive Systems – integration is performed by end-users.

There is a huge bunch of technology supporting the establishment of Component-based systems varying regarding the programming language of Components or regarding the underlying Component model. Prominent examples today are J2EE [Mic07b], CORBA [Gro04b], COM [Mic93] respectively its distributed version DCOM [Mic96], OSGi [All09], or .NET [Mic07a].

We will not discuss them further in the following. We will just highlight one new feature – the Managed Extensibility Framework (MEF) [Mic09] – of the latest

version 4.0 of the .NET Framework, which stresses, that Component-based systems deal with Dynamic Adaptive Systems.

MEF is made for Dynamic Adaptive Systems.

To establish a System Configuration at runtime the MEF provides a discovery and composition mechanism. An application in MEF is composed from so-called ComposableParts at runtime. Thus, MEF enables Dynamic Adaptive Systems. ComposableParts define required ComposableParts by so-called imports whereas the define their provided features by so-called exports.

Contracts associated with these im- and exports act as a filter during discovery of ComposableParts. Contracts in MEF are mainly defined by interfaces or abstract types, which a ComposablePart im- respectively exports.

Next to this syntactical declaration of im- and exports, metadata can be added to them. Metadata in MEF is a tag-value pair describing provided respectively required capabilities. It is used in MEF to further filter the set of matching parts during discovery.

Semantical Compatibility is only considered by tag-value pairs.

Tag-value pairs are quite widespread to filter results during discovery. Amongst others, you can find them in the Jini Lookup Service [New06] or in the Corba Trading Object Service [Gro00]. However, they provide only limited applicability to achieve Dependability: Obviously a (required respectively provided) behavior of a Component can hardly be captured by tag-value pairs. Thus, additional means are required to deal with Dependability of Component-based Dynamic Adaptive Systems.

Formal models like DisCComp [Rau07, Rau01] describing Component instances and their behavior in Component-based systems exist. DisCComp is capable of describing systems with dynamically changing System Configurations as well by describing snapshots of a synchronized system execution.

More powerful models exist – however, they are hardly applicable at runtime.

If we want to use these models, to decide about semantical Compatibility of two Components at runtime, we need to compare a specified provided and a specified required behavior of Components to be bound. To be more precise, we need to show, that the specification of provided behavior implies the one of the required behavior.

This implication is not calculable in general [Rau02]. [Rau05a], therefore, motivates the need for system tests during runtime in Dependable Dynamic Adaptive Systems. This approach is taken within this thesis.

2.1.2 Service Oriented Architecture

The Service oriented architecture (SOA) is characterized by the principle of service-orientation. The idea is to realize applications composed of several services. Each service is a so-called unit of logic [Erl05]. Thus, the underlying idea is separation

of concerns. According to [Erl05], services have the following characteristics[1].

- Services are reusable.
- Services share a formal contract.
- Services are loosely coupled.
- Services abstract underlying logic.
- Services are composable.
- Services are autonomous.
- Services are stateless.
- Services are discoverable.

Stateless Services versus stateful Components.

Comparing SOA to Component-based systems, we recognize, that a SOA service is the equivalence of a Component in Component-based systems. Many of the service characteristics are shared with those of Components. The main difference between them, is statelessness. While Components typically are stateful, a service in the SOA interpretation is stateless, meaning that it behaves identical all the time and its behavior is only influenced by passed parameters during a call of the service. You can find a deeper discussion of differences between SOA and Component-based systems in [Vog03].

Web Services

The most important application of SOA today are Web services. A Web service is defined using the Web Service Description Language (WSDL) [CCMW01]. A service requester typically retrieves web services at a central registry and directly communicates with these web services using SOAP (Simple Object Access Protocol) [W3C07] messages in the following as depicted in Figure 2.1.

The discovery mechanism in Web Services is related to our approach: During discovery, the service requester needs to submit his requirements regarding a service provider to the service registry. The registry needs to filter the registered services according to these requirements and return the matching service provider(s) to the service requester.

Semantical Compatibility is not addressed sufficiently.

WSDL originally only provided a syntactical specification of a Web Service. As semantical aspects also need to be considered to discover a matching service provider, this has been enhanced in specification languages like OWL-S [MBM+07] or WSMO [RKL+05]. To enhance service matching with semantics, they focus on describing pre- and postconditions of services. However, using these specification languages for service matching does not guarantee semantical Compatibility: They do

[1] For a more detailed explanation of these characteristics, see [Erl05], pages 290 - 311.

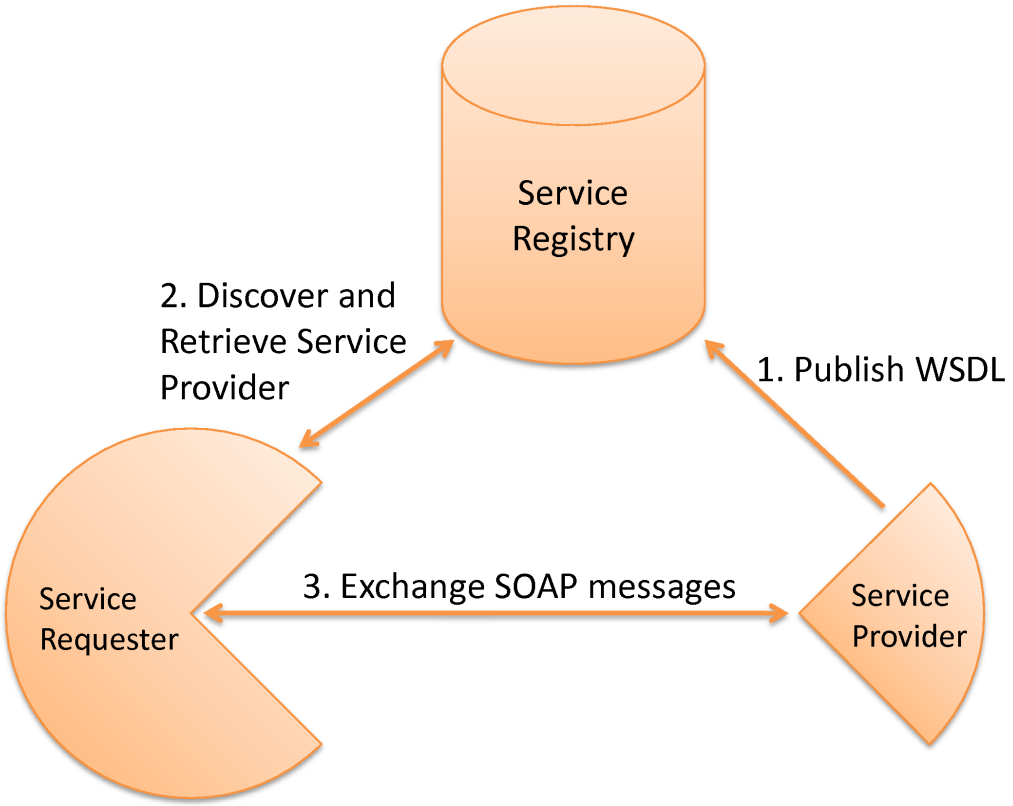


Figure 2.1: The SOA Triangle According to [MRP+07] or [Erl05].

not ensure, that a service provider guarantees the specified postcondition assuming that the precondition is fulfilled.

Summed up, Component-based systems as well as service oriented architecture provide discovery mechanisms as means to deal with Dynamic Adaptive Systems. However, these discovery mechanisms do only deal with semantical Compatibility – if at all – in a rudimentary way during matching a discovery request to a set of available Components respectively services. Moreover most discovery mechanisms require application developers to deal with discovery themselves – they need to submit their request and select a best fitting Component respectively service from the result set.

Within our Dependable System Infrastructure we want to provide a discovery and configuration mechanism, which extracts the provided and required Dependable Services of Dependable Dynamic Adaptive Components by a reflection mechanism and directly sets the Dependable Component Bindings between Components.

To be even more applicable for Dependable Dynamic Adaptive Systems, this mechanism should support adaptation of the Components themselves by embedding a Component model, which supports different configurations for a Component where a configuration is a mapping between provided and required Dependable Services.

Applying our discovery and configuration mechanism should consider the Dependability of the resulting system. This mechanism is implemented as a middleware service provided by the Dependable Configuration Component in our Dependable System Infrastructure DAiSI (Dependable Dynamic Adaptive System Infrastructure). Related to the definition of middleware services from [Ber96] this Dependable Configuration Component deals with system management as it acts as a configuration manager and as a fault detector within Dependable Dynamic Adaptive Systems.

A Dependability mechanism is required as a middleware service.

To explain, how we want to address Dependability, we define the meaning of Dependability in the sense of this thesis in the following.

2.2 Dependability

Dependability has been defined in [Sec92] as "the extent to which the system can be relied upon to perform exclusively and correctly the system task(s) under defined operational and environmental conditions over a defined period of time, or at a given instant of time". Some definitions of Trustworthiness refer to a similar concept when they define Trustworthiness as "assurance that a system will perform as expected" [ALR04].

[ALR04] gives an excellent overview of Dependability and the terms related to it. They define Dependability based on service failures. They define a service failure as an "event that occurs, when the delivered service deviates from correct service, either because the system does not comply with the specification, or because the specification did not adequately describe its function".

Dependability = ability to avoid service failure.

Consequently they define Dependability as the "ability to avoid service failures that are more frequent or more severe than acceptable". Within this thesis we want to detect these service failures, occurring at a Dependable Component Binding, and immediately change the System Configuration. We want to remove Dependable Component Bindings with service failures respectively replace them with bindings without service failures.

To achieve Dependability, [ALR04] distinguishes between three dimensions that need to be considered: Attributes, Means and Threats. Attributes of Dependability are Availability, Reliability, Safety, Confidentiality, Integrity, and Maintainability. In this thesis we focus on Reliability defined as "Continuity of correct service".

Reliability = continuity of correct service.

We address it by detecting service failures and reconfiguring the system to avoid catastrophic consequences and to reestablish correct service provision. Any-

how, since our approach is based on runtime testing, we cannot provide guarantees of correctness. We can only detect incorrect System Configurations if they are covered by defined test cases.

We remove faults and prevent error propagation.

Means of Dependability are fault prevention, fault tolerance, fault removal, and fault forecasting. We concentrate on fault prevention – means to prevent the occurrence or introduction of faults – and fault removal – means to reduce the number and severity of faults. By changing the System Configuration, whenever we detect faults, we *remove* faults from the Dependable Dynamic Adaptive System and *prevent* error propagation to other Components which could otherwise lead to subsequent faults.

Faults cause errors, which may lead to failures.

Threats to Dependability according to [ALR04] are faults, errors, and failures. According to [Lev95] faults are defined as "conditions, that may cause a reduction, or loss of, the capability of a functional unit to perform a required function". Thus, they act as triggers for software anomalies. An error describes a deviation of a unit's external state from its correct state. A failure is the inability of a unit to perform its function.

Summed up you can relate the three notions as follows: Faults cause errors, which may lead to failures.

Service failure, Service outage, Service restoration.

In addition, the notion of failure is also defined explicitly for Services: A Service failure is defined as "an event that occurs, when the delivered service deviates from correct service. The period of delivery of incorrect service is a service outage. The transition from incorrect service to correct service is a service restoration."

Considering these definitions, we concentrate on detecting state changes, which may act as faults. Whenever we recognize such a critical state change[2], we execute Compliance Test Cases to check, whether there is an error, which results in a Service failure.

Since we immediately change the System Configuration, when a Service failure is detected, we prevent Service outages and perform Service restoration as long as redundant service providers are present in the Dependable Dynamic Adaptive System. Even if no redundant service provider is present, we prevent the usage of this incorrect Service assuming, that it is better not to use a Service instead of using an incorrect one.

Potential faults are defined by Behavior Equivalence Classes.

Thus, we focus on detecting errors recognizable as Service failures. To detect these errors, we define potential faults by introducing state spaces of equivalent (expected) behavior of a Service using our concept of Behavior Equivalence Classes.

Summed up, the taxonomy of Dependability as defined in [ALR04] is depicted in Figure 2.2. Those aspects considered within this thesis are highlighted by depicting them in green color. Note that Failure is depicted in lighter green, as we only concentrate on *service* failures.

[2] In our approach such a change is characterized by the change of a Behavior Equivalence Class.

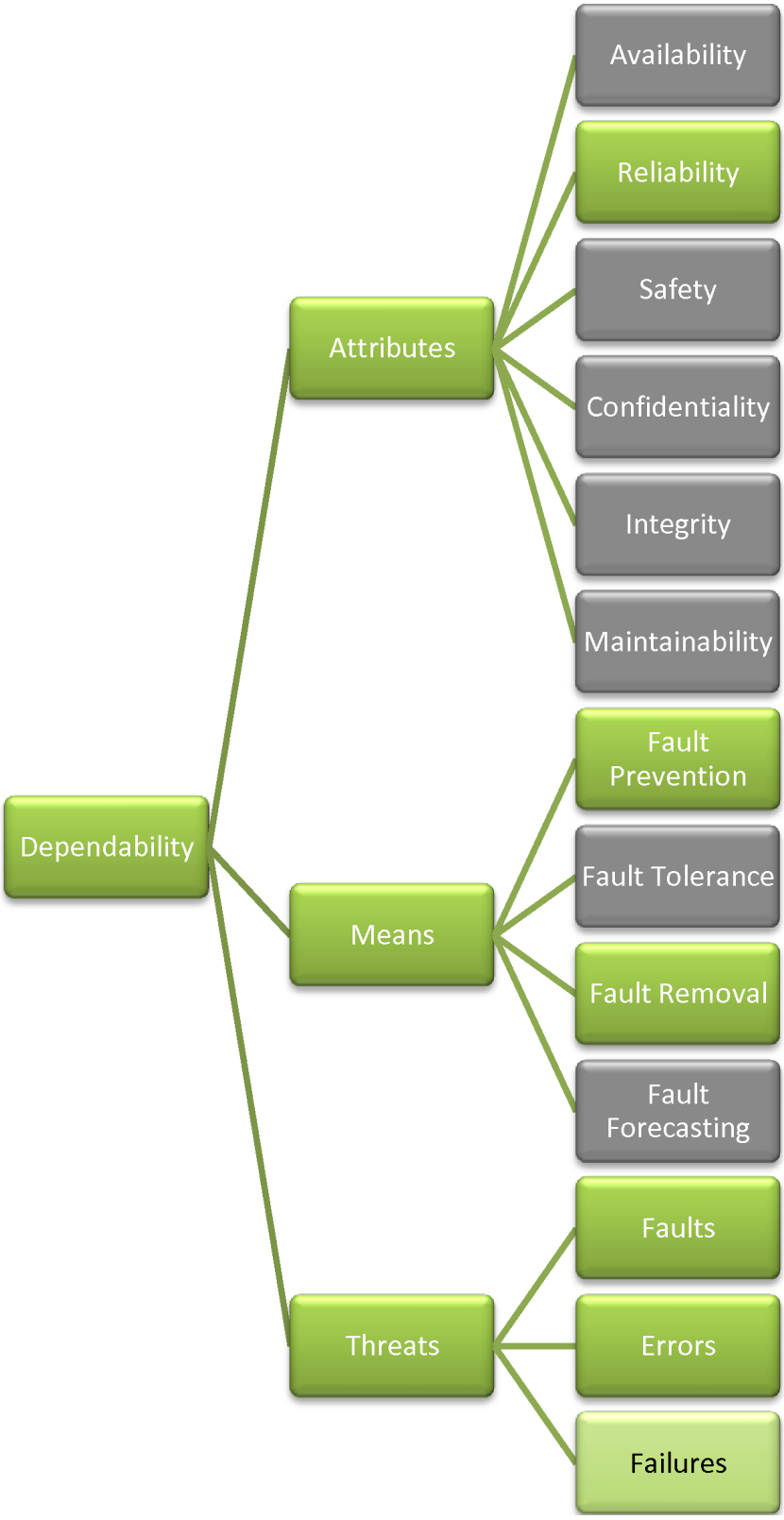


Figure 2.2: A Dependability Taxonomy According to [ALR04]. The Dependability Attributes Addressed in this Thesis are Highlighted in Color.

We need to deal with Dependability at runtime.

Dependability can be considered during the development phase as well as during the use phase. Since we cannot foresee the System Configurations in Dependable Dynamic Adaptive Systems at development time anymore, we propose to deal with Dependability at runtime of a Dependable Dynamic Adaptive System to detect service failures in a specific system at runtime. Thus, we focus on Dependability in the use phase. One failure class, that occurs at runtime are "operational, external faults" resulting from interaction of a system with a third-party Component.

We address this failure class by detecting service failures between Dependable Dynamic Adaptive Components within the Dependable Dynamic Adaptive System. 16 out of 18 failure types during operation of a system are operational, external faults [ALR04], which stresses the importance to address these type of failures.

Another classification related to Dependability is the separation between verification and validation. While verification tries to answer the question "Are we building the product right?", validation focuses on the question "Are we building the right product?" [Som06]. Thus, verification means checking, whether a product is compliant to a specification, while validation checks, whether a product fulfills the requirements of its users.

Verification or Validation? It depends on the Compliance Test Cases.

Since our approach uses runtime testing to establish Dependable Dynamic Adaptive Systems, it depends on the Compliance Test Cases, whether we perform verification or validation. If a Service User derives test cases based on his expectations of the functionality of a Service Provider, we perform validation. If he derives them from a specification of a Service Provider's behavior in a Domain Architecture, we perform verification. By deriving test cases from these two sources, we perform verification and validation.

To understand, which different verification and validation techniques exist, we will highlight an excerpt of the most relevant techniques based on the classification from [Lig02] in the following. This classification excerpt is depicted in Figure 2.3.

In general we can, therefore, distinguish between static and dynamic techniques for verification and validation. These are explained in the following.

2.2.1 Static Techniques

The idea of static verification techniques is to prove that a system has specific properties without executing it respectively to reveal errors. We distinguish analyzing techniques from verifying techniques.

Analyzing techniques like reviews or metrics, are performed without executing a system. They try to rate the quality of a system and may reveal errors. However, they do not provide general statements about Dependability of the system [Lig02] and are, therefore, not further considered within this thesis.

Verifying techniques try to derive / prove general statements.

The aim of verifying techniques is to provide such general statements about the Dependability of a system. While verifying techniques like symbolic execution

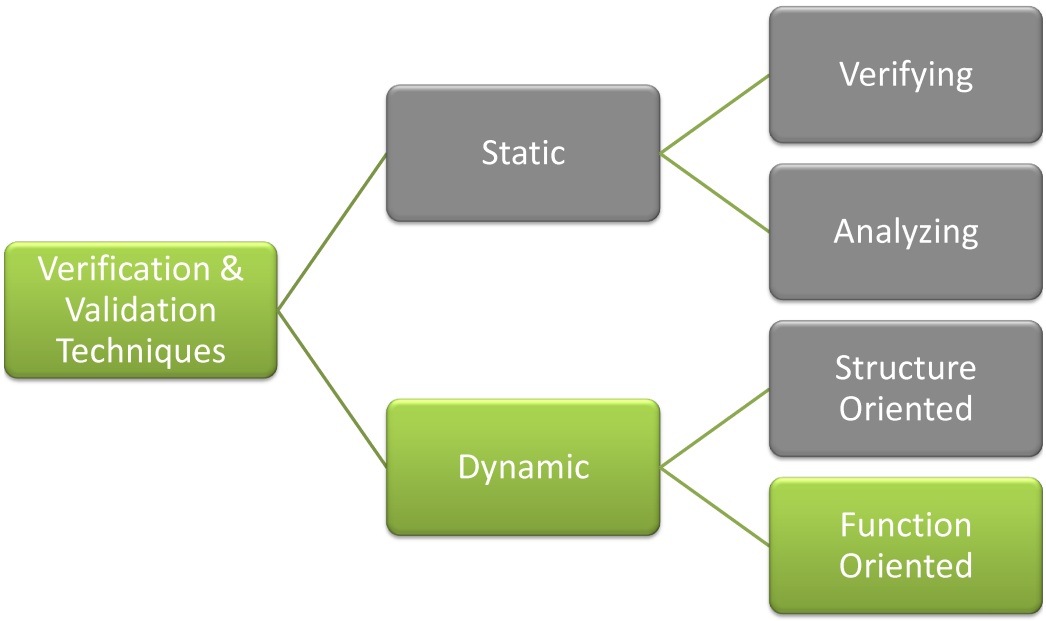


Figure 2.3: A Classification of the Most Relevant Verification and Validation Techniques According to [Lig02]. The Verification and Validation Technique Used in this Thesis is Highlighted in Color.

or formal proof techniques may provide such statements, there are some drawbacks, restricting us to use them as a single verification and validation technique in Dependable Dynamic Adaptive Systems.

They try to prove the compliance of a system to its specification. Since Dependable Dynamic Adaptive Systems are not specified as a whole, when trying to prove the correctness of a Dependable Dynamic Adaptive System, one needs to compare the behavior of a Dependable Dynamic Adaptive Component to the behavior desired by another Dependable Dynamic Adaptive Component.

Comparing a system to its specification is hardly applicable at runtime.

The behavior of a Component is usually given by the Component binary itself, as we cannot assume, that a Component vendor publishes the source code, as it would threaten his competitive advantage regarding other vendors. This is a challenge, as static verification approaches assume, that the source code or at least a model abstracting from the source code to facilitate verification by slicing the program for verification [Wei81], is available [FLL+02].

Even if Component vendors offered their source code: This usually is a Turing-

complete description and application-specific properties may be Turing-complete as well. Thus, two Turing-complete specifications need to be compared. This requires the evaluation of implications using second-order logic which refers to the decision problem. This has been proven to be not decidable in general by Turing and Church in 1936 [Tur36, Chu36] – neither at runtime nor at development time.

Thus, these techniques are hardly executable at runtime: Depending on the underlying specification that needs to be proven, these proofs may require human interaction, may perform badly as they are NP-complete, or may even be undecidable.

Trade-off between expressiveness and decidability.

Without restricting the expressiveness of the specification language, we cannot apply these techniques at runtime. Techniques, which could be applied at runtime, therefore, do not use turing-complete specifications. By using specifications of limited expressiveness, they focus on proving general properties like absence of deadlocks [BBSN08], whereas we want to check vendor-defined application-specific properties of Dependable Component Bindings.

Many static techniques reduce the specification expressiveness by using finite state automata [LPY97] or linear temporal logic for specification of the system respectively its provable properties. Although this is indeed necessary in order to be able to perform static verification, it is questionable, whether the remaining abstraction still enables vendors to express all properties, they want to specify.

Behavioral types [BCP07] are one example for a static technique to verify the correct interaction of software Components. They enhance syntactical type information by behavioral specifications like finite state machines or process calculi. However, [BCP07] explicitly states, that a good trade-off between verifiability and expressiveness needs to be found in order to make behavioral types applicable.

Many static checkers are unsound or incomplete.

Next to this problem of finding the right trade-off, many static checkers capable of static verification are unsound (reporting failures, which cannot occur) or incomplete (they miss failures). Users of static checkers like ESC/Java are annoyed especially by excessive warnings about non-bugs [FLL+02].

Testing cannot show the absence of failures.

While testing is incomplete as well, as it can only show the presence of failures, instead of proving their absence, it will only report real failures as long as the test cases are correct. Thus, some approaches like [CS05] combine static techniques with runtime testing by generating test cases to check, whether found failures really exist. However, they once again detect very general failures like dereferences to null, array-out-of-bound accesses, or divisions by zero.

Formal specification is a heavy burden for Component vendors.

Formal specification required for static verification and validation techniques is seen as a heavy burden by Component vendors [FLL+02]. [CS05] moreover stresses that testing is the "predominant way of discovering errors in software". Thus, our conclusion is, that using test cases to achieve Dependability in Dynamic Adaptive Systems enhances the chances for Component vendors to be able to reuse test cases they already used during Component development. At least they are

already capable of writing test cases and do not need to learn new specification techniques, which would have been the case, if we settled our approach on proving properties using any arbitrary formal specification language.

Writing Compliance Test Cases is well known.

Next to these problems, static techniques in Dependable Dynamic Adaptive Systems need to be performed at runtime, as the Components that are bound to each other within a specific System Configuration are not necessarily known at development time. Static checkers like jStar do not perform very well, needing about half a second to verify a program consisting of less than 100 lines of code [DJ08]. Such a performance would hardly be sufficient to analyze Dependable Dynamic Adaptive Systems at runtime.

Performance of runtime verification is bad.

2.2.2 Dynamic Techniques

In contrast to static techniques, dynamic techniques require the execution of a program to reason about its Dependability. Dynamic techniques can be classified further regarding the way, how test cases are derived for a system. In the context of this thesis, we focus on the two most relevant types of test case derivation [Lig02]: structure oriented techniques and function oriented techniques.

Structure oriented techniques in practice are used mainly during module tests [Lig02]. They focus on deriving test cases from the control flow of a system under test to get a sufficient test case coverage. Thus, they require this control flow, which is typically derived from the source code of the system respectively Component under test. As for static verification and validation techniques, this is hardly applicable, as we cannot assume, that the source code is available.

Structure oriented techniques can hardly be used in Dependable Dynamic Adaptive Systems.

As a consequence, they cannot be used to derive test cases with a good test coverage in Dependable Dynamic Adaptive Systems. However, dynamic techniques can be used to generate so called Behavior Equivalence Classes for a Dependable Service provided by a Dependable Dynamic Adaptive Component from the control flow.

Function oriented techniques derive test cases by interpreting a specification of the system under test. These techniques can be used for Dependable Dynamic Adaptive Systems. If a Component vendor develops a Dependable Dynamic Adaptive Component willing to access Dependable Services provided by an Unknown Service Partner, he will have a specification[3] for each required Dependable Service describing its desired behavior. Thus, he is able to derive test cases for Unknown Service Partners from these specifications.

Function oriented techniques – like equivalence class testing – are applicable.

One of the most known function oriented technique is equivalence class testing. The motivation of equivalence class testing is, to find a reasonable sized set of test cases, which still give a good chance to detect errors in the system under test. Thus,

[3] If this specification is not present explicitly as a document, then it will be at least in his head.

Equivalence class testing = test a *single* representative of each equivalence class.

it decomposes the whole testing domain[4] into several separate domains. The idea of this decomposition is, that the system behaves equivalently for all representatives of the specific domain. Assuming that these so-called equivalence classes represent the behavior of a Dependable Service adequately, you only need to test a single representative of each equivalence class.

We transferred this concept into runtime for our approach.

We applied the basic idea of equivalence class testing in our approach to Dependable Dynamic Adaptive Systems. As we need to execute tests at runtime, we want to reduce the set of test case executions. Thus, we introduced the concept of Behavior Equivalence Classes. At runtime, for each Dependable Service exactly one Behavior Equivalence Classes is active. In addition, for each Dependable Service Binding between Service User and Service Provider, exactly one Behavior Equivalence Classes is active.

Behavior Equivalence Classes are monitored at runtime. Changes cause execution of Compliance Test Cases.

Depending on the internal state of a Dependable Dynamic Adaptive Component, these Behavior Equivalence Classes can change at runtime. Such a change is used as trigger for new test case execution. The Compliance Test Cases for a Dependable Service are defined by the Component vendor willing to access this Dependable Service. They may differ depending on the Behavior Equivalence Class of the Service Binding. Summed up, our approach is a modified realization of equivalence class testing at runtime.

A related approach that already applied functional testing at runtime is [BB03]. They use built-in tests, to ensure correct functionality after system deployment. To test a specific function, they assume, that they can put a Component into a specific state during testing by calling a `set_to_state` method. This assumes, that Service User and Service Provider have an identical understanding of these states of a Service Provider, which is not necessarily given, if the specific Service Provider is not known at development time.

Design by Contract enables monitoring.

Another function oriented technique is monitoring. The idea of monitoring is, that specific properties are defined, which need to hold during system execution. This follows the *Design by Contract* methodology [Mey92]. In Design by Contract, a contract is defined between using and used Components. The used Components defines preconditions which need to be fulfilled when it is called. As a countermove, it guarantees a postcondition. In addition invariants may be specified, which hold during the whole system execution.

There are many specification languages aiming at the specification of such contracts for different programming languages, like JML [LBR06] or Spec# [BLS05]. They enable the specification of contracts and allow for generating monitors, that recognize a violation of these contracts at runtime.

Some of these approaches are even capable of dealing with properties of real

[4] A testing domain can specify the domain of input or output values. Consequently, there are some equivalence class testing techniques focusing on input values, while other focus on output values.

time systems like LARVA [CPS09]. However, they can only detect these violations at the moment, when they occur. Testing in contrast allows us to discover violations before they occur during system execution. However, monitors could be used as a test oracle during testing to indicate, whether a test passed.

Monitoring detects incompatibilities *at the moment, when they occur*.

[BMP07] provide such a monitoring approach, which could be used both – at development time as well as at runtime. Their approach is based on comparing message sequence charts, which have been generated from system execution traces, to a behavioral specification from a Component-based software architecture. Instead of using their approach to change the System Configuration, they only focus on detecting errors. Although they state, that this approach could be used at runtime, it seems to be hardly applicable due to its bad performance, as "the monitoring activity for a complex execution may require up to ten minutes".

Most specification languages for Design by Contract can be used for static techniques as well, as it is done in the Extended Static Checker for Java version 2 (ESC/Java2) [FLL+02].

By using dynamic techniques, a system's vendor can test the integration of specific Service Partners in advance during system development. These communication partners can then be integrated into the system at runtime, as the Compatibility could be checked in advance. This approach obviously explicitly excludes the integration of Unknown Service Partners. Nevertheless this is a common approach in practice. For example, automotive vendors like BMW define a set of cell phones, for which an integration into the car system can be guaranteed. These cell phones can be integrated and use the car stereo as a handsfree set.

***Testing in advance* enables Dependable Dynamic Adaptive Systems – as long as all Service Partners are known upfront.**

BMW tests the integration of these specific cell phones during development of the car or even proves, that the car's systems are still correct after the integration of the cell phone. Integrating cell phones, that have been developed after the car's development is a challenging task, if you follow this approach: After the release of a new cell phone you need to test the integration of it and need to update the in-car software of all cars on street when the cars are taken to a garage. This results in tremendous costs.

Another slightly modified application of this approach can be found in the area of personal computing, where the integration test is outsourced to the vendor of a Service Partner to be integrated. It showed up in the past, that the integration of device drivers into an operating system is a huge source of defect. If you look at MS Windows XP, for example, about 85 percent of all reported system failures were related to problems with drivers [SBL05].

Modern operating systems like MS Windows Vista, therefore, follow a special approach during integration of Plug&Play Components like USB printers, for example. As a precondition for the integration they require, that those drivers, that are needed to integrate these new Components are *certified* by the vendor of the operating system. At Microsoft, tests that need to be passed in order to receive the

Certification in operating systems is based on *testing in advance*.

certificate are defined by the Windows Hardware Quality Labs (WHQL) within the Windows Logo Program [Cor08b].

A vendor of a new Component executes these tests and submits test protocols at WHQL together with the driver. Microsoft then may execute additional tests and finally issues the certificate. The driver will then be delivered to end users together with this certificate by installation media or a mechanism like Windows Update [Her05, Cor08c]. The operating system, therefore, is able to integrate Unknown Service Partners, since it can be sure, that the integration has been tested in advance.

Negative side-effects are not ruled out by testing in advance.

However, this only tests, whether the integration of a new Component like a USB printer has negative side-effects regarding the core-functionality of the operating system. It cannot exclude negative side-effects regarding other integrated Components. Therefore, it cannot guarantee, that a previously integrated USB scanner still works without flaws after integration of a USB printer or that the combination of USB printer and USB scanner offers a new feature "*copy*" in a dependable way. Moreover, the time-span required for certification extends time-to-market for new products.

End user driven tests postpone testing towards deployment time.

Another option is letting the end user execute the integration test. For example, an end user needs to initiate a test print after integration of a new printer to check, whether a newly integrated Component works properly. Another example can be found during the installation of modems, where end users check by the push of a button, whether physical communication with the modem is possible.

These end user driven integration tests have several disadvantages: In general they work only properly with previously known Component types, as the integration test is already built into the operating system as a Component-specific test.

Disadvantage: The end user is responsible.

Additionally end user driven tests transfer a part of the responsibility to the end user. They need to evaluate the correct test execution – for example, they need to state, whether a test print is correct or whether a new software product installed correctly as depicted in Figure 2.4.

Therefore, they assume pretty rudimentary communication protocols like sending and receiving test strings, that can be evaluated by end users. Also these integration tests only check, whether newly integrated Components work properly while neglecting negative side-effects regarding other Components. The integration test after integrating a modem, for example, does not guarantee, that a printer still works properly.

In operating systems tests of Service Partners are not only present when integrating Plug&Play Components but also during installation of new software products. MS Windows Vista, for example, requires end users to confirm that a program installed properly, whenever you install a software, that has been intended to work with a previous version of MS Windows. For this reason the so called *Program Compatibility Assistant* has been created.

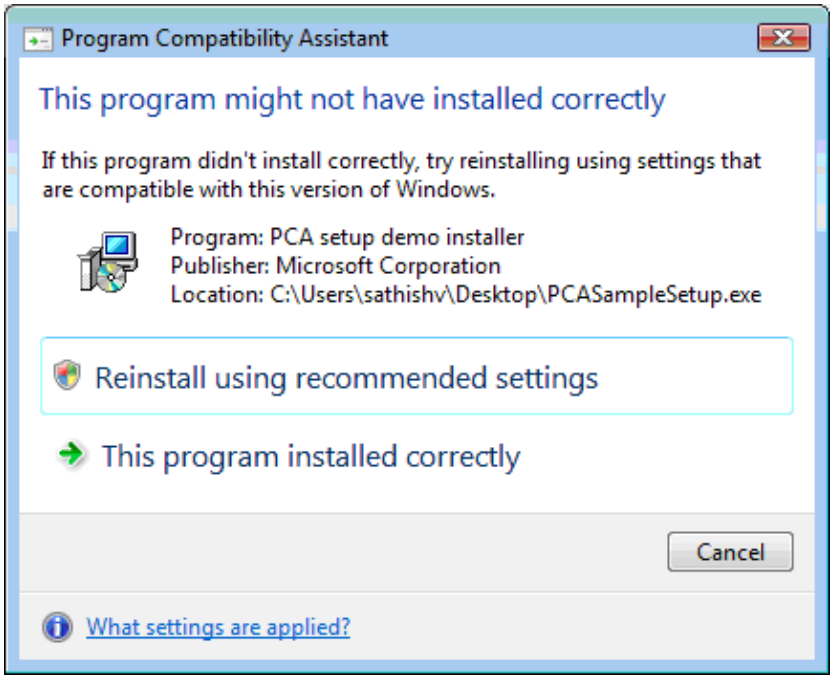


Figure 2.4: The Program Compatibility Assistant Querying the End User After an Installation of a Potentially Incompatible Software Product According to [Cor08a].

The Program Compatibility Assistant displays a dialog like you can see in Figure 2.4, where an end user needs to verify, whether the software Component has been installed properly. If it has not been installed properly, the end user can adjust the compatibility options of the operating system in order to make it behave according to a previous version like Windows XP or Windows 98 to enable the integration of this new software Component.

2.2.3 Standardization

Another approach enabling Dependability of Dynamic Adaptive Systems is standardization. In this case, vendors commit on common standards regarding interfaces of Dependable Dynamic Adaptive Components. These standards define syntactical and semantical properties of interfaces for new Components. During integration of new Components into a system using the standardization approach, it is assumed, that these Components implement these standards correctly. However, a correct implementation of the standard by a new Component is not guaranteed. Therefore, negative side-effects of the integration due to an incorrect implementation of the standard cannot be ruled out by using only the standardization approach.

Standards help developers to get a common understanding. Correct implementation of a standard is not verified in advance.

There are several examples for standardization as an approach to dynamic

integration of Unknown Service Partners. One of them is the Bluetooth standard [SIG07] which has been established as a standard for wireless connections between devices. Different services, that may be used to connect Components are specified as so called *profiles*. One example of such a profile is the Hands-free profile [SIG05], which specifies how a car stereo (or a similar hands-free set) needs to behave in order to enable a Bluetooth cell phone to use it as a hands-free set. It is not checked, whether a device, which claims to support a Bluetooth profile, has a correct implementation of this profile during integration.

Another example is the Universal Plug&Play (UPnP) standard [For06], which has been established by Microsoft. This standard is targeted mainly at Components from the home-entertainment domain. One of the features is the so called *zero-configuration* enabling Components to connect to each other just by wiring them together without any additional administration effort. The Components may query the capabilities of other Components based on service descriptions in XML. By this mechanism they can locate Components, which they can use for interaction by SOAP [W3C07] messages. Similar as for the Bluetooth standard, no testing of a correct service implementation is contained in the UPnP standard.

We need: Dependable Integration without side-effects on a running Dependable Dynamic Adaptive System.

Summed up, at the moment there is no approach enabling us to integrate Unknown Service Partners into a system at runtime while guaranteeing that it has no negative side-effects on the functionality of the system. A survey about Component interoperability [VHT00] concluded, that "all previous approaches either only partially capture the semantics of the objects they describe, or they make some assumptions which restrict the general applicability of their solutions". Anyhow such an approach is needed to establish Dependable Dynamic Adaptive Systems. Thus, we focus on providing such an approach within this thesis.

Grown-ups never understand anything by themselves, and it is tiresome for children to be always and forever explaining things to them.

Antoine de Saint Exupéry

3 Application Example

This chapter describes our application example. It is a small extract of a system assisting medical forces during a huge disaster. Our prototypical implementation of the application example has been exhibited at CeBIT 2009 [Mes09].

This example has been cut down in order to motivate the contribution of this thesis without distracting readers with unnecessary application details. If you are interested in the application example, you can find a more detailed description at the exhibit's webpage [NSH09] or in a German talk given at CeBIT Future Talk 2009 [Nie09] respectively in a German video describing this system [Nie10].

The objective of our application example is to enhance the readability of this thesis by referring to a vivid example.

The application example is used throughout the whole thesis to describe our approach to Dependable Dynamic Adaptive Systems more vividly. Nevertheless our approach is *not* limited to building systems for Emergency Assistance Systems. Instead you can use the approach, whenever you want to build a Dependable Dynamic Adaptive System respectively whenever you want to build a Dependable Dynamic Adaptive Component for such a system.

In the following we will describe the domain of Emergency Assistance Systems.

3.1 Domain Description

Imagine a huge disaster like the one, which occurred during an airshow in Ramstein in 1988 [Dum09]. 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.

In huge disasters, medics classify all casualties by Triage Classes, before they start treating casualties.

Since you will usually face much more casualties than medics in a disaster scenario, medics cannot start treating the casualties directly when they arrive. Instead they need to figure out, which casualties need to be treated first due to the severity of their injuries. A simplified overview of the overall rescue process during a huge disaster is depicted in Figure 3.1.

As you can see in this figure, the medics start with triage classification *before* they treat casualties. During this classification process they assign a so-called *Triage Class* [Gro05] to each casualty specifying the severity of his injury. This Triage Class is used as treatment priority during the following rescue operation.

In Germany and further countries, casualties are classified in five Triage Classes: *T I* to *T IV* as well as *EX*.

T I: Triage Class *T I* characterizes casualties, which are severely wounded and need to be treated immediately.

T II: A casualty classified as *T II* needs to be evacuated to a hospital for further treatment.

T III: Casualties classified as *T III* are only slightly injured and are treated only rudimentary on-site.

T IV: Triage Class *T IV* characterizes casualties, which have no chance to survive and are only treated in terms of terminal care.

EX: An additional Triage Class *EX* denotes casualties which have died.

3.2 Emergency Assistance System

To provide an IT system supporting medical forces in disaster situations, we think of three starting points in the context of this thesis:

1. Supporting the triage classification process by enabling automatic classification of casualties.

2. Supporting medical treatment by providing live information about vital data as well as about the treatment history.

3. Supporting an incident command by providing an overview of the disaster situation.

In the following we will sketch, how an IT system can provide support for these aspects.

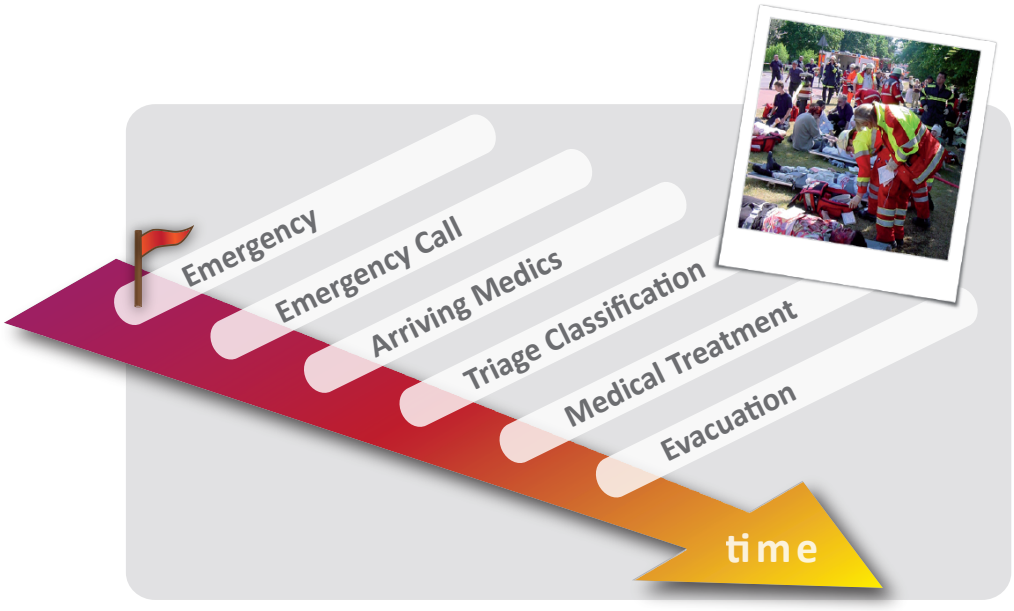


Figure 3.1: A sequence of events in case of a huge disaster.

3.2.1 Support of Triage Classification Process

The triage classification process can be supported by an IT system by capturing the Triage Class of each casualty within the system. In today's practice, medics in Germany use a casualty card – a card with a neck strap – to classify casualties and to capture treatment relevant information like administered drugs. Attached to this card is a small leaflet representing the Triage Class by a color code as depicted in Figure 3.2.

Casualty cards can be replaced by their digital equivalent: C-Units.

If a medic wants to assign a Triage Class to a casualty, he needs to fold this leaflet in a way, that it shows the color representing the casualty's Triage Class on top. In the following a medic needs to attach this folded leaflet to the casualty card. The unfolded leaflet is depicted in Figure 3.3. As you can imagine, capturing the Triage Class using a casualty card is a time-consuming, error-prone process. An IT system could support this process by providing an easy-to-use user interface, where a medic can easily assign a Triage Class to a casualty.

Folding casualty cards is time-consuming and error-prone.

To provide this, our Emergency Assistance System contains two types of Components: *Medic Units* (in the following abbreviated as M-Unit) and *Casualty Units* (abbreviated as C-Unit). A M-Unit is the interaction device of a medic in our application example. A medic can use a M-Unit to assign Triage Classes to casualties, view their vital data or treatment history. You can think of a M-Unit as a smartphone or netbook application. A M-Unit prototypically realized by us is depicted

M-Units are interaction devices used by medics to assign Triage Classes or to view vital data information of casualties.

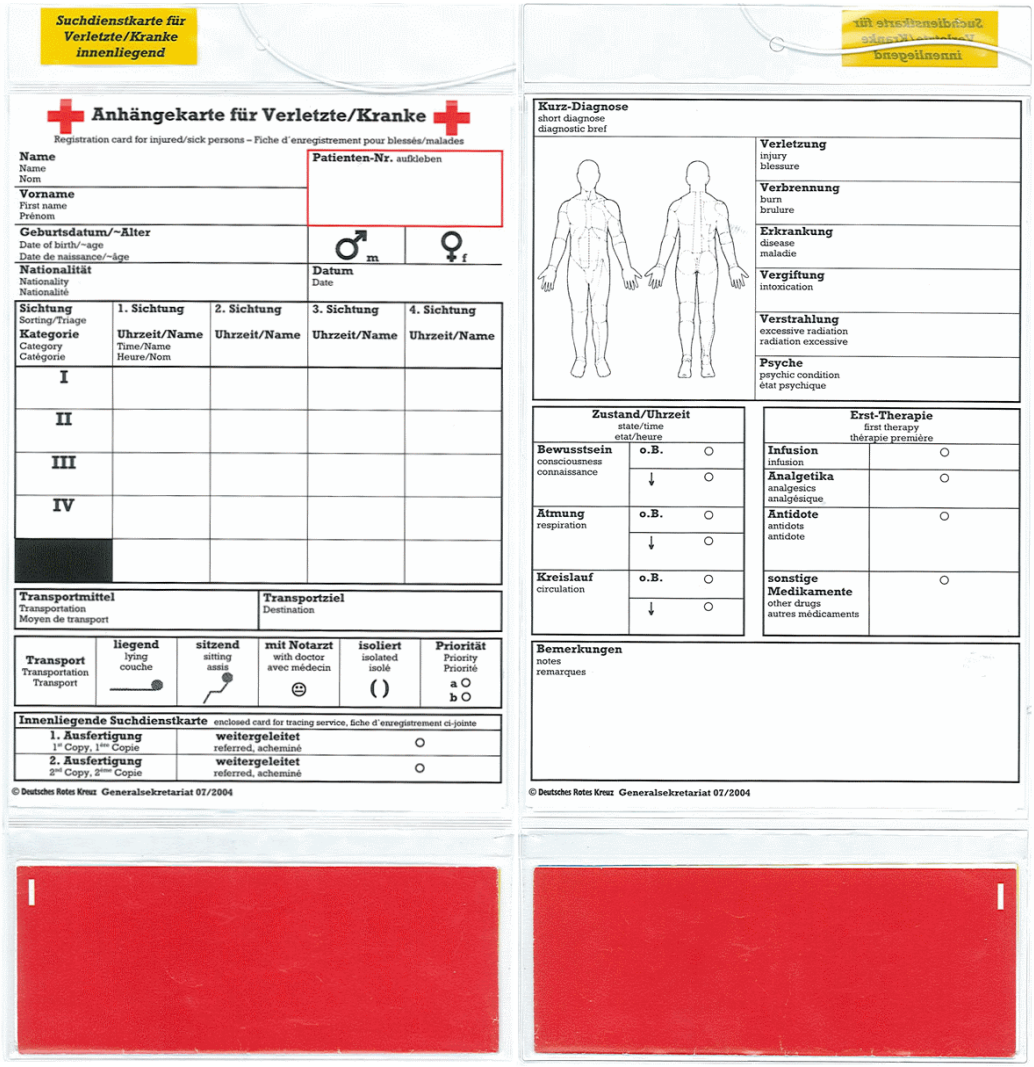


Figure 3.2: A German Casualty Card's Front and Rear View with a Triage Class Leaflet Attached at the Bottom.

in Figure 3.4.

C-Units simply store data of a casualty like his Triage Class, gender, a treatment history, or all other information shown in Figure 3.2. They may be small microcontrollers like Crossbow's MICAz Motes [Tec09] or Sun SPOTs [SHDC06]. C-Units are deployed by medics at casualties – one C-Unit for each casualty – and remain there during the rescue operation. For our prototypical implementation of the Emergency Assistance System's C-Units, we chose Sun SPOTs and cased them with a neck strap for easy application as depicted in Figure 3.5.

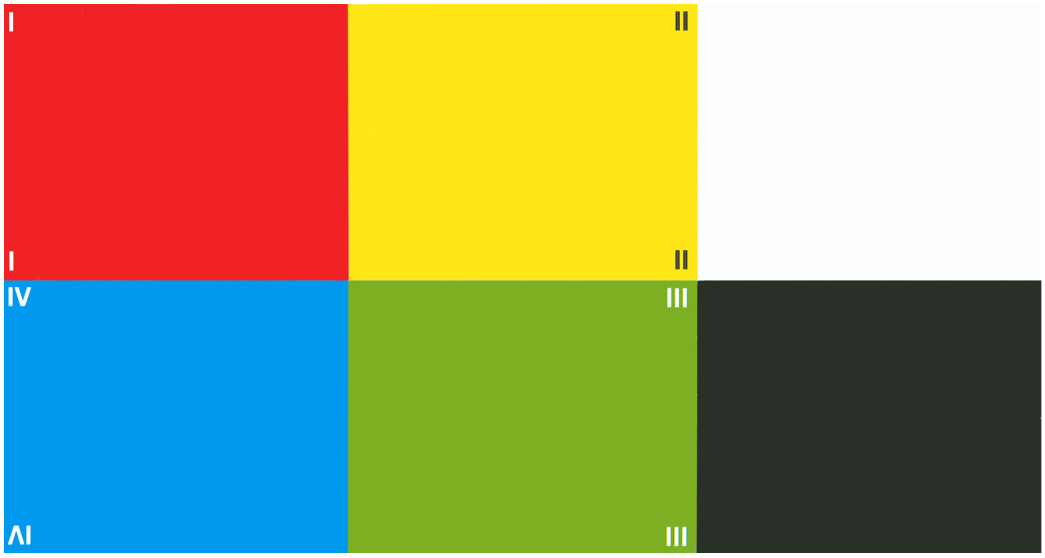


Figure 3.3: An Unfolded Triage Class Leaflet – Triage Class EX is Represented by the Black Color.

If C-Units only store information, low-cost Components like RFIDs would be sufficient. However, the use of microcontrollers enables us to enhance their capabilities. Our C-Units are, therefore, not only capable of storing information. In addition, medics, which are not equipped with M-Units can toggle the Triage Class of a casualty by simply pushing a button at the casualty's C-Unit. Therefore, our C-Units provide a simple visualization of the current Triage Class by color LEDs. In addition C-Units also may provide positioning information about the casualty by connecting a GPS receiver.

Next to this, our C-Units are capable of assisting medics during the triage classification process even more: If an optional *Peripheral Unit* (abbreviated as P-Unit in the following) is attached to a casualty and, therefore, enters the Dependable Dynamic Adaptive System, the C-Unit can *calculate* the Triage Class of a casualty based on his vital state. It has been shown in a clinical context in [PBC+06] that such an electronical support during triage classification is meaningful.

P-Units measure vital data of casualties – using a P-Unit, a C-Unit can automatically calculate a Triage Class.

A P-Unit features vital data sensors like a pulse rate sensor or a blood pressure sensor. If a medic attaches these sensors to a casualty, vital data of the casualty is transmitted to the corresponding C-Unit and can be monitored by a medic. You can see our prototypical implementation of a P-Unit in Figure 3.6. It features a pulse rate sensor and a blood pressure sensor. The pulse rate sensor can measure aeration as well.

In our prototypical implementation the sensor values are collected by a Sun SPOT as well, which is hidden inside the bag depicted in Figure 3.6. This Sun SPOT

Figure 3.4: A M-Unit for the Emergency Assistance System.

acts as a proxy to the sensors and, therefore, represents the P-Unit in our example. It transmits the sensor values wirelessly to the corresponding associated C-Unit.

3.2.2 Support of Medical Treatment

In order to support medics during medical treatment, the Emergency Assistance System's M-Units are able to provide medics with the treatment history of each casualty which is queried at the corresponding C-Unit. The history contains, which drugs have been administered yet to prevent overdoses as well as reciprocal effects. Medics can capture performed treatment steps easily by pushing a button on their M-Unit.

Moreover the history of a casualty's vital data can be displayed on the M-Unit. M-Units assist medics by recommending further treatment steps considering a casualty's vital state and treatment history.

Figure 3.5: A Cased Sun SPOT Acting as a C-Unit in our Emergency Assistance System.

3.2.3 Support of Incident Command

To support the incident command in a rescue operation, we provide an Incident Command Unit (abbreviated as IC-Unit) in our application example. The IC-Unit may be a large display mounted to the inside of an ambulance vehicle as depicted in Figure 3.7. It enables an incident command, which coordinates the rescue operation, to assign medics to casualties efficiently. Therefore, the IC-Unit displays a general map, where all medics and casualties are represented as icons. It displays current activities of all medics (queried from their M-Units) as well as vital data and Triage Classes of all casualties (queried at the corresponding C-Units respectively P-Units) on site.

The incident command assigns medics to casualties using the IC-Unit.

Our prototypical IC-Unit implementation is capable of touch-based interaction allowing intuitive operation by the incident command. By integrating C-Units and P-Units into the Emergency Assistance System as described before, the incident command can recognize critical state changes of specific casualties even if no medic is currently present at their place. This is a very important feature in a disaster situation, as disasters are characterized by numbers of casualties which exceed the numbers of medics by far.

Figure 3.6: A Cased Sun SPOT With Vital Data Sensors Attached. It is Acting as a P-Unit in our Emergency Assistance System.

Figure 3.7: A Mounted Wall Display Acting as an IC-Unit in our Emergency Assistance System.

Figure 3.8: Overview of the Different Components Within our Application Example.

Since the IC-Unit provides an overview of all medics, their location and current activities, it enables the incident command to assign medics to casualties very efficiently *on demand*. The orders given to medics are transmitted to the corresponding M-Units and displayed there, supplementing voice radio.

The Components involved in the overall scenario are depicted in Figure 3.8.

Our application example imposes threats to Dependability that need to be faced in order to fully benefit from the different Components. These problems will be discussed in the following by considering an excerpt from the application example. We do not consider the IC-Unit during this discussion, since it does not introduce additional Dependability threats, which are not already covered by considering M-Units, C-Units, and P-Units.

3.3 Dependability Threats Derived from the Application Example

Components of different vendors cause incompatibilities.

First of all, Emergency Assistance Systems have to bind several Dynamic Adaptive Components like M-Units, C-Units, or P-Units at runtime. Medics may come along with different versions of M-Units, C-Units, and P-Units, which are not compatible with each other. For example, some medics may bring along C-Units, which cause

problems, if they are connected to P-Units introduced by another medic. Therefore, a mechanism needs to be available, which is capable of deciding, which Dynamic Adaptive Components can be bound to each other.

Matching syntax is not sufficient – semantics need to be considered as well.

Since it is very likely, that Dynamic Adaptive Components appearing in this scenario have been developed independently from each other, their vendors need to agree on a common Domain Architecture for the Emergency Assistance domain. This Domain Architecture contains Service Interfaces, which may be required respectively provided by Components they implement. However, we can not assume that they are semantical compatible whenever their required respectively provided Service Interfaces are syntactically compatible. Dynamic Adaptive Components requiring a Service Interface may assume a different (semantical) behavior than it is provided by another Dynamic Adaptive Component although they refer to *syntactically* compatible Service Interfaces. Therefore, an interoperability test has to be performed, before two Dynamic Adaptive Components are bound.

Our example contains Components provided by a Dutch and a German vendor.

In order to present our approach to Dependable Dynamic Adaptive Systems capable of Dependable Integration we consider the following chain of events within our example system:

1. A large catastrophe occurs in Aachen (Germany) which is located near the border to the Netherlands. Emergency forces of these two countries are notified.

2. A German ambulance arrives at the catastrophe site first. The emergency staff is equipped with M-Units. They move about and equip wounded casualties with German C-Units and P-Units.

   German P-Units = two physical sensors.

   German P-Units consist of two physical sensors: a fingerclip measuring the pulse rate and a wrist cuff measuring the blood pressure. The M-Units of bypassing medics connect to C-Units. A C-Unit may be connected to a P-Unit for automatic Triage Class calculation. The resulting system configuration is depicted in Figure 3.9 (1).

3. Now a Dutch ambulance arrives at the site. The Dutch medical forces come along with P-Units, which differ physically from the German ones as they consist of a single physical sensor measuring pulse rate and blood pressure at once. In addition Dutch medical forces introduce their own C-Units and M-Units. Dutch M-Units, C-Units, and P-Units are designed to operate with each other, while the interoperability between Dutch and German Dynamic Adaptive Components has not been tested during development, since they have been developed independently by different vendors.

   Dutch P-Units = a single physical sensor.

Specific about Dutch C-Units is, that they behave differently than German C-Units in some cases. Dutch C-Units assume during automatic Triage Class calculation,

that the pulse rate measured by a P-Unit may only be equal to zero, if the blood pressure is equal to zero at the same time. This assumption has been derived from the specific Dutch P-Unit: blood pressure and pulse rate are measured by the same physical sensor – a wrist cuff. Thus, for Dutch P-Units this assumption holds. Due to this assumption, the Dutch C-Units classify a casualty as dead (Triage Class EX), whenever the pulse rate is equal to zero without considering the blood pressure in addition.

Dutch C-Units behave different than German C-Units.

Consider a situation, where a casualty is equipped with a Dutch C-Unit and a German P-Unit. This may work without any problems most of the time. However, we may face problems if the fingerclip of this casualty slips off. In this moment the Dutch C-Unit will recognize a pulse rate equal to zero and, therefore, classify the casualty as dead because it does not take the blood pressure value into account. In a disaster situation with a huge amount of casualties this would cause the effect, that no medic would be sent to this casualty as casualties with better survival chances need to be treated first.

Binding a Dutch C-Unit to a German P-Unit may lead to semantical incompatibilities.

Therefore, Dutch C-Units are semantically incompatible with German P-Units in this situation. The incompatibility has an effect, whenever the pulse rate measured by a P-Unit is equal to zero while the blood pressure measured by the P-Unit is not equal to zero or vice versa – this corresponds to situations where a single sensor slips off. Dutch C-Units must not rely on the sensor values of the German C-Units in this case. They need to detect this incompatibility and need to fall back to a configuration allowing medics to manually set the Triage Class of the casualty. The overall possible Component Bindings in our example are depicted in Figure 3.9 (2).

3.4 A Software View on the Application Example

Looking at our example from a software point of view, each hardware unit introduced before (IC-Unit, M-Unit, C-Unit, and P-Unit) is hosting a corresponding software Component – a Dynamic Adaptive Component. These Dynamic Adaptive Component require respectively provide different Service Interfaces. These Service Interfaces are standardized in a Domain Architecture for Emergency Assistance Systems, which is introduced in the following.

3.4.1 Domain Architecture for Emergency Assistance Systems

Dynamic Adaptive Systems, like the Emergency Assistance System in our application example, are based on a standardized Domain Architecture containing Service Interface specifications. The standardized Service Interface specification of a Domain Architecture allows different vendors to implement Dynamic Adaptive Com-

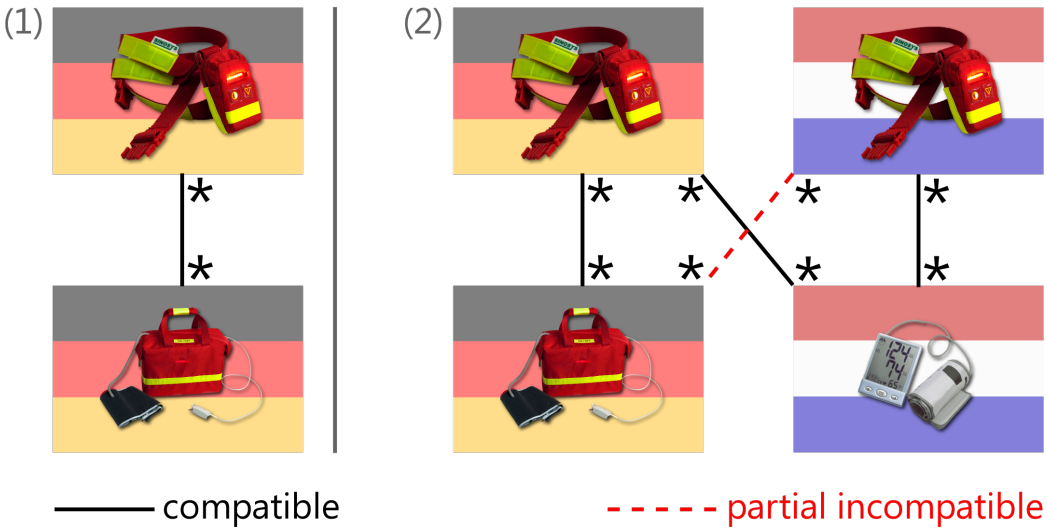


Figure 3.9: The Component Bindings Within our Application Example and Their Compatibility.

ponents for Dynamic Adaptive Systems of this specific domain. They can do this by developing Dynamic Adaptive Components, which require respectively provide these specified Service Interfaces. In a specific Dynamic Adaptive System these Components can be bound at runtime to Dynamic Adaptive Component built by third-parties.

Dependable Dynamic Adaptive Components of our example are bound using Service Interfaces from a Domain Architecture.

For our application example the Domain Architecture for Emergency Assistance Systems contains Service Interface `PeripheralUnitIf` extending the Service Interfaces `BloodPressureSensorIf` and `PulseRateSensorIf` for the P-Units as depicted in Figure 3.10. This Service Interface provides access to the vital data of a casualty.

C-Units may use this Service Interface in order to calculate the Triage Class of a casualty and to provide their Service Interface `CasualtyUnitIf`. M-Units provide Service Interface `MedicUnitIf` where the IC-Unit can query, where a medic is located and what he is currently doing.

The syntactical specification of these Service Interfaces is sketched in the upper part of Figure 3.10. In the lower left part of the figure you can see the specification of datatypes occurring in these Service Interfaces, while the lower right part contains a specification of typical Dynamic Adaptive Components prominent in this domain like P-Units.

A Component specification is not mandatory in Dynamic Adaptive Systems, since we assume, that an infrastructure is available in these systems. This infrastructure needs to be capable of binding Dynamic Adaptive Components at runtime based on

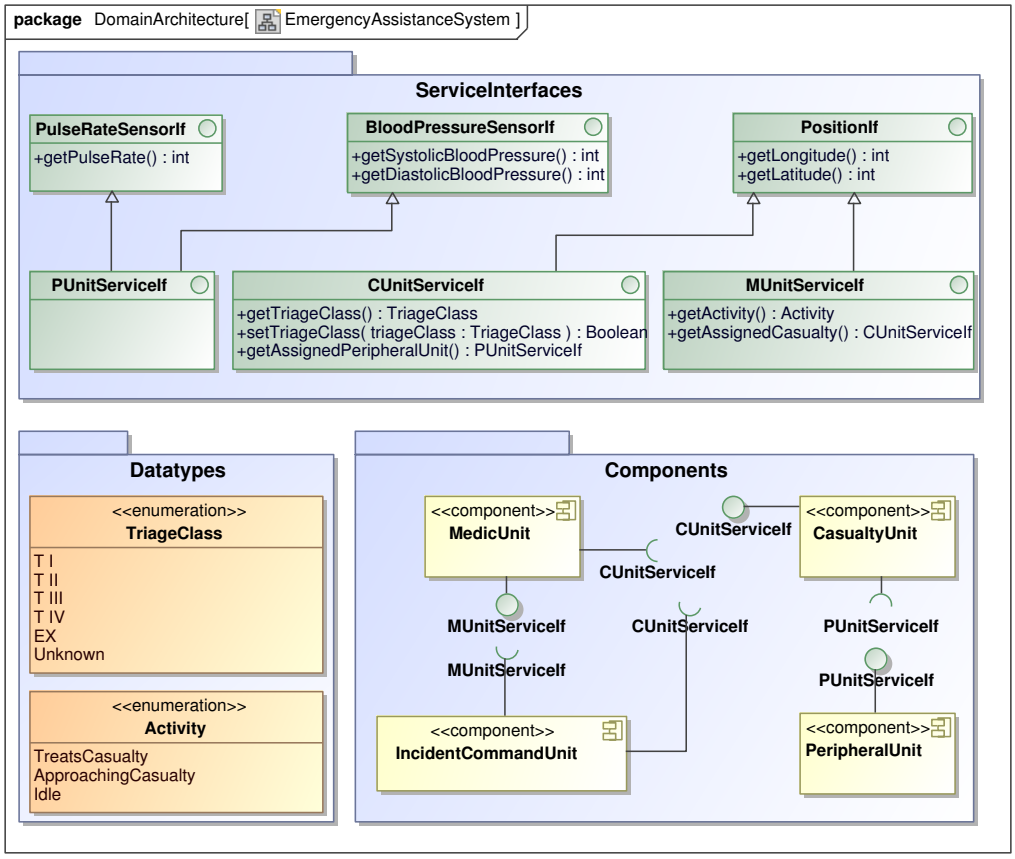


Figure 3.10: A Domain Architecture for Emergency Assistance Systems.

provided and required Service Interfaces. However, a Component specification can help developers to understand the domain and how Dynamic Adaptive Components were intended to look like during standardization.

Since we want to build a Dependable Dynamic Adaptive System from Dependable Dynamic Adaptive Components developed by different vendors, a Domain Architecture does not only contain syntactical information like method signatures or datatypes occurring in the Service Interfaces provided or required by Dependable Dynamic Adaptive Components. It also may contain a semantical specification following the Design by Contract [Mey92] approach. To specify pre- and post-conditions and invariants in our example we used the Java Modeling Language (JML) [LBR06], since it is a very mature specification technique.

The specifications for the Service Interfaces provided by the P-Unit in our application example are straightforward. Service Interface `PulseRateSensorIf`

specifies a single method `getPulseRate()`, which must not return[1] a negative value, indicated by a postcondition depicted in line 4 of Listing 3.1.

```
1 package domainModel.pulseRateSensor;
2
3 public interface PulseRateSensorIf {
4   /*@ ensures (\result >=0); @*/
5   public /*@ pure @*/ int getPulseRate();
6 }
```

Listing 3.1: Semantical Specification of Service Interface PulseRateSensorIf.

The same postcondition is specified for the two methods `getSystolicBloodPressure()` and `getDiastolicBloodPressure()` (lines 7, 10 in Listing 3.2) contained in Service Interface `BloodPressureSensorIf`. Moreover an invariant may state medicine's knowledge, that the systolic blood pressure must be greater than or equal to the diastolic blood pressure (lines 4–5 in Listing 3.2).

```
1 package domainModel.bloodPressureSensor;
2
3 public interface BloodPressureSensorIf {
4   /*@ public invariant getSystolicBloodPressure()
5   @ >= getDiastolicBloodPressure(); @*/
6
7   /*@ ensures (\result >=0) @*/
8   public /*@ pure @*/ int getSystolicBloodPressure();
9
10  /*@ ensures (\result >=0) @*/
11  public /*@ pure @*/ int getDiastolicBloodPressure();
12 }
```

Listing 3.2: Semantical Specification of Service Interface BloodPressureSensorIf.

More complex is the specification of Service Interface `CasualtyUnitIf`. This interface contains the specification of the `getTriageClass()` method, which will be used subsequently to illustrate the need for Dependable Dynamic Adaptive Systems. Method `getTriageClass()` returns the Triage Class of a casualty based on the pulse rate and blood pressure[2] if a P-Unit is connected. If no P-Unit is present, it returns the Triage Class which has been manually set by a medic before.

Especially important about this specification is, that `TriageClass.EX` should only be returned, whenever the pulse rate as well as the systolic blood pressure is

[1] *\result* is a JML notation referring to the return value of the method for which a postcondition is specified.

[2] Note that the values for the vital conditions occurring in the specification are rather arbitrary to give you a clue, so don't expect them to be meaningful in medicine's sense.

zero, which is specified in lines 13–14 in Listing 3.3. If only one of these sensor values is zero, this may mean, that this sensor might have slipped off. Consequently `TriageClass.Unknown` should be returned. This is specified in lines 16–20 in Listing 3.3.

A casualty *must not* be classified as `TriageClass.EX`, if pulse rate or blood pressure are above zero.

```
1 package domainModel.casualtyUnit;
2
3 import domainModel.peripheralUnit.PeripheralUnitIf;
4 import domainModel.casualtyData.TriageClass;
5
6 public interface CasualtyUnitIf {
7   public /*@ pure @*/ PeripheralUnitIf
       getAssignedPeripheralUnit();
8   // ... further methods operating with personal data of the
       casualty
9
10  /*@
11  @ requires peripheralUnit!=null;
12  @ {|
13  @ requires peripheralUnit.getPulseRate()==0 &&
       peripheralUnit.getSystolicBloodPressure()==0;
14  @ ensures \result == TriageClass.EX;
15  @ also
16  @ requires peripheralUnit.getPulseRate()==0 &&
       peripheralUnit.getSystolicBloodPressure()!=0;
17  @ ensures \result == TriageClass.Unknown;
18  @ also
19  @ requires peripheralUnit.getPulseRate()!=0 &&
       peripheralUnit.getSystolicBloodPressure()==0;
20  @ ensures \result == TriageClass.Unknown;
21  @ also
22  @ {|
23  @   requires peripheralUnit.getPulseRate()>0;
24  @   requires peripheralUnit.getPulseRate()<=30;
25  @   ensures \result == TriageClass.T_IV;
26  @ also
27  @   [...]
28  @ |}
29  @ |}
30  @ also
31  @ [...]
32  @*/
33  public /*@ pure @*/ TriageClass getTriageClass();
34
35  /*@ ensures setTriageClass == triageClass; @*/
36  public void setTriageClass(TriageClass triageClass);
```

```
37 }
```

Listing 3.3: Semantical Specification of Service Interface CasualtyUnitIf in the Domain Architecture.

Now that we have a Domain Architecture, vendors can implement Dynamic Adaptive Components for Emergency Assistance Systems. In the following we will have a look at Dynamic Adaptive Components built by different vendors represented by different countries from our example sketched before in Section 3.3[3]. We will take a close look at the implementation of the `getTriageClass()` method as it is used to show, why some of these Dynamic Adaptive Components are not compatible with each other.

3.4.2 Dynamic Adaptive Components Provided by a German Vendor

A German vendor has developed two Components for German medical forces that may remain at each casualty: a German C-Unit and a German P-Unit. We will describe the implementation of the Service Interfaces from the Domain Architecture described in Section 3.4.1 provided by these German Dynamic Adaptive Components in the following.

German C-Unit

The German C-Unit implements Service Interface `CasualtyUnitIf`, to enable M-Units and the IC-Unit to display the Triage Class of its casualty.

While analyzing required Service Interfaces the German vendor had two different Component Configurations in mind:

1. Component Configuration 1, where no Service Interface is required. This represents a casualty, where the medic manually sets the Triage Class – no vital data sensors are present.

2. Component Configuration 2, where Service Interface `PeripheralUnitIf` is required. This represents a casualty, which is currently equipped with vital data sensors monitoring his physical condition. Therefore, his Triage Class can be calculated automatically.

The German C-Unit is a straightforward implementation of the Domain Architecture.

The implementation of the German C-Unit is directly derived from the specification. The German vendor implemented the `getTriageClass()` method exactly the way, as it had been specified in the domain architecture, considering the two

[3]The implementations of the M-Unit as well as the IC-Unit are not considered in the following, as they are not required to motivate Dependable Dynamic Adaptive Systems.

relevant cases: If both sensor values are equal to zero `TriageClass.EX` is returned (cf. lines 32 – 34 of Listing 3.4) whereas `TriageClass.Unknown` is returned if only a single sensor value is zero (cf. lines 35 – 37 of Listing 3.4).

```
 1 package germanVendor.casualtyUnit;
 2
 3 import domainModel.peripheralUnit.PeripheralUnitIf;
 4 import domainModel.casualtyData.TriageClass;
 5 import germanVendor.ConfigurableCasualtyUnitIf;
 6
 7 public class CasualtyUnitImpl implements
       ConfigurableCasualtyUnitIf{
 8   private /*@ spec_public nullable @*/ PeripheralUnitIf
        myPeripheralUnit = null;
 9   private TriageClass myTriageClass = TriageClass.Unknown;
10
11   public CasualtyUnitImpl(){
12   }
13
14   /*@
15   @ assignable myPeripheralUnit;
16   @ ensures (myPeripheralUnit == peripheralUnit);
17   @*/
18   public CasualtyUnitImpl(PeripheralUnitIf peripheralUnit){
19     myPeripheralUnit = peripheralUnit;
20   }
21
22   public PeripheralUnitIf getAssignedPeripheralUnit() {
23     return myPeripheralUnit;
24   }
25
26   public void setAssignedPeripheralUnit(PeripheralUnitIf
        peripheralUnit) {
27     myPeripheralUnit = peripheralUnit;
28   }
29
30   public TriageClass getTriageClass() {
31     if (myPeripheralUnit!=null){
32       if ((myPeripheralUnit.getSystolicBloodPressure()==0) && (
            myPeripheralUnit.getPulseRate()==0)){
33         return TriageClass.EX;
34       }
35       if (((myPeripheralUnit.getSystolicBloodPressure()==0) &&
            (myPeripheralUnit.getPulseRate()>0)) || ((
            myPeripheralUnit.getSystolicBloodPressure()>0) && (
            myPeripheralUnit.getPulseRate()==0))){
36         return TriageClass.Unknown;
37       }
38       [...]
```

Figure 3.11: The German P-Unit – Two Separate Sensors.

```
39      }
40      [...]
41    }
42
43    public void setTriageClass(TriageClass triageClass) {
44      /*@ set setVitalCondition = vitalCondition; @*/
45      myTriageClass = triageClass;
46    }
47  }
```

Listing 3.4: Implementation of the German C-Unit.

German P-Unit

The German P-Unit implements Service Interface `PeripheralUnitIf`, to enable C-Units, M-Units, and the IC-Unit to query vital data of a casualty. The German P-Unit consists of two separate physical sensors – a fingerclip measuring pulse rate and aeration of the casualty and a wrist cuff measuring the blood pressure. Thus, the pulse rate of a German P-Unit can be equal to zero while the blood pressure is still above zero. This is the case if the fingerclip slips off, for example. An example of such a German P-Unit is depicted in Figure 3.11.

3.4.3 Dynamic Adaptive Components Provided by a Dutch Vendor

A Dutch vendor has developed two Components for Dutch medical forces that may remain at each casualty: a Dutch C-Unit and a Dutch P-Unit. We will describe the implementation of the Service Interfaces from the Domain Architecture described in Section 3.4.1 provided by the Dutch Dynamic Adaptive Components in the following.

Dutch C-Units

The Dutch vendor has implemented C-Units which calculate the Triage Class slightly different than specified in the Domain Architecture: as the Dutch vendor had a P-Unit in mind, which has only a single combined physical sensor which measures blood pressure, pulse rate, and aeration. Therefore, the case, that only a single value is equal to zero is irrelevant, as it may not happen.

Dutch C-Units do not consider blood pressure to decide, whether a casualty is dead.

Thus, Dutch C-Units only consider the pulse rate, when deciding whether a person is dead or not, as depicted in Listing 3.5 in lines 28–30. The Dutch vendor does not consider this as a violation of the Domain Architecture, as the implementation of the Dutch P-Unit guarantees, a single sensor value will never be zero. However, it is an incorrect implementation of the Domain Architecture, as Dutch C-Units may violate the Service Interface specification, when they are bound to P-Units provided by other vendors[4].

```
1 package dutchVendor.casualtyUnit;
2
3 import domainModel.peripheralUnit.PeripheralUnitIf;
4 import domainModel.casualtyData.TriageClass;
5
6 public class CasualtyUnitImpl implements CasualtyUnitIf {
7   private /*@ spec_public nullable @*/ PeripheralUnitIf
        myPeripheralUnit = null;
8   private TriageClass myTriageClass = TriageClass.Unknown;
9
10  public CasualtyUnitImpl(){
11  }
12
13  /*@ ensures (myPeripheralUnit == peripheralUnit); @*/
14  public CasualtyUnitImpl(PeripheralUnitIf peripheralUnit){
15    myPeripheralUnit = peripheralUnit;
16  }
17
```

[4] Such a situation could occur even *without an incorrect implementation of the Domain Architecture*. In our example this would directly be the case, if the Domain Architecture does not specify the semantics of the `getTriageClass()` method. Domain Architectures are typically underspecified, since vendors want to provide Components with unique selling points.

```
18  public PeripheralUnitIf getAssignedPeripheralUnit() {
19    return myPeripheralUnit;
20  }
21
22  public void setAssignedPeripheralUnit(PeripheralUnitIf
       peripheralUnit){
23    myPeripheralUnit = peripheralUnit;
24  }
25
26  public TriageClass getTriageClass() {
27    if (myPeripheralUnit!=null){
28      if (myPeripheralUnit.getPulseRate()==0){
29        return TriageClass.EX;
30      }
31      [...]
32    }
33    [...]
34  }
35
36  public void setTriageClass(TriageClass triageClass) {
37    /*@ set setVitalCondition = vitalCondition; @*/
38    myTriageClass = triageClass;
39  }
40 }
```

Listing 3.5: Implementation of the Dutch C-Units.

Dutch P-Unit

Dutch P-Units guarantee, that a pulse rate of zero implies a blood pressure of zero.

The Dutch P-Unit is a combined sensor which may be built as a wrist cuff and, therefore, cannot slip off during usage – especially pulse rate and blood pressure can only be equal to zero at the same time[5]. An example of such a Dutch P-Unit is depicted in Figure 3.12.

Consequently the Dutch vendor adds an invariant to describe this additional property of the combined sensor as depicted in the invariant in lines 10–11 of Listing 3.6.

```
1 package dutchVendor.peripheralUnit;
2
3 import domainModel.PeripheralUnitIf;
4
5 public class PeripheralUnitImpl implements PeripheralUnitIf {
6   /**
7    * NOTE: Physical pulse sensor is a wrist cuff, which cannot
         slip off.
8    */
```

[5] We assume, that the sensor unit is not broken.

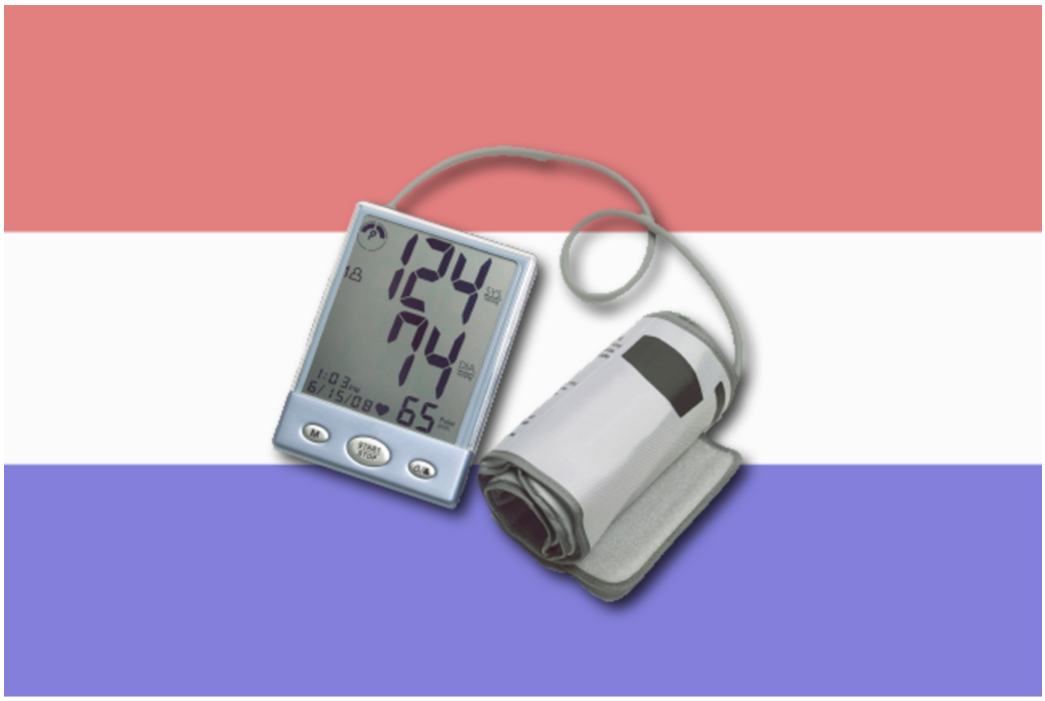


Figure 3.12: The Dutch P-Unit – a Combined Sensor.

```
 9
10  /*@ public invariant getPulseRate()==0 <==>
       getDiastolicBloodPressure()==0; @*/
11  /*@ public invariant getDiastolicBloodPressure()==0 <==>
       getSystolicBloodPressure()==0; @*/
12
13  /**
14   * Calculates current pulse rate based on physical pulse
        sensor data. Left out here intentionally.
15   */
16  public int getPulseRate() {
17    [...]
18  }
19
20  /**
21   * Calculates current diastolic blood pressure based on
        physical blood pressure sensor data. Left out here
        intentionally.
22   */
23  public int getDiastolicBloodPressure() {
24    [...]
25  }
26
```

```
27  /**
28   * Calculates current systolic blood pressure based on
        physical blood pressure sensor data. Left out here
        intentionally.
29   */
30  public int getSystolicBloodPressure() {
31    [...]
32  }
33 }
```

Listing 3.6: Implementation of the Dutch P-Unit.

In the following we will analyze, whether C-Units and P-Units provided by these two vendors in our example are compatible with each other.

3.4.4 Compatibility of Dynamic Adaptive Components in our Example

When looking at the C-Units and P-Units from our example, C-Units act as Service Providers for the M-Units as C-Units provide the Service Interface `CasualtyUnitIf` required by the M-Units. C-Units act as Service Users requiring the Service Interface `PeripheralUnitIf`, which is provided by P-Units which act as Service Providers. Summed up, in our example M-Units use C-Units while C-Units use P-Units. In the following we will have a look at the Compatibility between the C-Units and P-Units provided by the two different vendors in our application example.

Compatibility can be split into *syntactical* and *semantical* Compatibility.

The Compatibility of a Dynamic Adaptive Component acting as Service Provider and a Dynamic Adaptive Component acting as Service User can be seen at two levels:

1. *Syntactical Compatibility*: This means, that all methods and attributes required by a Service Reference of a Service User are syntactically compatible with those provided by a Service of a Service Provider.

2. *Semantical Compatibility*: A Service Provider and a Service User are semantically compatible, when the Service provided by the Service Provider behaves as expected by the Service User.

M-Units are syntactically compatible with C-Units; C-Units are syntactically compatible with P-Units.

Syntactical Compatibility of Dynamic Adaptive Components in our Example

For syntactical Compatibility between C-Units and P-Units within our example, we can derive, that both C-Units are syntactically compatible with each P-Unit, as the P-Units all provide Service Interface `PeripheralUnitIf`, which is required by the C-Units. In this case, this is trivial, as Service User and Service Provider refer

to the same Service Interface from the Domain Architecture. However, syntactical Compatibility becomes more complex, if we consider inheritance relations of Service Interfaces as well. This is done in our system model in Chapter 4 by the relation $\simeq_{\text{Syntactical}}$.

Next to syntactical Compatibility, we have to decide, whether Components fit together in terms of semantics as well when we want to bind a system from Dynamic Adaptive Components at runtime. For example, we have to check, whether the Dutch C-Unit is semantically compatible with a German P-Unit. The German P-Unit providing a Service implementing Service Interface `PeripheralUnitIf` needs to behave as it is supposed to by the Dutch C-Unit. In the following we will look at the semantical Compatibility of bindings between C-Units and P-Units within our application example.

Semantical Compatibility of Dynamic Adaptive Components in our Example

The semantical Compatibility between C-Units and P-Units in our example is depicted in Table 3.1. Dutch C-Units are partially incompatible with German P-Units. This is due to the fact, that Dutch C-Units assume, that pulse rate and blood pressure can only be equal to zero at the same time. Since this is not mandatory for German P-Units as, for example, the fingerclip measuring the pulse rate may slip off, they are semantically incompatible in these situations. The different situations measured by P-Units and the corresponding Triage Classes calculated by C-Units are depicted in Figure 3.13.

There is a semantical incompatibility between Dutch C-Units and German P-Units, *if the fingerclip of the P-Unit slips off.*

↓ C-Unit / P-Unit →	German	Dutch
German	Compatible	Compatible
Dutch	Partial Incompatible	Compatible

Table 3.1: Semantical Compatibility of C-Units and P-Units in our Example.

The reason for the semantical incompatibility occurring here is the fact, that the Dutch C-Unit has an implicit assumption regarding P-Units, namely that they must fulfill the invariant `getPulseRate()==0 <==> getDiastolicBloodPressure()==0 <==> getSystolicBloodPressure()==0`. As they did not make this assumption explicit, a System Infrastructure for Dependable Dynamic Adaptive Systems cannot detect this semantical incompatibility and change the Component Binding accordingly.

The Dutch vendor needs to make his assumptions explicit.

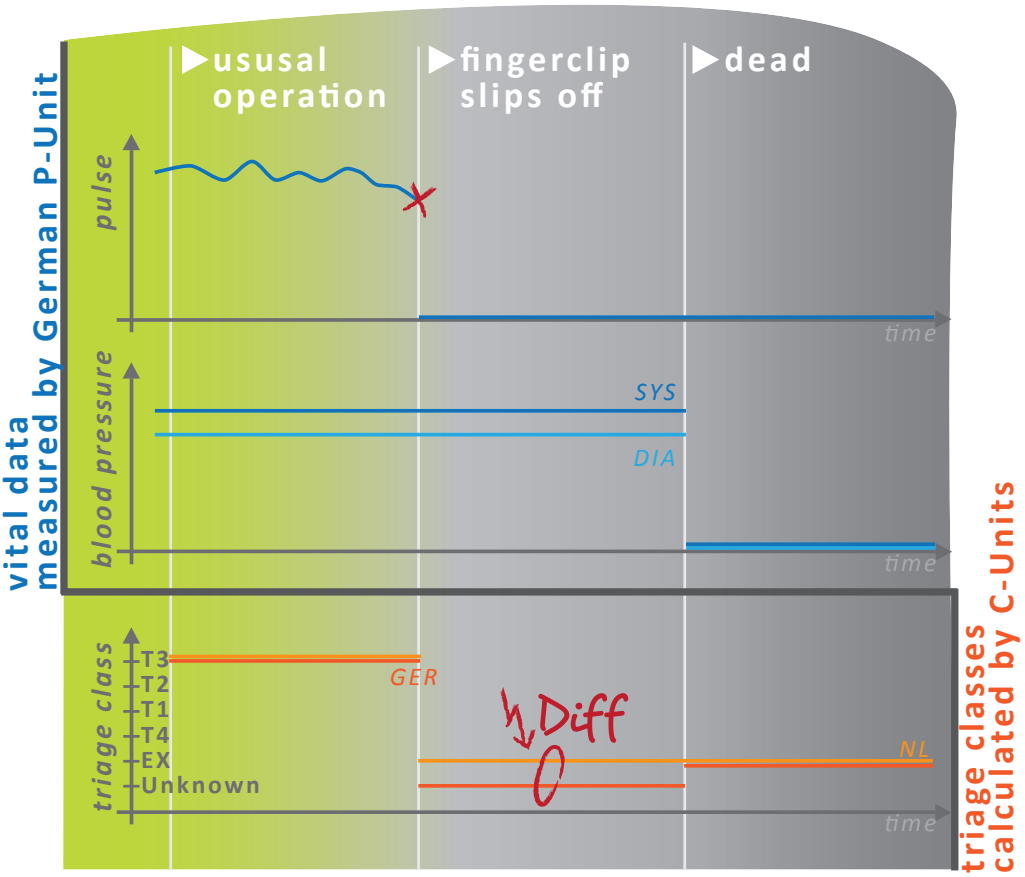


Figure 3.13: Compatibility Between the Different C-Units and P-Units Depending on Their Internal State.

3.5 Requirements Derived from the Example

We now take a closer look at the example and derive requirements, that need to be fulfilled by our approach provided in this thesis to enable Dependable Dynamic Adaptive Systems and to avoid the problems described before. These requirements are introduced and explained in the following.

3.5.1 Support for Adaptation

Since we are talking about Dependable Dynamic *Adaptive* Systems, we need to provide means for developers to express adaptation capabilities. Therefore, the system model needs to enable adaptation of Dependable Dynamic Adaptive Com-

ponents (local adaptation) as well as adaptation of Dependable Dynamic Adaptive Systems (global adaptation).

A first requirement of the overall approach to Dependable Dynamic Adaptive Systems, therefore, is that we need a system model, which supports runtime binding of Dependable Dynamic Adaptive Components. Since we want to achieve systems, which change this Component Binding at runtime in order to integrate Components or remove Components from a system, this is a prerequisite of our approach. Next to the system model we need to provide a System Infrastructure, which automates this Component Binding and, therefore, enables us to build Dependable Dynamic Adaptive Systems from Dependable Dynamic Adaptive Components easily.

We need a system model and an infrastructure that support adaptation.

The application example introduced before needs to be executable based on this System Infrastructure. The system model should be able to describe, whether Components are syntactically compatible. This syntactical Compatibility should be considered on method level, meaning that two Components are considered as syntactically compatible, if for each method respectively attribute in the required Service Interface exists a syntactically compatible method respectively attribute in the provided Service Interface.

Syntactical Compatibility needs to be defined in the system model.

Thus, two Components should be treated as syntactically compatible although they are *not* syntactically *identical* if the provided Service Interface contains additional methods or attributes, which are not contained in the required Service Interface.

Local adaptation capability may be provided by enabling Dependable Dynamic Adaptive Components to support different sets of provided respectively required Services – so called Component Configurations – that can be activated at runtime alternatively. Global adaptation capability may be provided by dynamically changing Component Bindings within a Dependable Dynamic Adaptive System which is one of the key features of our system model.

The System Infrastructure needs to take advantage of this adaptation support. Thus, it needs to be able to switch between Component Configurations of a Dependable Dynamic Adaptive Component and dynamically change Component Bindings as well.

3.5.2 Support of Decoupled Development

Our hypothesis is that the targeted Dependable Dynamic Adaptive Systems are not developed as a whole but evolve over time instead. Therefore, we need to take into account, that no single vendor is developing all Dependable Dynamic Adaptive Components for such a system. Instead we have multiple vendors, which develop Components at different points in time. Moreover these vendors do not target a single system with the Components to be developed but a set of similar systems. Thus, a Domain Architecture specifying Service Interfaces is needed to

Our approach *must not require a system vendor*.

enable Dependable Dynamic Adaptive Systems.

Due to decoupled development we need to provide a guideline for Component vendors describing, what they need to specify for their Dependable Dynamic Adaptive Components in order to achieve a Dependable Dynamic Adaptive System by following our approach. Due to the nature of Dependable Dynamic Adaptive Systems the specification of required Service Interfaces by a Service User needs to be independent from specific realizations by Service Providers – otherwise a Service User would be restricted to a specific Service Provider which contradicts the idea of Dependable Dynamic Adaptive Systems.

3.5.3 Detect Semantical Incompatibilities

We want to detect semantical incompatibilities, *before* they occur.

The probably most challenging requirement is, that we want to be able to detect semantical incompatibilities at runtime by using our approach. These semantical incompatibilities need to be detected before they may cause an effect on the system. Semantical incompatibilities are inconsistencies between a specification of expected behavior defined by a Service User for a Service Interface and the specific behavior of a Service Provider.

We need to consider state changes and corresponding changes of semantical Compatibility.

As semantical Compatibility may change over time, it needs to be checked whenever the state of a Service Provider changes. In our example you can see this, as a Compatibility check between a Dutch C-Unit and a German P-Unit will be successful as long as the fingerclip did not slip off, while it will fail after the clip slipped off. Since checks after state changes correspond more or less to a bisimulation, we want to minimize the amount of checks to a reasonable number.

Therefore, a requirement for our approach is, that it is able to detect semantical incompatibilities in a dependable way but reduces the amount of tests compared to a classic bisimulation approach significantly. Therefore, our approach should enable a description of state spaces, for which different behavior is expected respectively provided.

At runtime this enables us to realize, whenever we enter a state space, where semantical Compatibility needs to be rechecked. Therefore, we can establish a Component Binding between two Dependable Dynamic Adaptive Components, if they became compatible respectively remove a Component Binding between them, if they became incompatible. The Compatibility of a state space is evaluated by test execution, which is performed, whenever a new state space is entered at runtime.

Instead of just simulating the next execution step like in bisimulation, we cover whole sets of method invocations during test execution leading to a higher certainty, that future system execution steps will not cause the system to fail. The testing approach itself, however, leads us to another requirement of our approach.

3.5.4 Free of Side Effects

Executing Compliance Test Cases needs to be free of side effects.

Since we execute semantical Compatibility checks within a running Dependable Dynamic Adaptive System, we need to provide means, which ensure, that these checks have no side effects on the system. This means, that the state of the system must be maintained during these checks – it must not be changed during these checks respectively the system has to be taken to a special testing state during the checks and the state needs to be rolled back after these checks. This must be done careless of a check's results.

These requirements – derived from our application example – will be addressed by our approach based on our system model. We will introduce this formal system model in the following. We explain the different model elements and their relations using our application example.

A rock pile ceases to be a rock pile the moment a single man contemplates it,
bearing within him the image of a cathedral.

Antoine de Saint Exupéry

4

Structural Model for Dependable Dynamic Adaptive Systems

If we want to achieve Dependable Dynamic Adaptive Systems, we first need to have a clear definition, how such systems look like. Therefore, we need to provide a model, which enables us to describe Dependable Dynamic Adaptive Systems. We need to describe two aspects of these systems: The structural aspects of Dependable Dynamic Adaptive Systems as well as the behavioral aspects. The structural aspects of Dependable Dynamic Adaptive Systems are described in the following. This includes the description how a system can be build from several smaller parts, how these parts are structured internally, and how they are bound to each other.

These structural aspects can be divided into two groups: One, describing the structure of specific *instances* of Dependable Dynamic Adaptive Components within a Dependable Dynamic Adaptive Systems. We describe these aspects in Sections 4.3 – 4.6. Next to these instance-related aspects, the structural model also contains *type* aspects, describing the Service Interfaces between the Dependable Dynamic Adaptive Component instances. We describe type-related aspects in Sections 4.7 – 4.8.

Our structural model describes *instances* and *types*.

The behavioral aspects, meaning how a system respectively a part of it behaves in terms of externally observable behavior are described in Section 5. A graphical view of the structural model introduced in the following is depicted in Figure 4.1 – this Figure helps you, to understand our system model. We will use the colors from this Figure during the following sections when we describe the system from our application example using our system model. As a reminder, we will depict

The margins depict model elements from Figure 4.1.

corresponding model elements from Figure 4.1 in the margins, whenever a model element is described in the following sections.

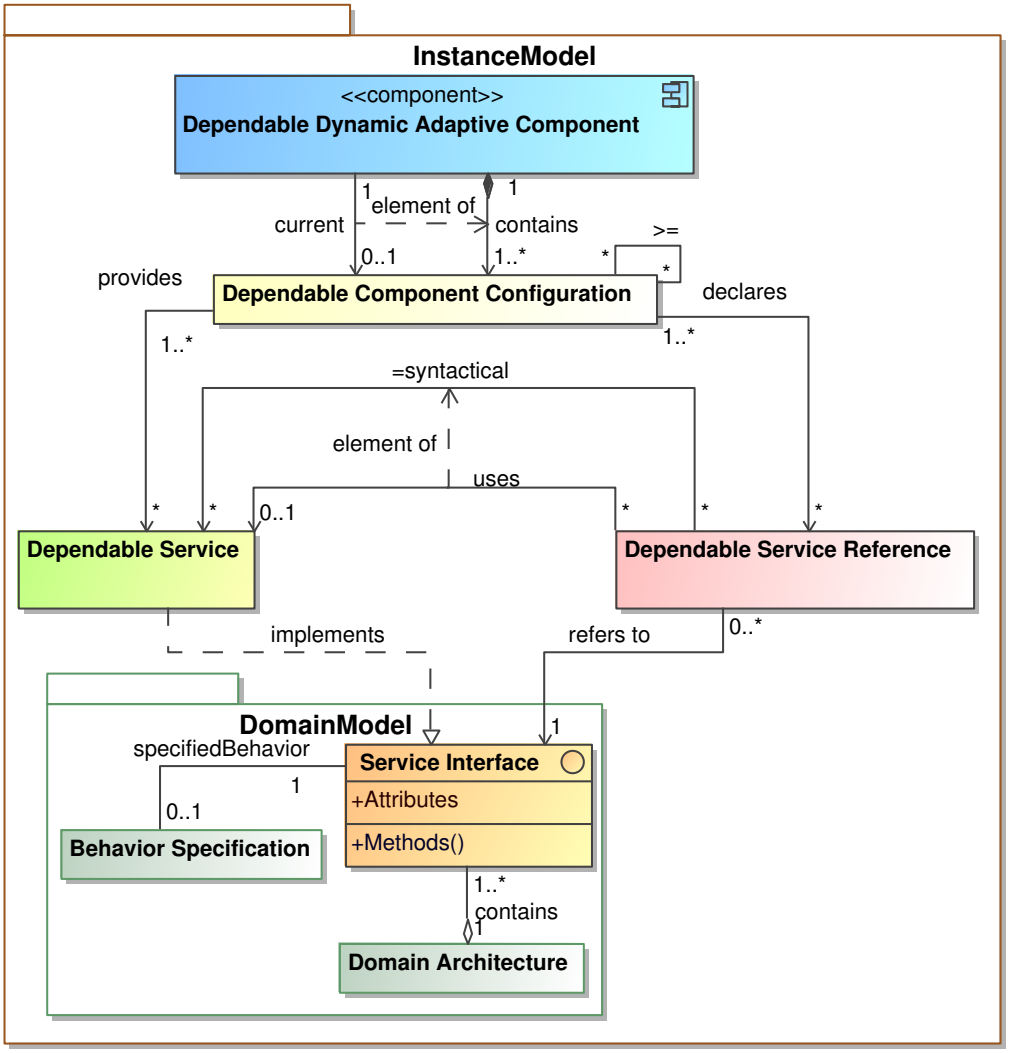


Figure 4.1: Graphical View of our Structural System Model.

The Domain Architecture and Behavior Specifications, which may exist for specific Service Interfaces of a Domain Architecture are depicted only for completeness here. They are not part of our formal model in the following, as our approach to Dependable Dynamic Adaptive Systems does not require them necessarily. The Behavior Specification, however, could be used to verify the correct implementation

of Dependable Services with respect to this specification[1].

In the following we will describe, how we model Dependable Dynamic Adaptive Systems. Therefore, we will have a short look back at our application example introduced before.

4.1 Looking Back at the Application Example

Our model needs to be capable of describing the Emergency Assistance System introduced in section 3. The idea of the Emergency Assistance System was to support medics in a rescue operation. Therefore, medics come along with M-Units. These M-Units display general information about a casualty they are currently treating as well as orders from an incident command.

The incident command is equipped with an IC-Unit enabling them to get a quick overview of the disaster situation. They can get detailed information about each casualty enabling them to figure out, which casualty is most needy of help. Therefore, the IC-Unit provides a solid decision foundation for the incident command. They can figure out, which orders need to be given to medics. They can directly give these orders using the IC-Unit.

Information about casualties is received by C-Units. Medics equip each casualty with a C-Unit, which stores and transmits personal information about the casualty. This C-Unit can also calculate the Triage Class of the casualty, if it is (wirelessly) connected to a so called P-Unit.

A P-Unit contains vital data sensors and, therefore, is able to provide live information about the vital condition of a casualty. Thus, P-Units enable an incident command to recognize changes in vital condition of casualties without medics being present at their place. However, not each casualty needs to be equipped with a P-Unit – e.g. only casualties in critical conditions may be equipped with such a unit due to financial reasons.

Important about our example is the idea, that *different vendors* will develop these Dependable Dynamic Adaptive Components occurring in our example. Thus, different P-Units may appear in our example. They may be different in terms of structure as well as in terms of behavior. In our example we sketched different Dependable Dynamic Adaptive Components developed by a German and a Dutch vendor. When we describe the Emergency Assistance System using our system model we will, therefore, refer to the different instances by adding the postfix *German* respectively *Dutch*.

Components of different vendors are characterized by a postfix *German* respectively *Dutch*.

[1] However, this would not guarantee a Dependable Dynamic Adaptive System, as this would be only verification without considering means for validation. Validation here means answering the question, whether a Dependable Service behaves as expected by a Service User.

Dependability Checkpoints denote situations, which threaten Dependability.

We split this application example into 7 subsequent steps labeled from t_0 to t_0+6 in order to explain our structural system model. These steps denote points in time, where the Dependability of the system is threatened due to a change of structure or behavior within the Dependable Dynamic Adaptive System. We call these points in time *Dependability Checkpoints* in the following. We describe these Dependability Checkpoints shortly in the following and. In addition they are depicted in Figure 4.2.

t_0 denotes the Dependability Checkpoint, when an emergency has occurred. There are no Dependable Dynamic Adaptive Components present, as no helper has arrived at the location yet.

$t_0 + 1$ describes the situation where a German Ambulance arrives, which consists of a single medic, who has his own M-Unit.

$t_0 + 2$ is the point in time where this German medic discovered a casualty and equips him with a C-Unit.

$t_0 + 3$ denotes the Dependability Checkpoint, when this German medic decides that this casualty is so severely wounded, that continuous monitoring of his vital condition is required. Thus, he equips this casualty with an additional P-Unit.

$t_0 + 4$ describes the situation where a Dutch Ambulance arrives, which consists of a single medic, who has his own M-Unit.

$t_0 + 5$ is the point in time where this Dutch medic discovered a casualty and equips him with a C-Unit.

$t_0 + 6$ denotes the Dependability Checkpoint, when this Dutch medic decides that this casualty is so severely wounded, that continuous monitoring of his vital condition is required. Thus, he equips the casualty with an additional P-Unit.

Before we explain our system model top-down, we will first introduce all basic sets, which we use in our system model to capture the structural aspects of Dependable Dynamic Adaptive Systems in the following.

4.2 Basic Sets

Components

The first set, to which our system model refers, is the set *DependableDynamicAdaptiveComponent*. It denotes all Dependable Dynamic Adaptive Components we can think of.

Configurations

Since we want to be able to describe *adaptive* Components, we introduce so called Dependable Component Configurations describing a mapping between

Figure 4.2: Dependability Checkpoints in our Example.

provided Dependable Services and required Dependable Services. All Dependable Component Configurations we can think of are contained in the set *DependableComponentConfiguration*, while the set *DependableService* contains all possible provided Dependable Services and the set *DependableServiceReference* contains all possible required Dependable Services.

Services respectively Service References

Our system model states that Dependable Dynamic Adaptive Components are bound to each other during runtime by binding a provided Dependable Service of a Component to a required Dependable Service of another Component. In order to decide, whether they are syntactically compatible, they are associated with a Service Interface (either by implementing it (provided service) or by declaring it (required service)). The set *ServiceInterface* contains all Service Interfaces we can define in Dependable Dynamic Adaptive Systems.

Service Interfaces

To further specify a Service Interface, the sets *MethodDeclaration* and *AttributeDeclaration* contain the syntactical declaration of all possible methods respectively attributes, which may occur in Dependable Dynamic Adaptive Systems.

Methods and attributes

Methods as well as attributes refer to the basic set *Type*. This set contains all types – primitive types as well as user-defined reference types – possible in Dependable Dynamic Adaptive Systems. Next to *Type*, methods and attributes refer to *String*. This set contains all sequences of characters, which can be used as names for attributes respectively methods.

Types

Strings

Since we assume, that Dependable Dynamic Adaptive Systems are not devel-

oped as a whole, our model must not be a static one, where all instances of Dependable Dynamic Adaptive Components belonging to a system are already known in advance. Therefore, we don't want to define structural respectively behavioral changes of a system s over time in advance but want to be able to reason about structure and behavior of a system at any specific point in time during execution instead.

Systems

Thus, we introduce two sets: Set S contains all Dependable Dynamic Adaptive Systems, which can be described using our system model and set T, which is the set of natural numbers used to denote Dependability Checkpoints, which threaten Dependability of a Dependable Dynamic Adaptive System $s \in S$ during system runtime. There are two triggers, that need to be considered regarding a system's Dependability:

Sequence of Dependability Checkpoints

1. *Structural changes* within a system: For example, the set of Dependable Dynamic Adaptive Components present in the system changes or a binding among Components is changed.

2. *Behavioral changes* within a system: A Dependable Dynamic Adaptive Component behaves differently now, as its internal state has changed.

As we describe only Dependability-relevant points during execution time – so called Dependability Checkpoints – using our system model, $t \in T$ is only increased, if a structural change or a behavioral change threatening Dependability has occurred.

S and T are used, when defining the relations used within our system model in the following. These parameters indicate, that these relations may have a difficult meaning, if ...

1. ... the system is considered at a different Dependability Checkpoint.

2. ... a different system is considered.

In the following we will use these sets to describe Dependable Dynamic Adaptive Systems in our system model. The system from our application example will be described using our system model for a better understanding. This system is called $s_{ae} \in S$ where ae stands for *application example*.

4.3 Dependable Dynamic Adaptive System Structure

Our model is capable of describing a specific Dependable Dynamic Adaptive System as snapshots of the set of Dependable Dynamic Adaptive Component instances ApplicationComponents$_S^T$, that belong to a system at a specific Dependability Checkpoint. This set is changing dynamically whenever an instance is integrated into the system respectively an instance is removed from the system at runtime.

If Component c_1 is integrated into a system s at Dependability Checkpoint $t_i \in T$, this means that $\text{ApplicationComponents}_s^{t_i} = \text{ApplicationComponents}_s^{t_i-1} \cup \{c_1\}$. If a Dependable Dynamic Adaptive Component instance $c_2 \in \text{ApplicationComponents}_s^{t_i}$ is removed from system s at Dependability Checkpoint $t_i + 1$ this leads to a new snapshot of Components: $\text{ApplicationComponents}_s^{t_i+1} = \text{ApplicationComponents}_s^{t_i} \setminus \{c_2\}$.

Summed up, we get a system snapshot $\text{ApplicationComponents}_S^T$, which can be used to describe a system at a specific Dependability Checkpoint:

$$\begin{aligned} &\text{ApplicationComponents}_S^T =_{def} \\ &T \times S \rightarrow \mathcal{P}(\text{DependableDynamicAdaptiveComponent}) \end{aligned} \tag{4.1}$$

If we look at system s_{ae} from the application example introduced before, we have different snapshots of application Component sets during system runtime. These snapshots are described in the following using our system model. A trivial snapshot is the one at Dependability Checkpoint t_0 before the first ambulance has arrived. At this point the set containing the application Components is empty:

$$\text{ApplicationComponents}_{s_{ae}}^{t_0} = \emptyset \tag{4.2}$$

In the following, the German Ambulance arrives at Dependability Checkpoint t_0+1, which may consist of a single medic, who has his own M-Unit[2]. This would result in a snapshot, where we have a single Dependable Dynamic Adaptive Component within the system:

$$\text{ApplicationComponents}_{s_{ae}}^{t_0+1} = \{\text{mUnit}_{\text{German}}\} \tag{4.3}$$

If we now consider, that this German medic, which is equipped with $\text{mUnit}_{\text{German}}$ equips a casualty with a C-Unit at Dependability Checkpoint $t_0 + 2$, the following snapshot would result:

$$\text{ApplicationComponents}_{s_{ae}}^{t_0+2} = \{\text{mUnit}_{\text{German}}, \text{cUnit}_{\text{German}}\} \tag{4.4}$$

In the following he equips this casualty with an additional P-Unit resulting in a snapshot at Dependability Checkpoint $t_0 + 3$:

$$\text{ApplicationComponents}_{s_{ae}}^{t_0+3} = \{\text{mUnit}_{\text{German}}, \text{cUnit}_{\text{German}}, \text{pUnit}_{\text{German}}\} \tag{4.5}$$

In order to be able to show the capabilities of our system model in the following we look at another Dependability Checkpoint: Imagine, that $t_0 + 6$ is the Dependability Checkpoint, where a Dutch medic equipped with a M-Unit has arrived and

[2]We leave out the incident command with it's IC-Unit, since it is not necessary to explain our system model.

has equipped a casualty with a C-Unit and a P-Unit already[3]. Thus, the set of application Components looks as follows at $t_0 + 6$

$$\begin{aligned} \text{ApplicationComponents}_{s_{ae}}^{t_0+6} = \{&\text{mUnit}_{\text{German}}, \text{cUnit}_{\text{German}}, \text{pUnit}_{\text{German}}, \\ &\text{mUnit}_{\text{Dutch}}, \text{cUnit}_{\text{Dutch}}, \text{pUnit}_{\text{Dutch}}\} \end{aligned} \tag{4.6}$$

This set of application Components is also depicted in Figure 4.3.

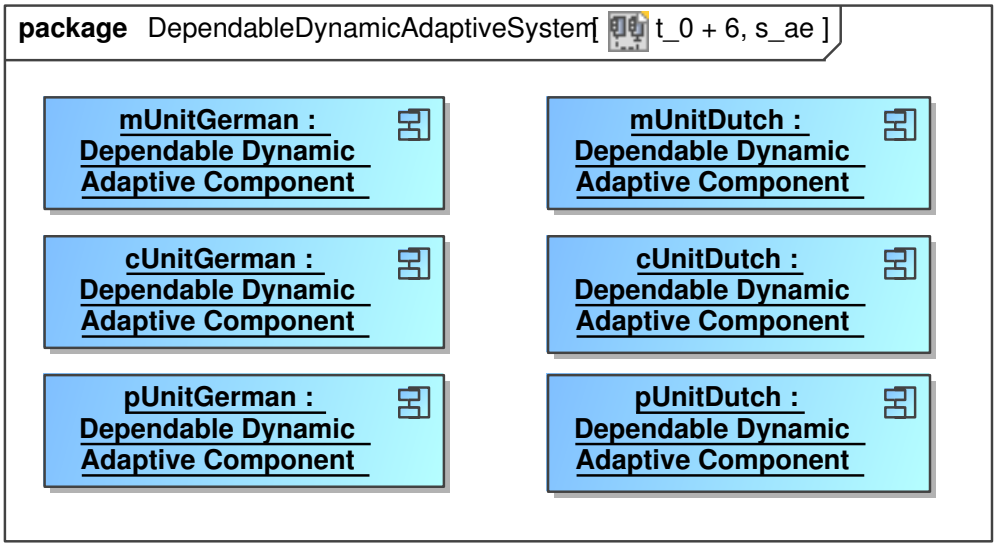


Figure 4.3: Application Components present in our Application Example at $t_0 + 6$.

After looking at the system structure from this very abstract view, the next step towards a structural model for Dependable Dynamic Adaptive Systems is to describe, how a Dependable Dynamic Adaptive Component is further structured.

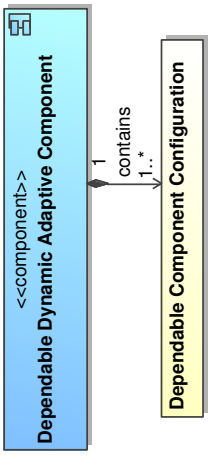
4.4 Dependable Dynamic Adaptive Component Structure

Each Dependable Dynamic Adaptive Component instance in our system may contain several different Dependable Component Configuration instances. They describe the capability of a Dependable Dynamic Adaptive Component depending on Dependable Services, which need to be provided by other Service Providers within a Dependable Dynamic Adaptive System.

[3] We skip $t_0 + 4$ where the Dutch M-Unit has been integrated and $t_0 + 5$ where the Dutch C-Unit has been integrated – they can be found in appendix A for the sake of completeness.

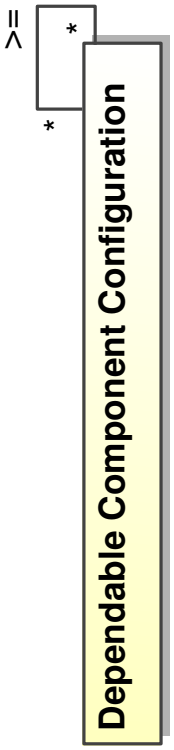
A Dependable Component Configuration, therefore, is a mapping between required and provided Dependable Services. It states, that this Dependable Dynamic Adaptive Component provides these Dependable Services only, if other Service Providers are available, providing those required Dependable Services.

If a Service Provider is available for each required Dependable Service of a specific Dependable Component Configuration, we call this Dependable Component Configuration *activatable*, as it could be activated by a Dependable System Infrastructure. A Component activated in a specific Dependable Component Configuration provides all Dependable Services of this configuration by using its Dependable Service References.



Since we want to be able to bind Dependable Dynamic Adaptive Systems from Dependable Dynamic Adaptive Components *automatically*, we need to reason about the quality of a Dependable Component Configuration. This is necessary to decide, which is the best Dependable Component Configuration, if multiple Dependable Component Configurations of a Dependable Dynamic Adaptive Component are activatable in a system.

As a simple approach we claim, that Dependable Component Configurations need to be ordered regarding their quality by a $\geq$ relation. The intuitive meaning of $\text{configuration}_a \geq \text{configuration}_b$ is, that the quality of configuration_a is equal or better than the quality of configuration_b.



For simplicity we assume in the following, that the quality of two different Dependable Component Configuration instances contained in a specific Dependable Dynamic Adaptive Component instance is only equal, if the Dependable Component Configuration instances are equal (*antisymmetric* relation).

Moreover the $\geq$ relation is *reflexive* meaning that $\text{configuration}_a \geq \text{configuration}_a$. Finally we assume, that the $\geq$ relation is *transitive* for all Dependable Component Configurations contained in a Dependable Dynamic Adaptive Component. This means, that the Dependable Component Configuration instances of a Dependable Dynamic Adaptive Component instance are a partially ordered set.

The $\geq$ relation is dynamic, meaning that it may change over time. A dynamic $\geq$ relation can be useful, when considering quality of service[4]. For example, a Dependable Component Configuration of a Dependable Dynamic Adaptive Component located on a mobile device might be the best while the device's battery is fully charged, while it is worse, if the battery is running low as this Dependable

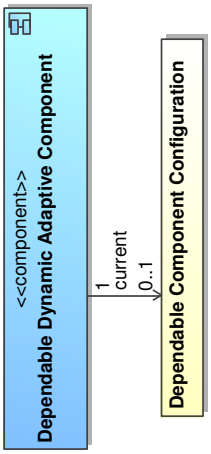
[4]When we refer to quality of service, we do not understand the "classical" definition of network-related quality of service, aiming at the network domain by guaranteeing network-related properties like a certain bandwidth or jitter. Instead we mean *application-related quality of service properties* describing the quality of a service as rated by an end user. For example, a user may prefer a voice output service to a graphical output service and, therefore, rate the quality of voice output services better.

Component Configuration should be avoided to enhance battery life.

Summed up, the following two relations can be used to describe instances of Dependable Dynamic Adaptive Components.

$$\begin{aligned}\text{Contains}_S^T =_{def}\ & \text{DependableDynamicAdaptiveComponent} \times T \times S \rightarrow \\ & \mathcal{P}(\text{DependableComponentConfiguration}) \ : \\ & \text{configuration} \in \text{Contains}_s^t(\text{component}_1) \wedge \\ & \text{configuration} \in \text{Contains}_s^t(\text{component}_2) \Rightarrow \\ & \text{component}_1 = \text{component}_2 \\ & \forall \text{configuration} \in \text{DependableComponentConfiguration}, \\ & \forall s \in S, \forall t \in T, \forall \text{component}_1, \text{component}_2 \in \\ & \text{ApplicationComponents}_s^t\end{aligned} \tag{4.7}$$

$$\begin{aligned}\geq_S^T =_{def}\ & \text{DependableComponentConfiguration} \times \\ & \text{DependableComponentConfiguration} \times T \times S \rightarrow \text{Boolean} \ : \\ & ((c_1 \geq_s^t c_2) \wedge (c_2 \geq_s^t c_3) \Rightarrow c_1 \geq_s^t c_3) \ \wedge (c_1 \geq_s^t c_1) \wedge \\ & (((c_1 \geq_s^t c_2) \wedge (c_2 \geq_s^t c_1)) \Leftrightarrow (c_1 = c_2)) \\ & \forall c \in \text{ApplicationComponents}_s^t, \\ & \forall c_1, c_2, c_3 \in \text{Contains}_s^t(c), \forall s \in S, \forall t \in T\end{aligned} \tag{4.8}$$



By this, a Dependable System Infrastructure can easily determine for each Dependable Dynamic Adaptive Component instance the best Dependable Component Configuration instance, which can be activated.

Next to the set of contained Dependable Component Configurations and their order, we need to describe, which Dependable Component Configuration is the current Dependable Component Configuration, meaning that this Dependable Component Configuration is currently activated by the Dependable System Infrastructure.

$$\begin{aligned}\text{Current}_S^T =_{def}\ & \text{DependableDynamicAdaptiveComponent} \times T \times S \rightarrow \\ & \text{DependableComponentConfiguration} \cup \{\emptyset\} \ : \\ & \text{Current}_s^t(c) \in \text{Contains}_s^t(c) \cup \{\emptyset\} \ \forall s \in S, \forall t \in T, \\ & \forall c \in \text{ApplicationComponents}_s^t\end{aligned} \tag{4.9}$$

Applied to our system s_{ae} from our application example, at Dependability Checkpoint $t_0 + 6$ we know six Components, namely $\text{mUnit}_{\text{German}}$, $\text{cUnit}_{\text{German}}$, $\text{pUnit}_{\text{German}}$, $\text{mUnit}_{\text{Dutch}}$, $\text{cUnit}_{\text{Dutch}}$, and $\text{pUnit}_{\text{Dutch}}$. Each M-Unit ($\text{mUnit}_{\text{German}}$ respectively $\text{mUnit}_{\text{Dutch}}$) contains two Dependable Component Configurations:

1. mConfiguration1$_{\text{German}}$ respectively mConfiguration1$_{\text{Dutch}}$, where no Service Interface is required. This represents a medic, which is currently not treating a casualty.

2. mConfiguration2$_{\text{German}}$ respectively mConfiguration2$_{\text{Dutch}}$, where Service Interface `CUnitServiceIf` is required. This represents a medic, which is currently treating a casualty equipped with a C-Unit.

Each C-Unit (cUnit$_{\text{German}}$ respectively cUnit$_{\text{Dutch}}$) from our application example contains two Dependable Component Configurations as well:

1. cConfiguration1$_{\text{German}}$ respectively cConfiguration1$_{\text{Dutch}}$, where no Service Interface is required. This represents a casualty, where a medic manually sets the Triage Class – no vital data sensors are present.

2. cConfiguration2$_{\text{German}}$ respectively cConfiguration2$_{\text{Dutch}}$, where Service Interface `PUnitServiceIf` is required. This represents a casualty, which is currently equipped with vital data sensors monitoring his physical condition. Therefore, the Triage Class of this casualty can be calculated automatically.

The P-Units (pUnit$_{\text{German}}$ respectively pUnit$_{\text{Dutch}}$) contain only a single Dependable Component Configuration (pConfiguration1$_{\text{German}}$ respectively pConfiguration1$_{\text{Dutch}}$) where they do not require any Service Interface and provide Service Interface `PUnitServiceIf`.

The Dependable Component Configurations contained in the Dependable Dynamic Adaptive Components from our system s_{ae} of our application example at $t_0 + 6$ are described formally according to our system model in the following.

$$\text{Contains}_{s_{ae}}^{t_0+6}(\text{mUnit}_{\text{German}}) = \{\text{mConfiguration1}_{\text{German}}, \text{mConfiguration2}_{\text{German}}\} \tag{4.10}$$

$$\text{Contains}_{s_{ae}}^{t_0+6}(\text{cUnit}_{\text{German}}) = \{\text{cConfiguration1}_{\text{German}}, \text{cConfiguration2}_{\text{German}}\} \tag{4.11}$$

$$\text{Contains}_{s_{ae}}^{t_0+6}(\text{pUnit}_{\text{German}}) = \{\text{pConfiguration1}_{\text{German}}\} \tag{4.12}$$

$$\text{Contains}_{s_{ae}}^{t_0+6}(\text{mUnit}_{\text{Dutch}}) = \{\text{mConfiguration1}_{\text{Dutch}}, \text{mConfiguration2}_{\text{Dutch}}\} \tag{4.13}$$

$$\text{Contains}_{s_{ae}}^{t_0+6}(\text{cUnit}_{\text{Dutch}}) = \{\text{cConfiguration1}_{\text{Dutch}}, \text{cConfiguration2}_{\text{Dutch}}\} \tag{4.14}$$

$$\text{Contains}_{s_{ae}}^{t_0+6}(\text{pUnit}_{\text{Dutch}}) = \{\text{pConfiguration1}_{\text{Dutch}}\} \tag{4.15}$$

In our application example the Dependable Component Configuration requiring Service Interface `CUnitServiceIf` is the best configuration of M-Units followed by the Dependable Component Configuration requiring nothing. A similar order of the Dependable Component Configurations can be derived for the C-Units from our application example. The configurations are, therefore, ordered as depicted in the following formulas.

$$\text{mConfiguration2}_{\text{German}} \geq_{s_{ae}}^{t_0+6} \text{mConfiguration1}_{\text{German}} \tag{4.16}$$

$$\text{cConfiguration2}_{\text{German}} \geq_{s_{ae}}^{t_0+6} \text{cConfiguration1}_{\text{German}} \tag{4.17}$$

$$\text{mConfiguration2}_{\text{Dutch}} \geq_{s_{ae}}^{t_0+6} \text{mConfiguration1}_{\text{Dutch}} \tag{4.18}$$

$$\text{cConfiguration2}_{\text{Dutch}} \geq_{s_{ae}}^{t_0+6} \text{cConfiguration1}_{\text{Dutch}} \tag{4.19}$$

The remaining question regarding the configuration structure within our application example at this point is, which Dependable Component Configuration is the current Dependable Component Configuration for each Dependable Dynamic Adaptive Component at the different Dependability Checkpoints. As an example[5], we will have a look at $t_0 + 2$ and $t_0 + 6$.

At $t_0 + 2$ we only have two Dependable Dynamic Adaptive Components present within our Dependable Dynamic Adaptive System: a German M-Unit and a German C-Unit. Thus, the German M-Unit is in its best Dependable Component Configuration ($\text{mConfiguration2}_{\text{German}}$), as a C-Unit is available providing `CUnitServiceIf`. The German C-Unit is in its worst Dependable Component Configuration ($\text{cConfiguration1}_{\text{German}}$) as no Service Provider of `PUnitServiceIf` is present.

$$\text{Current}_{s_{ae}}^{t_0+2}(\text{mUnit}_{\text{German}}) = \text{mConfiguration2}_{\text{German}} \tag{4.20}$$

$$\text{Current}_{s_{ae}}^{t_0+2}(\text{cUnit}_{\text{German}}) = \text{cConfiguration1}_{\text{German}} \tag{4.21}$$

At $t_0 + 6$ for each Dependable Dynamic Adaptive Component from our example their best Dependable Component Configuration is the current Dependable

[5] A complete overview of the current Dependable Component Configurations during execution of our application example can be found in Appendix A.

Component Configuration, as Service Providers for all required Service Interfaces of the Service Users are available.

$$\mathsf{Current}_{s_{ae}}^{t_0+6}(\mathsf{mUnit}_{\mathsf{German}}) = \mathsf{mConfiguration2}_{\mathsf{German}} \tag{4.22}$$

$$\mathsf{Current}_{s_{ae}}^{t_0+6}(\mathsf{cUnit}_{\mathsf{German}}) = \mathsf{cConfiguration2}_{\mathsf{German}} \tag{4.23}$$

$$\mathsf{Current}_{s_{ae}}^{t_0+6}(\mathsf{pUnit}_{\mathsf{German}}) = \mathsf{pConfiguration1}_{\mathsf{German}} \tag{4.24}$$

$$\mathsf{Current}_{s_{ae}}^{t_0+6}(\mathsf{mUnit}_{\mathsf{Dutch}}) = \mathsf{mConfiguration2}_{\mathsf{Dutch}} \tag{4.25}$$

$$\mathsf{Current}_{s_{ae}}^{t_0+6}(\mathsf{cUnit}_{\mathsf{Dutch}}) = \mathsf{cConfiguration2}_{\mathsf{Dutch}} \tag{4.26}$$

$$\mathsf{Current}_{s_{ae}}^{t_0+6}(\mathsf{pUnit}_{\mathsf{Dutch}}) = \mathsf{pConfiguration1}_{\mathsf{Dutch}} \tag{4.27}$$

The Component structure resulting at $t_0 + 6$ is depicted in Figure 4.4. Current Dependable Component Configurations are marked by green checks in this figure.

In the following we will continue investigating the structure of a Dependable Dynamic Adaptive Component by taking a closer look at the internal structure of Dependable Component Configurations.

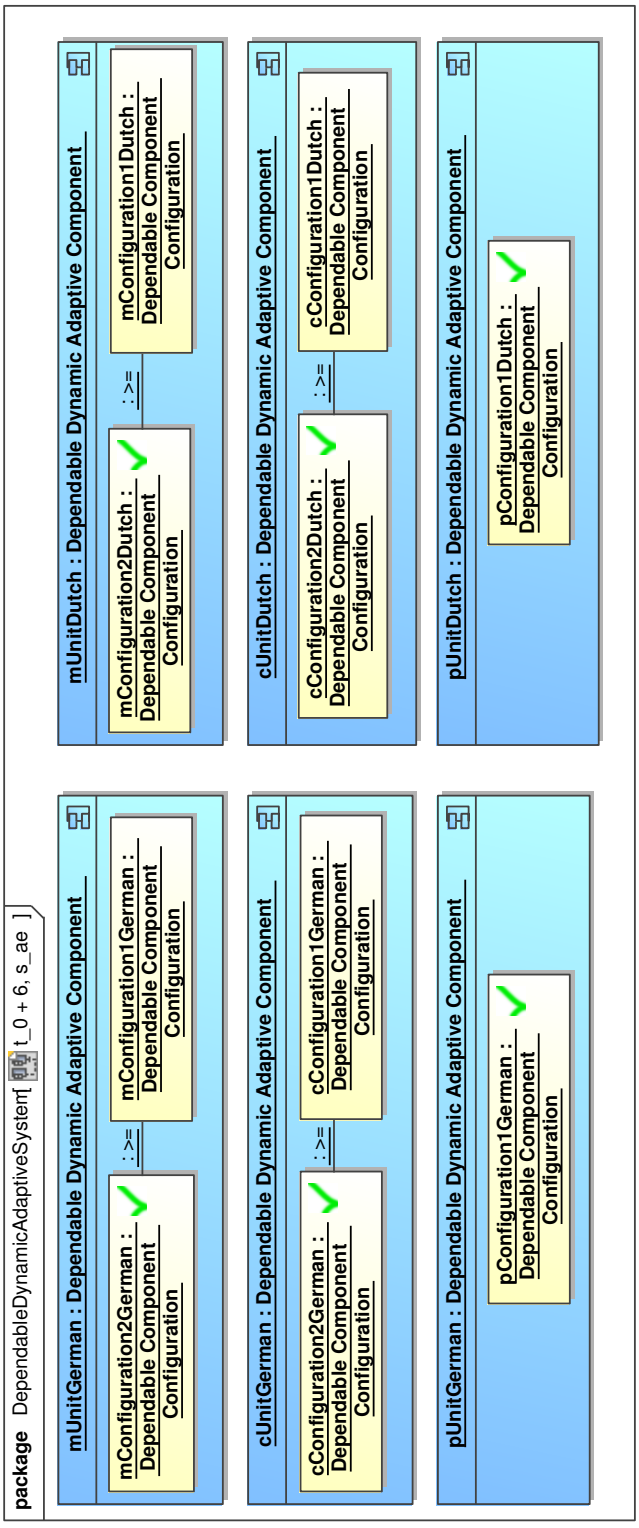
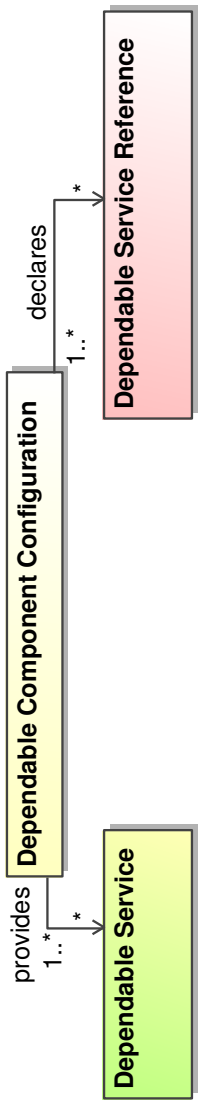


Figure 4.4: A Closer Look at the Application Components Present in our Application Example at $t_0 + 6$.

4.5 Dependable Component Configuration Structure



A Dependable Component Configuration instance is specified by two sets – one set containing all *provided* Dependable Services, the other set containing Dependable Service References for all Dependable Services *declared* as required by this specific Dependable Component Configuration instance.

The concept of a Dependable Component Configuration is, that all provided Dependable Service instances of a currently active Dependable Component Configuration can be used by other Dependable Dynamic Adaptive Components which act as Service Users. On the other hand all declared Dependable Service References of the currently active Dependable Component Configuration need to point to Dependable Service instances, which are provided by currently active Dependable Component Configurations of other Dependable Dynamic Adaptive Components acting as Service Providers.

The provided respectively required Dependable Services are denoted by the relations Provides and Declares in our system model.

$$
\begin{aligned}
\text{Provides}_S^T =_{def}\ & \text{DependableComponentConfiguration} \times T \times S \rightarrow \\
& \mathcal{P}(\text{DependableService}) \ : \\
& \mathsf{c}_1 \neq \mathsf{c}_2 \Rightarrow \text{Provides}_s^t(\text{conf}_1) \cap \text{Provides}_s^t(\text{conf}_2) = \emptyset \\
& \forall \mathsf{c}_1, \mathsf{c}_2 \in \text{DependableDynamicAdaptiveComponent}, \\
& \forall \text{conf}_1 \in \text{Contains}_s^t(\mathsf{c}_1), \forall \text{conf}_2 \in \text{Contains}_s^t(\mathsf{c}_2), \\
& \forall s \in S, \forall t \in T
\end{aligned}
\tag{4.28}
$$

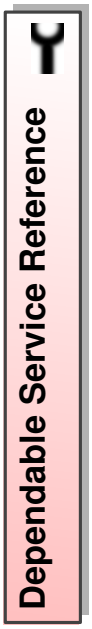
$$
\begin{aligned}
\text{Declares}_S^T =_{def}\ & \text{DependableComponentConfiguration} \times T \times S \rightarrow \\
& \mathcal{P}(\text{DependableServiceReference}) \ : \\
& \mathsf{c}_1 \neq \mathsf{c}_2 \wedge \text{conf}_1 \in \text{Contains}_s^t(\mathsf{c}_1) \wedge \forall \text{conf}_2 \in \text{Contains}_s^t(\mathsf{c}_2) \Rightarrow \\
& \text{Declares}_s^t(\text{conf}_1) \cap \text{Declares}_s^t(\text{conf}_2) = \emptyset \\
& \forall \mathsf{c}_1, \mathsf{c}_2 \in \text{DependableDynamicAdaptiveComponent}, \\
& \forall s \in S, \forall t \in T
\end{aligned}
\tag{4.29}
$$

In the following we will only show the structure of those Dependable Component Configurations of the German Dependable Dynamic Adaptive Components[6].

In our example the German M-Unit provides a Dependable Service $\text{mUnitService}_{\text{German}}$ in all Component Configurations. This may be the same Dependable Service instance for all configurations.

[6] You can find a complete specification of our application example in Appendix A.

In order to be able To provide this Dependable Service, it may require Service Interface `CUnitServiceIf` or no Service Interface, depending on the Dependable Component Configuration instance – this corresponds to situations, where a medic is assigned to a casualty or where he is not assigned. Thus, it declares a Dependable Service Reference $\text{cUnitReference}_{\text{German}}$ or no Dependable Service Reference depending on the Dependable Component Configuration instance.



The two Component Configurations of the German C-Unit provide a Dependable Service $\text{cUnitService}_{\text{German}}$ and declare no Dependable Service Reference respectively a Dependable Service Reference $\text{pUnitReference}_{\text{German}}$ – this corresponds to situations, where no P-Unit is attached to this casualty respectively where this casualty is equipped with a P-Unit.

$$\begin{aligned}
\text{Provides}_{s_{ae}}^{t_0+6}(\text{mConfiguration1}_{\text{German}}) &= \{\text{mUnitService}_{\text{German}}\} \\
\text{Declares}_{s_{ae}}^{t_0+6}(\text{mConfiguration1}_{\text{German}}) &= \emptyset \\
\text{Provides}_{s_{ae}}^{t_0+6}(\text{mConfiguration2}_{\text{German}}) &= \{\text{mUnitService}_{\text{German}}\} \\
\text{Declares}_{s_{ae}}^{t_0+6}(\text{mConfiguration2}_{\text{German}}) &= \{\text{cUnitReference}_{\text{German}}\}
\end{aligned} \tag{4.30}$$

$$\begin{aligned}
\text{Provides}_{s_{ae}}^{t_0+6}(\text{cConfiguration1}_{\text{German}}) &= \{\text{cUnitService}_{\text{German}}\} \\
\text{Declares}_{s_{ae}}^{t_0+6}(\text{cConfiguration1}_{\text{German}}) &= \emptyset \\
\text{Provides}_{s_{ae}}^{t_0+6}(\text{cConfiguration2}_{\text{German}}) &= \{\text{cUnitService}_{\text{German}}\} \\
\text{Declares}_{s_{ae}}^{t_0+6}(\text{cConfiguration2}_{\text{German}}) &= \{\text{pUnitReference}_{\text{German}}\}
\end{aligned} \tag{4.31}$$

$$\begin{aligned}
\text{Provides}_{s_{ae}}^{t_0+6}(\text{pConfiguration1}_{\text{German}}) &= \{\text{pUnitService}_{\text{German}}\} \\
\text{Declares}_{s_{ae}}^{t_0+6}(\text{pConfiguration1}_{\text{German}}) &= \emptyset
\end{aligned} \tag{4.32}$$

The German P-Unit has only a single Dependable Component Configuration. In this configuration it provides a Dependable Service $\text{pUnitService}_{\text{German}}$ and requires no Dependable Service.

The structure of the Dependable Dynamic Adaptive Components and their Dependable Component Configurations resulting at $t_0 + 6$ is depicted in Figure 4.5. Following the UML notation provided Dependable Services are represented by a filled circle, whereas declared Dependable Service References are represented by an unfilled semicircle as depicted in the margin.

In the following we will show, how Dependable Dynamic Adaptive Components are bound to each other by binding Dependable Service References to Dependable Services.

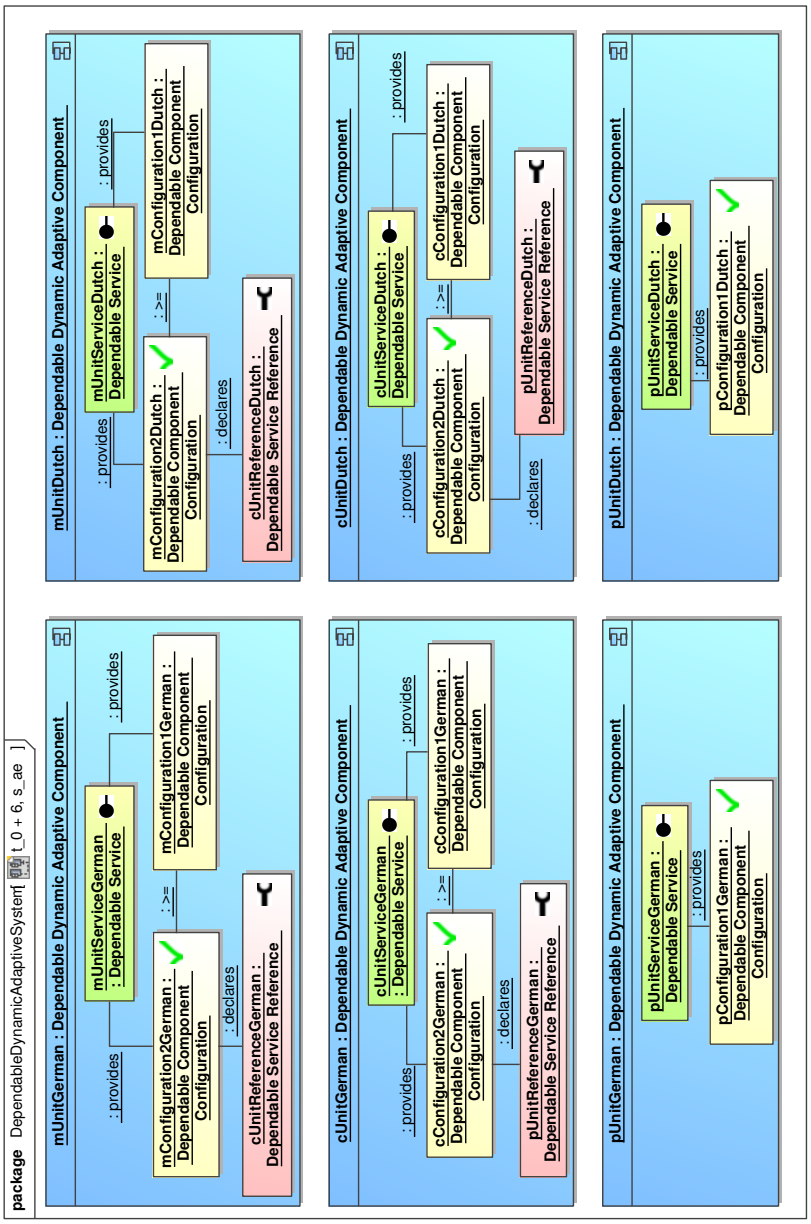
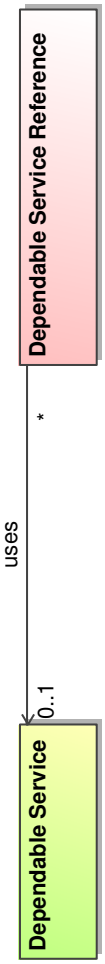


Figure 4.5: A Closer Look at the Application Components and their Configurations at $t_0 + 6$.



4.6 Binding Structure

Next to the declaration of required and provided Dependable Services, we need to be able to describe, which Dependable Service Reference uses which Dependable Service – this is also called a *binding*. This binding is described by the Uses relation in our system model. It associates a Dependable Service Reference with a Dependable Service. The Dependable Service used by a Dependable Service Reference has to be provided by another Dependable Dynamic Adaptive Component instance in its Current Configuration.

$$\begin{aligned} \mathsf{Uses}_S^T =_{def} \mathsf{DependableServiceReference} \times T \times S \rightarrow \\ \mathsf{DependableService} \cup \{\emptyset\} \ : \\ \mathsf{Uses}_s^t \neq \emptyset \Rightarrow \exists \mathsf{c}_{\mathsf{prov}} \in \mathsf{ApplicationComponents}_s^t \mid \\ \mathsf{c}_{\mathsf{prov}} \neq \mathsf{c}_{\mathsf{req}} \wedge \mathsf{Uses}_s^t(\mathsf{ref}) \in \mathsf{Provides}_s^t(\mathsf{Current}_s^t(\mathsf{c}_{\mathsf{prov}})) \\ \forall \mathsf{ref} \in \mathsf{Declares}_s^t(\mathsf{Current}_s^t(\mathsf{c}_{\mathsf{req}})), \\ \forall \mathsf{c}_{\mathsf{req}} \in \mathsf{ApplicationComponents}_s^t, \forall s \in S, \forall t \in T \end{aligned} \tag{4.33}$$

For our application example at t_0+6 we may observe, that the German M-Unit is bound to the German C-Unit, which is bound to the German P-Unit[7]. The Uses relation in our example system, therefore, is defined as follows at $t_0 + 6$:

$$\mathsf{Uses}_{s_{ae}}^{t_0+6}(\mathsf{cUnitReference}_{\mathsf{German}}) = \mathsf{cUnitService}_{\mathsf{German}} \tag{4.34}$$

$$\mathsf{Uses}_{s_{ae}}^{t_0+6}(\mathsf{pUnitReference}_{\mathsf{German}}) = \mathsf{pUnitService}_{\mathsf{German}} \tag{4.35}$$

The bindings of Dependable Dynamic Adaptive Components at $t_0 + 6$ in our example are depicted in Figure 4.6 by uses relations between Dependable Services and Dependable Service References.

Now that we have defined a binding structure by introducing a Uses relation, we can refine the Current relation from Equation 4.9. We can now specify, that each current Dependable Component Configuration needs to have each Dependable Service Reference bound to a Dependable Service provided by a different Dependable Dynamic Adaptive Component in its current Dependable Component Configurations. The revised Current relation is specified as follows.

[7] This is only one specific binding, you can think of in our application example. We assume this specific binding in the following – of course, the German M-Unit could be bound to a Dutch C-Unit at $t_0 + 6$ as well. Any binding between German and Dutch Dependable Dynamic Adaptive Components which are syntactically compatible would be possible here.

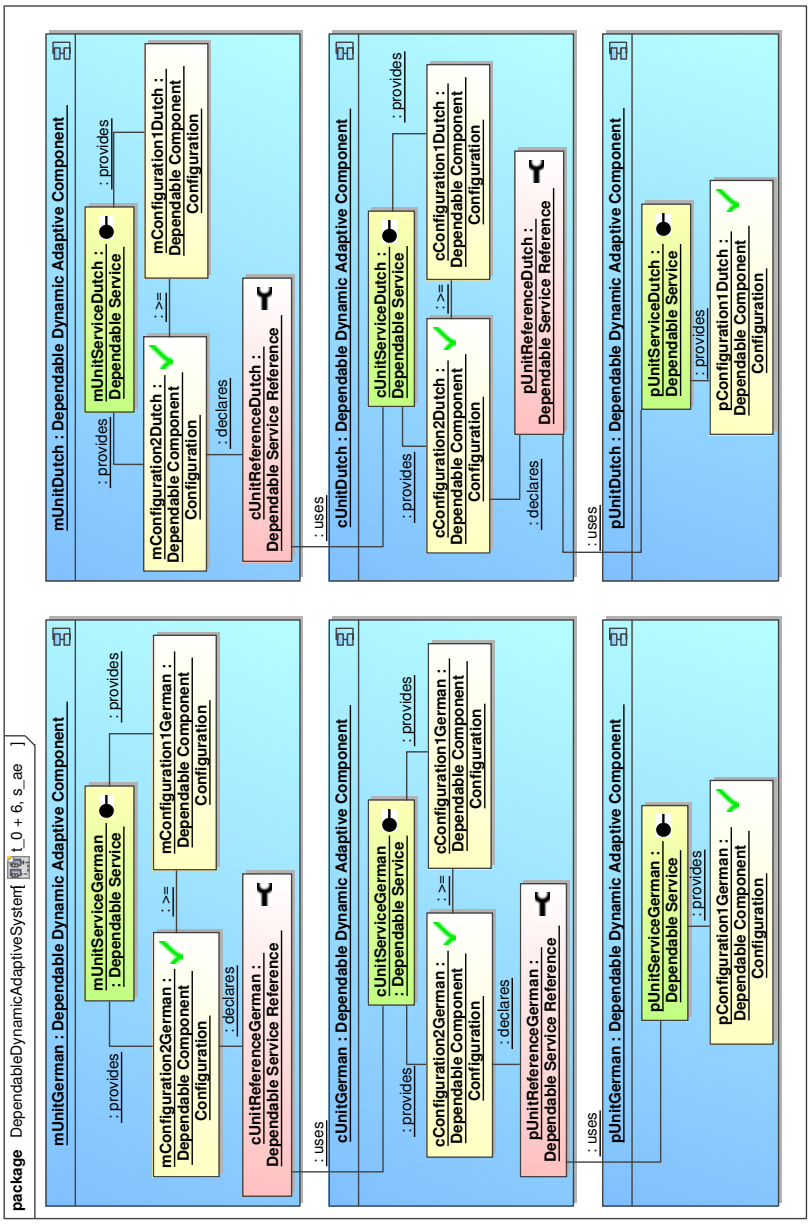
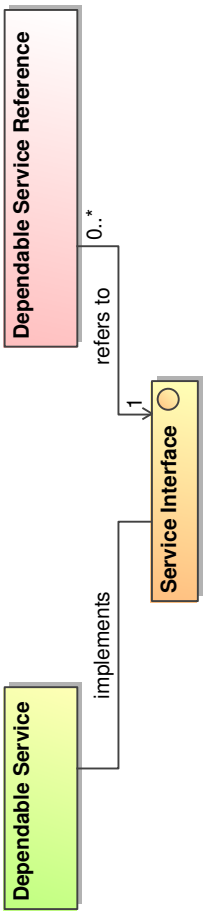


Figure 4.6: A Closer Look at the Bindings of the Application Components at $t_0 + 6$.

$$\begin{aligned}\text{Current}_S^T =_{def}\ & \text{DependableDynamicAdaptiveComponent} \times T \times S \rightarrow \\ & \text{DependableComponentConfiguration} \cup \{\emptyset\}\ : \\ & (\text{Current}_s^t(\text{c}) \in \text{Contains}_s^t(\text{c}) \cup \{\emptyset\}) \wedge \\ & (\text{Current}_s^t(\text{c}) \neq \emptyset\ \Rightarrow \text{Uses}_s^t(\text{ref}) \neq \emptyset) \forall s \in S, \forall t \in T, \\ & \forall \text{c} \in \text{ApplicationComponents}_s^t, \forall \text{ref} \in \text{Declares}_s^t(\text{Current}(\text{c}))\end{aligned} \tag{4.36}$$



In the following we will have a closer look, how the type of a Dependable Service respectively Dependable Service Reference is further described in our system model by associating them with Service Interfaces. We assume in our system model, that this type information is not *dynamic*, meaning that a Service Interface provided respectively required by a Dependable Dynamic Adaptive Component has the same type all over the time and in each system. Therefore, the following relations do not consider T and S anymore.

4.7 Dependable Service and Dependable Service Reference Structure

Both, Dependable Services and Dependable Service References are associated with Service Interfaces. Dependable Services implement a Service Interface, while Dependable Service References refer to a Service Interface to denote their syntactical requirements regarding a specific Dependable Service they require. Thus, the relations Implements and RefersTo are defined in our structural model to express these associations. These relations are static: instances of Dependable Services or Dependable Service References cannot change a Service Interface they implement respectively refer to over time. The relations are, therefore, defined as follows:

$$\text{Implements} =_{def} \text{DependableService} \rightarrow \text{ServiceInterface} \tag{4.37}$$

$$\text{RefersTo} =_{def} \text{DependableServiceReference} \rightarrow \text{ServiceInterface} \tag{4.38}$$

In our application example Dependable Service $\text{mUnitService}_{\text{German}}$ provided by the German M-Unit implements Service Interface mUnitServiceIf while $\text{cUnitService}_{\text{German}}$ provided by the German C-Unit implements Service Interface cUnitServiceIf. Finally, Dependable Service $\text{pUnitService}_{\text{German}}$ provided by the German P-Unit implements Service Interface pUnitServiceIf.

When looking at Dependable Service References, $\text{cUnitReference}_{\text{German}}$ refers to Service Interface cUnitServiceIf and $\text{pUnitReference}_{\text{German}}$ refers to Service Interface pUnitServiceIf.

Summed up those Dependable Services provided respectively those Dependable Service References declared by the German Dependable Dynamic Adaptive Components in our example are associated with Service Interfaces as follows:

$$\text{Implements}(\text{mUnitService}_{\text{German}}) = \text{mUnitServiceIf} \tag{4.39}$$

$$\text{Implements}(\text{cUnitService}_{\text{German}}) = \text{cUnitServiceIf} \tag{4.40}$$

$$\text{Implements}(\text{pUnitService}_{\text{German}}) = \text{pUnitServiceIf} \tag{4.41}$$

$$\text{RefersTo}(\text{cUnitReference}_{\text{German}}) = \text{cUnitServiceIf} \tag{4.42}$$

$$\text{RefersTo}(\text{pUnitReference}_{\text{German}}) = \text{pUnitServiceIf} \tag{4.43}$$

You can see these type relations within our application example in the middle of Figure 4.7. In the following we will have a closer look at a Service Interface's internal structure.

4.8 Service Interface Structure

Service Interface = methods + attributes.

The structure of a Service Interface is defined by two sets. The first set contains a declaration of all methods, while the second contains a declaration of all attributes of a Service Interface.

Methods associated with a provided Dependable Service instance (through the Implements relation) can be called by a Service User through his Dependable Service Reference. Attributes associated with a provided Dependable Service can be set or queried by a Service User of this Dependable Service as well using the same mechanism.

$$\text{Methods} =_{def} \text{ServiceInterface} \rightarrow \mathcal{P}(\text{MethodDeclaration}) \tag{4.44}$$

$$\text{Attributes} =_{def} \text{ServiceInterface} \rightarrow \mathcal{P}(\text{AttributeDeclaration}) \tag{4.45}$$

If we look at Service Interface cUnitServiceIf and pUnitServiceIf provided by C-Units respectively P-Units from our application example, this looks as follows.

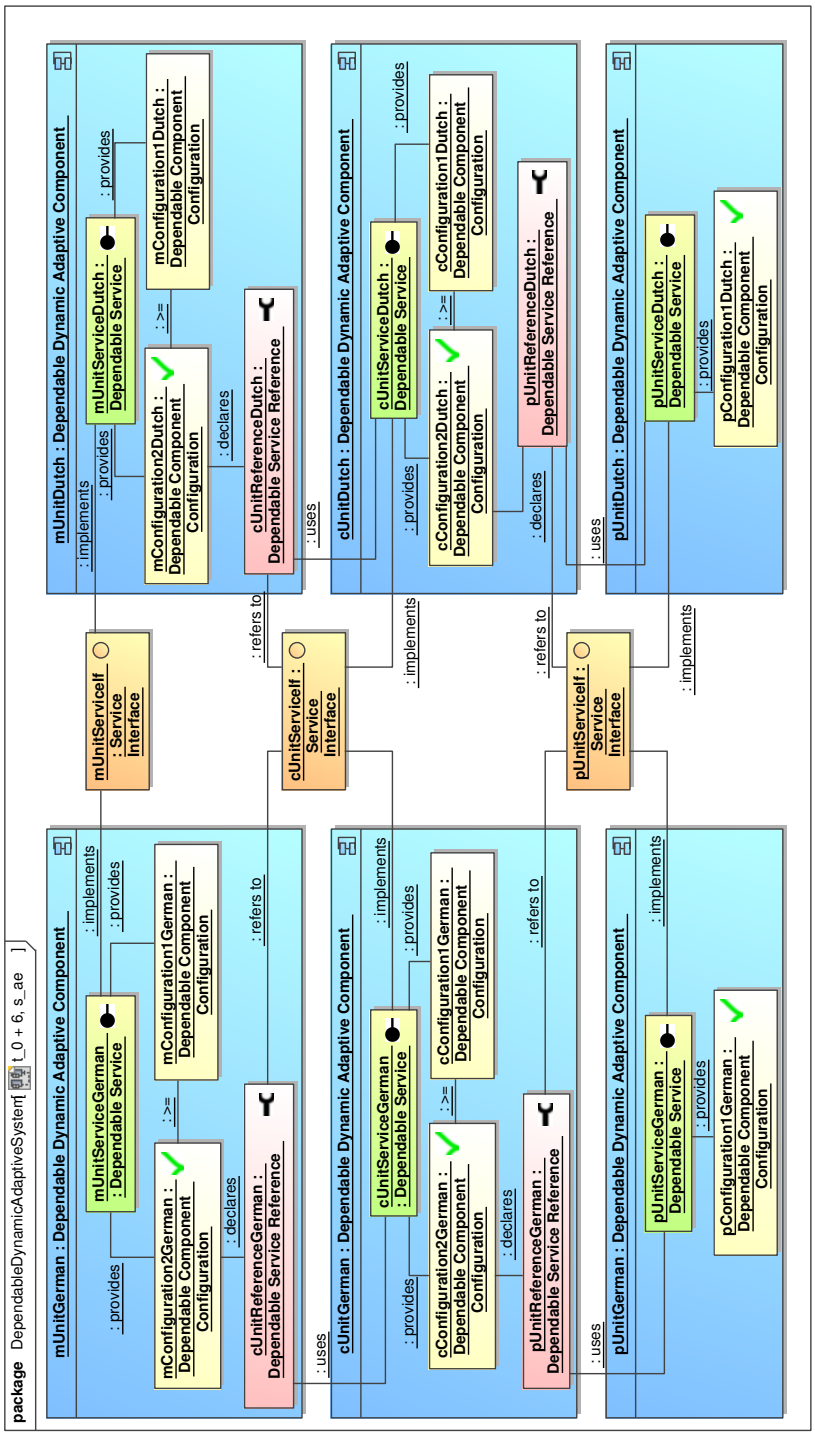


Figure 4.7: A Closer Look at Dependable Services and Dependable Service References from our Application Example.

$$\text{Methods}(\text{cUnitServiceIf}) = \{\text{getLongitude}_C, \text{getLatitude}_C, \text{getTriageClass}_C, \text{setTriageClass}_C, \text{getAssignedPeripheralUnit}_C\} \tag{4.46}$$

$$\text{Methods}(\text{pUnitServiceIf}) = \{\text{getSystolicBloodPressure}_P, \text{getDiastolicBloodPressure}_P, \text{getPulseRate}_P\} \tag{4.47}$$

$$\text{Attributes}(\text{cUnitServiceIf}) = \emptyset \tag{4.48}$$

$$\text{Attributes}(\text{pUnitServiceIf}) = \emptyset \tag{4.49}$$

The specification of those Service Interfaces offered respectively required by the M-Units from our application example can be found in Appendix A.

Now that we specified the Service Interfaces, we need to define, how the declarations of methods respectively attributes look like. We will take a look at these declarations in the following.

4.8.1 Method Declaration

A method declaration contains three different types of structural information:

1. The name of this method.
2. The type of this method's return value.
3. A set of attributes, which have to be passed to this method as input parameters.

More formally this results in three relations, which are denoted in the following.

$$\text{MethodName} =_{def} \text{MethodDeclaration} \rightarrow \text{String} \tag{4.50}$$

$$\text{ReturnType} =_{def} \text{MethodDeclaration} \rightarrow \text{Type} \tag{4.51}$$

$$\begin{aligned} \text{Parameters} =_{def} \text{MethodDeclaration} \rightarrow \mathcal{P}(\text{AttributeDeclaration}) : \\ \text{Parameters}(m) \cap \text{Attributes}(c) = \emptyset \\ m_1 \neq m_2 \Rightarrow \text{Parameters}(m_1) \cap \text{Parameters}(m_2) = \emptyset \\ \forall m, m_1, m_2 \in \text{MethodDeclaration}, \\ \forall c \in \text{DependableDynamicAdaptiveComponent} \end{aligned} \tag{4.52}$$

If we look at those methods provided by Service Interface cUnitServiceIf, which is the interface between M-Units and C-Units in our application example, their declaration looks as follows:

$$\begin{aligned} \text{MethodName}(\text{getLongitude}_C) &= \text{"getLongitude"} \\ \text{ReturnType}(\text{getLongitude}_C) &= \text{int} \\ \text{Parameters}(\text{getLongitude}_C) &= \emptyset \end{aligned} \tag{4.53}$$

$$\begin{aligned} \text{MethodName}(\text{getLatitude}_C) &= \text{"getLatitude"} \\ \text{ReturnType}(\text{getLatitude}_C) &= \text{int} \\ \text{Parameters}(\text{getLatitude}_C) &= \emptyset \end{aligned} \tag{4.54}$$

$$\begin{aligned} \text{MethodName}(\text{setTriageClass}_C) &= \text{"setTriageClass"} \\ \text{ReturnType}(\text{setTriageClass}_C) &= \text{void} \\ \text{Parameters}(\text{setTriageClass}_C) &= \{\text{triageClass}_C\} \end{aligned} \tag{4.55}$$

$$\begin{aligned} \text{MethodName}(\text{getTriageClass}_C) &= \text{"getTriageClass"} \\ \text{ReturnType}(\text{getTriageClass}_C) &= \text{TriageClass} \\ \text{Parameters}(\text{getTriageClass}_C) &= \emptyset \end{aligned} \tag{4.56}$$

$$\begin{aligned} \text{MethodName}(\text{getAssignedPeripheralUnit}_C) &= \text{"getAssignedPeripheralUnit"} \\ \text{ReturnType}(\text{getAssignedPeripheralUnit}_C) &= \text{PUnitServiceIf} \\ \text{Parameters}(\text{getAssignedPeripheralUnit}_C) &= \emptyset \end{aligned} \tag{4.57}$$

The method declaration of the Service Interfaces mUnitServiceIf and pUnitServiceIf can be found in Appendix A.

4.8.2 Attribute Declaration

An attribute declaration contains two types of structural information: the attribute's name as well as the attribute's type. Thus, the following two relations declare an attribute.

$$\text{AttributeName} =_{def} \text{AttributeDeclaration} \rightarrow \text{String} \tag{4.58}$$

$$\text{AttributeType} =_{def} \text{AttributeDeclaration} \rightarrow \text{Type} \tag{4.59}$$

If we look at attribute triageClass_C, which occurred as a parameter in method setTriageClass_C in the previous section, this looks as follows:

$$\begin{aligned} \text{AttributeName}(\text{triageClass}_C) &= \text{"triageClass"} \\ \text{AttributeType}(\text{triageClass}_C) &= \text{TriageClass} \end{aligned} \tag{4.60}$$

The specification of those attributes declared in Service Interfaces mUnitServiceIf and pUnitServiceIf can be found in Appendix A. We depicted it in Figure 4.8 in addition.

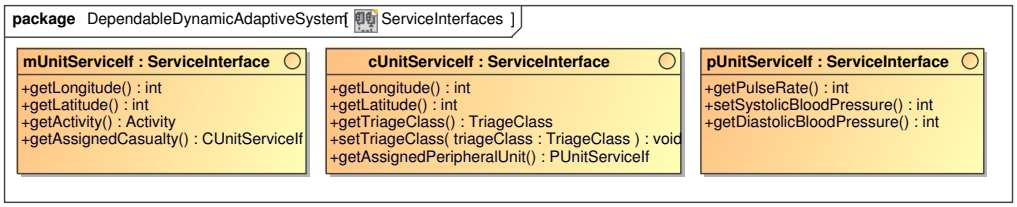


Figure 4.8: Specification of Service Interfaces in our Application Example.

4.9 Syntactical Compatibility

Now, that we specified the structure of Dependable Services as well as the structure of Dependable Service References and also the structure of methods and attributes within our system model, we can describe, whether a Dependable Service is syntactically compatible with a Dependable Service Reference. Therefore, we specify a $\simeq_{\text{Syntactical}}$ relation between Dependable Service Reference and Dependable Service.

$$\begin{aligned} \simeq_{\text{Syntactical}} =_{def}\ & \text{DependableServiceReference} \times \text{DependableService} \rightarrow \\ & \text{Boolean} \ : \text{ref} \simeq_{\text{Syntactical}} \text{serv} \Rightarrow \\ & \text{RefersTo}(\text{ref}) \simeq_{\text{InterfaceSyntactical}} \text{Implements}(\text{serv}) \\ & \forall \text{ref} \in \text{DependableServiceReference}, \\ & \forall \text{serv} \in \text{ServiceInterface} \end{aligned} \tag{4.61}$$

Having this specification, we need to refine the Uses relation from Equation 4.33, as a Dependable Service Reference may only use a Dependable Service, if

they are syntactically compatible. Thus, the Uses relation is defined as follows:

$$\begin{aligned}\mathsf{Uses}_S^T =_{def}\ & \mathsf{DependableServiceReference} \times T \times S \rightarrow \\ & \mathsf{DependableService} \cup \{\emptyset\}\ : \\ & \mathsf{Uses}_s^t(\mathsf{ref}) \neq \emptyset \Rightarrow \mathsf{ref} \simeq_{\mathsf{Syntactical}} \mathsf{Uses}_s^t(\mathsf{ref}) \wedge \\ & \exists \mathsf{c}_{\mathsf{prov}} \in \mathsf{ApplicationComponents}_s^t \mid \\ & \mathsf{c}_{\mathsf{prov}} \neq \mathsf{c}_{\mathsf{req}} \wedge \mathsf{Uses}_s^t(\mathsf{ref}) \in \mathsf{Provides}_s^t(\mathsf{Current}_s^t(\mathsf{c}_{\mathsf{prov}})) \\ & \forall \mathsf{ref} \in \mathsf{Declares}_s^t(\mathsf{Current}_s^t(\mathsf{c}_{\mathsf{req}})), \\ & \forall \mathsf{c}_{\mathsf{req}} \in \mathsf{ApplicationComponents}_s^t, \forall s \in S, \forall t \in T\end{aligned} \tag{4.62}$$

However, we still need to define, whether two Service Interfaces are syntactically compatible, as this notion of syntactical Compatibility between Service Interfaces is used within the definition of syntactical Compatibility between Dependable Service Reference and Dependable Service.

Definition of syntactical Compatibility of Service Interfaces.

Two Service Interfaces $\mathsf{if}_{\mathsf{req}}$ and $\mathsf{if}_{\mathsf{prov}}$ are syntactically compatible, if $\mathsf{if}_{\mathsf{prov}}$ contains a syntactically compatible method respectively attribute for each method and attribute of $\mathsf{if}_{\mathsf{req}}$.

Thus, a provided Service Interface can have more attributes or methods than a required Service Interface but may still be syntactically compatible to it. Consequently this relation is not commutative. The left hand side of the relation takes a Service Interface, which a Dependable Service Reference refers to, while the right hand side takes a Service Interface, which a Dependable Service implements.

$$\begin{aligned}\simeq_{\mathsf{InterfaceSyntactical}} =_{def}\ & \mathsf{ServiceInterface} \times \mathsf{ServiceInterface} \rightarrow \mathsf{Boolean}\ : \\ & \mathsf{if}_{\mathsf{req}} \simeq_{\mathsf{InterfaceSyntactical}} \mathsf{if}_{\mathsf{prov}} \Rightarrow \\ & ((\forall \mathsf{m}_{\mathsf{req}} \in \mathsf{Methods}(\mathsf{if}_{\mathsf{req}}) \exists \mathsf{m}_{\mathsf{prov}} \in \mathsf{Methods}(\mathsf{if}_{\mathsf{prov}}) \mid \\ & \mathsf{m}_{\mathsf{req}} \simeq_{\mathsf{MethodSyntactical}} \mathsf{m}_{\mathsf{prov}}) \wedge \\ & (\forall \mathsf{a}_{\mathsf{req}} \in \mathsf{Attributes}(\mathsf{if}_{\mathsf{req}}) \exists \mathsf{a}_{\mathsf{prov}} \in \mathsf{Attributes}(\mathsf{if}_{\mathsf{prov}}) \mid \\ & \mathsf{a}_{\mathsf{req}} \simeq_{\mathsf{AttributeSyntactical}} \mathsf{a}_{\mathsf{prov}}) \\ & \forall \mathsf{if}_{\mathsf{req}}, \mathsf{if}_{\mathsf{prov}} \in \mathsf{ServiceInterface}\end{aligned} \tag{4.63}$$

Methods are syntactical compatible, if they have the same name and their parameters and return types are syntactically compatible.

Since we want to be able to talk about syntactical Compatibility of Service Interface instances, we need to define the syntactical Compatibility on the lowest level of abstraction between two method declarations. We call this relation $\simeq_{\mathsf{MethodSyntactical}}$.

This relation between two methods is *not* commutative meaning that $a \simeq_{\mathsf{MethodSyntactical}} b \not\Rightarrow b \simeq_{\mathsf{MethodSyntactical}} a$. This is caused by the fact, that return types or parameters of a required and a provided method must not be of the same type – they may be in an inheritance relation.

For the return type of methods, we state that the return type of a required method m_{req} needs to be the same or a supertype of the return type of a provided method m_{prov}[8]. This is denoted by $\text{ReturnType}(m_{req}) \geq \text{ReturnType}(m_{prov})$ in the following.

$$
\begin{aligned}
\simeq_{\text{MethodSyntactical}} =_{def}\ & \text{MethodDeclaration} \times \text{MethodDeclaration} \rightarrow \\
& \text{Boolean} \ : m_{req} \simeq_{\text{MethodSyntactical}} m_{prov} \Rightarrow \\
& \text{MethodName}(m_{req}) = \text{MethodName}(m_{prov}) \wedge \\
& \text{ReturnType}(m_{req}) \geq \text{ReturnType}(m_{prov}) \wedge \\
& \exists p_{prov} \in \text{Parameters}(m_{prov}) | \\
& p_{req} \simeq_{\text{ParameterSyntactical}} p_{prov} \wedge \\
& |\text{Parameters}(m_{req})| = |\text{Parameters}(m_{prov})| \\
& \forall p_{req} \in \text{Parameters}(m_{req}), \\
& \forall m_{req}, m_{prov} \in \text{MethodDeclaration}
\end{aligned}
\tag{4.64}
$$

Parameters are syntactical compatible, if they have the same name and their type is syntactical compatible.

For parameters of methods, we need parameters with the same name, which are of the same type or of a subtype compared to the parameters of the required method: If we require a method with a parameter named p of type T, we could use each method, taking a parameter named p of type T or a supertype of T. The relation $\simeq_{\text{ParameterSyntactical}}$, therefore, is defined as follows:

$$
\begin{aligned}
\simeq_{\text{ParameterSyntactical}} =_{def}\ & \text{AttributeDeclaration} \times \text{AttributeDeclaration} \rightarrow \\
& \text{Boolean} \ : p_{req} \simeq_{\text{ParameterSyntactical}} p_{prov} \Rightarrow \\
& \text{AttributeName}(p_{req}) = \text{AttributeName}(p_{prov}) \wedge \\
& \text{AttributeType}(p_{req}) \leq \text{AttributeType}(p_{prov}) \\
& \forall p_{req}, p_{prov} \in \text{AttributeDeclaration}
\end{aligned}
\tag{4.65}
$$

Given this definition of syntactical Compatibility, we need to refine our definition of a Service Interface's methods declaration from Equation 4.44. The relation $\simeq_{\text{MethodSyntactical}}$ allows us, to specify, that a Service Interface must not contain two methods, which are pairwise syntactical compatible, since this would mean, that these two methods have the same names, parameters and return types, which does not make sense in practice within a namespace.

[8] Assume that we have two types: Car and FastCar, where Car is a supertype of FastCar. If we require a method named m with a return type Car, we could use a method named m returning a FastCar, since we could use the return value as if it would be a car.

If we require a method named m with a return type FastCar, we could *not* use a method named m returning a Car, since the return value does not necessarily have the same properties as a FastCar. Therefore, only methods returning the same or a subtype of the returned type defined in a required method are syntactically compatible.

The revised definition of the Methods relation, therefore, looks as follows.

$$\begin{aligned}\text{Methods} =_{def} \text{ServiceInterface} \rightarrow \mathcal{P}(\text{MethodDeclaration}) : \\ ((\text{m}_1 \simeq_{\text{MethodSyntactical}} \text{m}_2) \wedge ((\text{m}_2 \simeq_{\text{MethodSyntactical}} \text{m}_1)) \Rightarrow \\ \text{m}_1 = \text{m}_2 \; \forall \text{m}_1, \text{m}_2 \in \text{Methods}(\text{i}), \forall \text{i} \in \text{ServiceInterface}\end{aligned} \tag{4.66}$$

Since we want to be able to talk about syntactical Compatibility of Service Interfaces, we need to define the syntactical Compatibility not only between method declarations but between attribute declarations as well. We call this relation $\simeq_{\text{AttributeSyntactical}}$.

Attributes are syntactical compatible, if they have the same name, and the *same* type.

In contrast to syntactical Compatibility between parameters of a method, compatible attributes *must* be of the same type, as Components requiring an attribute, may read its value ($\text{AttributeType}(\text{a}_{\text{req}}) \geq \text{AttributeType}(\text{a}_{\text{prov}})$) or may set it to a new value ($\text{AttributeType}(\text{a}_{\text{req}}) \leq \text{AttributeType}(\text{a}_{\text{prov}})$).

The relation, therefore, is defined as follows.

$$\begin{aligned}\simeq_{\text{AttributeSyntactical}} =_{def} \text{AttributeDeclaration} \times \text{AttributeDeclaration} \rightarrow \\ \text{Boolean} \; : \text{a}_{\text{req}} \simeq_{\text{AttributeSyntactical}} \text{a}_{\text{prov}} \Rightarrow \\ \text{AttributeName}(\text{a}_{\text{req}}) = \text{AttributeName}(\text{a}_{\text{prov}}) \wedge \\ \text{AttributeType}(\text{a}_{\text{req}}) = \text{AttributeType}(\text{a}_{\text{prov}}) \\ \forall \text{a}_{\text{req}}, \text{a}_{\text{prov}} \in \text{AttributeDeclaration}\end{aligned} \tag{4.67}$$

Given this definition of syntactical Compatibility, we need to extend the definition of a Service Interface's attributes declaration from Equation 4.45. The relation $\simeq_{\text{AttributeSyntactical}}$ allows us, to specify, that no Service Interface may contain two attributes, which are syntactical compatible, since this would mean, that these two attributes have same names and types, which does not make sense in practice within a namespace.

The revised definition of the Attributes relation, therefore, looks as follows:

$$\begin{aligned}\text{Attributes} =_{def} \text{ServiceInterface} \rightarrow \mathcal{P}(\text{AttributeDeclaration}) : \\ \text{a}_1 \simeq_{\text{AttributeSyntactical}} \text{a}_2 \Rightarrow \text{a}_1 = \text{a}_2 \\ \forall \text{a}_1, \text{a}_2 \in \text{Methods}(\text{i}), \forall \text{i} \in \text{ServiceInterface}\end{aligned} \tag{4.68}$$

In our example we realize, that Dependable Service Reference $\text{pUnitReference}_{\text{German}}$ defined by the German C-Unit's best Dependable Component Configuration ($\text{cConfiguration2}_{\text{German}}$) is syntactically compatible with Dependable Service $\text{pUnitService}_{\text{Dutch}}$ provided by the Dutch P-Unit as they refer to a Service Interface containing the same methods and attributes.

Summed up, when considering syntactical Compatibility between Dependable Service References and Dependable Services of the German Dependable Dynamic

Adaptive Components we realize the following situation in our application example:

$$\begin{aligned} \text{cUnitReference}_{\text{German}} &\simeq_{\text{Syntactical}} \text{cUnitService}_{\text{German}} \\ \text{pUnitReference}_{\text{German}} &\simeq_{\text{Syntactical}} \text{pUnitService}_{\text{German}} \end{aligned} \tag{4.69}$$

Syntactical Compatibility between all Dependable Services and Dependable Service References from our application example is depicted in Figure 4.9. For reasons of clarity, we left out the surrounding Dependable Dynamic Adaptive Components and Dependable Component Configurations in this figure.

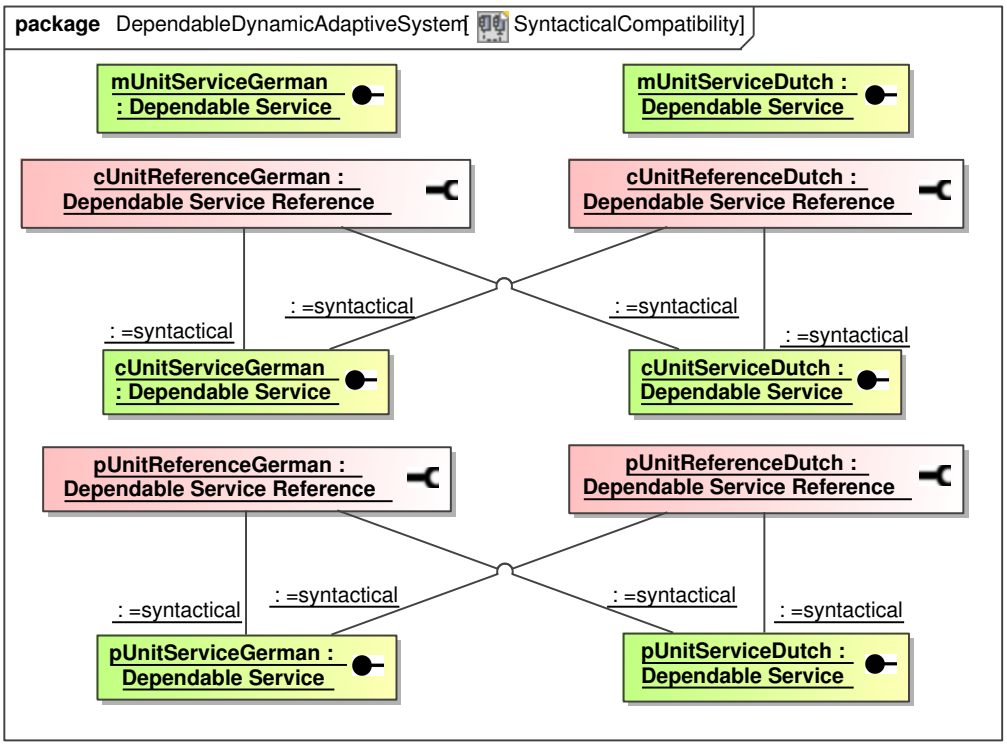


Figure 4.9: Syntactical Compatibility in our Application Example.

Our previously presented system model supports dynamic properties of Dependable Dynamic Adaptive Systems. This is achieved by considering different Dependability Checkpoints from set T within the relations of our system model. As motivated when introducing T in section 4.2, $t \in T$ is only increased, if a structural change or a behavioral change threatening Dependability has occurred.

We will investigate in the following, which Dependability-relevant structural changes can occur according to our system model. We call these changes *struc-*

tural reconfiguration triggers, as each of them may lead to a reconfiguration of the Dependable Dynamic Adaptive System.

4.10 Structural Reconfiguration Triggers

Structural reconfiguration triggers indicate a structural change, which threatens Dependability.

Dependable Dynamic Adaptive Systems are expected to reconfigure themselves at runtime. Therefore, we need to understand, which changes within the structure of a system respectively its Components may trigger such a reconfiguration. To get an idea of these triggers we will investigate a system $s \in S$ at each level of our system model as introduced before.

Each time, where a reconfiguration may occur due to one of these triggers we increase the Dependability Checkpoint by 1 – thus, $t_{new} = t_{latest} + 1$ where $t_{latest} \in T$ is the latest Dependability Checkpoint before this reconfiguration trigger has been active.

A Component enters or leaves.

First of all, a Dependable Dynamic Adaptive Component may enter or leave a Dependable Dynamic Adaptive System. This means, that set ApplicationComponents has changed. We call it AppComponentsChange and define it formally by the following trigger.

$$\text{AppComponentsChange}_S^T =_{def} T \times S \rightarrow \text{boolean} \tag{4.70}$$

$$\begin{aligned} \text{AppComponentsChange}_s^{t_{new}+1} := & \text{ApplicationComponents}_s^{t_{new}+1} \neq \\ & \text{ApplicationComponents}_s^{t_{new}} \\ & \forall t_{new} \in T, \forall s \in S \end{aligned} \tag{4.71}$$

The *set* of offered Configurations of a Component changes.

A second reconfiguration trigger is, that a Dependable Dynamic Adaptive Component changes the set of offered Dependable Component Configurations resulting in a newly offered Dependable Component Configuration or in a Dependable Component Configuration, which is not offered anymore. This means, that the Contains relation of a Component changes, which is expressed by the trigger ConfigurationChange.

$$\text{ConfigurationChange}_S^T =_{def} T \times S \rightarrow \text{boolean} \tag{4.72}$$

$$\begin{aligned} \text{ConfigurationChange}_s^{t_{new}+1} := & \exists c \in (\text{ApplicationComponents}_s^{t_{new}+1} \cap \\ & \text{ApplicationComponents}_s^{t_{new}}) \mid \\ & \text{Contains}_s^{t_{new}+1}(\mathsf{c}) \neq \text{Contains}_s^{t_{new}}(\mathsf{c}) \\ & \forall t_{new} \in T, \forall s \in S \end{aligned} \tag{4.73}$$

Another reconfiguration trigger related to Dependable Component Configurations is a change in the order of Dependable Component Configurations. This is expressed by trigger ConfigOrderChange, which is defined as follows.

$$\mathsf{ConfigOrderChange}_S^T =_{def} T \times S \rightarrow \mathsf{boolean} \tag{4.74}$$

$$\begin{aligned}\mathsf{ConfigOrderChange}_s^{t_{new}+1} :=& \exists \mathsf{c} \in (\mathsf{ApplicationComponents}_s^{t_{new}+1} \cap \\ & \mathsf{ApplicationComponents}_s^{t_{new}}) \wedge \exists \mathsf{conf}_1, \mathsf{conf}_2 \in \\ & (\mathsf{Contains}_s^{t_{new}+1}(\mathsf{c}) \cap \mathsf{Contains}_s^{t_{new}}(\mathsf{c}))| \\ & \mathsf{conf}_1 \neq \mathsf{conf}_2 \wedge \\ & \mathsf{conf}_1 \geq_s^{t_{new}+1} \mathsf{conf}_2 \wedge \mathsf{conf}_2 \geq_s^{t_{new}} \mathsf{conf}_1 \\ & \forall t_{new} \in T, \forall s \in S\end{aligned} \tag{4.75}$$

The *order* of offered Configurations of a Component changes.

The Current relation is no trigger for an upcoming reconfiguration. Instead a change in this relation means, that a reconfiguration of the Dependable Dynamic Adaptive System has occurred: a Dependable Component Configuration has been activated by the Dependable System Infrastructure – therefore, it represents a reconfiguration effect, an *adaptation*.

The Current Configuration changes (= Adaptation).

As we want to describe effects of reconfiguration as well, we will consider changes in the Current relation as reconfiguration triggers in the following. Thus, we define a trigger CurrentConfigurationChange, which is true, if the Dependable Component Configuration of a Dependable Dynamic Adaptive Component has changed.

$$\mathsf{CurrentConfigurationChange}_S^T =_{def} T \times S \rightarrow \mathsf{boolean} \tag{4.76}$$

$$\begin{aligned}\mathsf{CurrentConfigurationChange}_s^{t_{new}+1} :=& \exists \mathsf{c} \in (\mathsf{ApplicationComponents}_s^{t_{new}+1} | \\ & \mathsf{Current}_s^{t_{new}+1}(\mathsf{c}) \neq \mathsf{Current}_s^{t_{new}}(\mathsf{c}) \\ & \forall t_{new} \in T, \forall s \in S\end{aligned} \tag{4.77}$$

The set of provided Services respectively declared Service References of a Configuration changes.

Considering the internal structure of a Dependable Component Configuration, we can identify two changes, which trigger a reconfiguration.

1. A change in the set of provided Dependable Services, expressed by trigger ServiceChange.

2. A change in the set of declared Dependable Service References, expressed by trigger ReferenceChange.

These triggers are specified as follows.

$$\text{ServiceChange}_S^T =_{def} T \times S \rightarrow \text{boolean} \tag{4.78}$$

$$\begin{aligned}\text{ServiceChange}_s^{t_{new}+1} :=& \exists \text{c} \in (\text{ApplicationComponents}_s^{t_{new}+1} \cap \\ & \text{ApplicationComponents}_s^{t_{new}}) \wedge \\ & \exists \text{conf} \in (\text{Contains}_s^{t_{new}+1}(\text{c}) \cap \text{Contains}_s^{t_{new}}(\text{c}))| \\ & \text{Provides}_s^{t_{new}+1}(\text{conf}) \neq \text{Provides}_s^{t_{new}}(\text{conf}) \\ & \forall t_{new} \in T, \forall s \in S\end{aligned} \tag{4.79}$$

$$\text{ReferenceChange}_S^T =_{def} T \times S \rightarrow \text{boolean} \tag{4.80}$$

$$\begin{aligned}\text{ReferenceChange}_s^{t_{new}+1} :=& \exists \text{c} \in (\text{ApplicationComponents}_s^{t_{new}+1} \cap \\ & \text{ApplicationComponents}_s^{t_{new}}) \wedge \\ & \exists \text{conf} \in (\text{Contains}_s^{t_{new}+1}(\text{c}) \cap \text{Contains}_s^{t_{new}}(\text{c}))| \\ & \text{Declares}_s^{t_{new}+1}(\text{conf}) \neq \text{Declares}_s^{t_{new}}(\text{conf}) \\ & \forall t_{new} \in T, \forall s \in S\end{aligned} \tag{4.81}$$

Change of a Used Service.

Like changes in the Current relation, changes in the Uses relation are no trigger for reconfiguration. Instead a change in this relation means, that a reconfiguration of a Dependable Dynamic Adaptive System has occurred as a Dependable Service Reference now points to a different Dependable Service than before – therefore, it represents another reconfiguration effect, an *adaptation*. Adaptation here means, that a Service Binding has changed. We specify these changes in the Uses relation within a reconfiguration trigger BindingChange as follows.

$$\text{BindingChange}_S^T =_{def} T \times S \rightarrow \text{boolean} \tag{4.82}$$

$$\begin{aligned}\mathsf{BindingChange}_s^{t_{new}+1} :=& \exists \mathsf{c} \in (\mathsf{ApplicationComponents}_s^{t_{new}+1}) \wedge \\ & \exists \mathsf{conf} \in (\mathsf{Contains}_s^{t_{new}+1}(\mathsf{c})) \wedge \\ & \exists \mathsf{ref} \in (\mathsf{Declares}_s^{t_{new}+1}(\mathsf{conf}))| \\ & \mathsf{Uses}_s^{t_{new}+1}(\mathsf{ref}) \neq \mathsf{Uses}_s^{t_{new}}(\mathsf{ref}) \\ & \forall t_{new} \in T, \forall s \in S\end{aligned} \tag{4.83}$$

The type information in our system model is no reconfiguration trigger, as it does not change during runtime – instead it is static by definition. Thus, types and their related syntactical Compatibility cannot trigger a reconfiguration and, therefore, are not considered here.

Summed up, we identified two different types of structural reconfiguration triggers: one type enabling us, when it may be necessary to perform a reconfiguration and another one enabling us to reason about effects of a reconfiguration. In our formal model the first type is unified in a boolean relation StructuralChange as follows.

$$\mathsf{StructuralChange}_S^T =_{def} T \times S \rightarrow \mathsf{boolean} \tag{4.84}$$

$$\begin{aligned}\mathsf{StructuralChange}_s^{t_{new}} :=& \mathsf{AppComponentsChange}_s^{t_{new}} \vee \\ & \mathsf{ConfigurationChange}_s^{t_{new}} \vee \\ & \mathsf{ConfigOrderChange}_s^{t_{new}} \vee \\ & \mathsf{ServiceChange}_s^{t_{new}} \vee \mathsf{ReferenceChange}_s^{t_{new}} \\ & \forall t_{new} \in T, \forall s \in S\end{aligned} \tag{4.85}$$

Looking back at our application example, we can only recognize AppComponentsChange triggers, as we only face entering or leaving Components and do not consider changes to the internal structure of Dependable Dynamic Adaptive Components in our application example.

The second type of reconfiguration triggers is unified in a boolean relation StructuralReconfiguration as follows.

$$\mathsf{StructuralReconfiguration}_S^T =_{def} T \times S \rightarrow \mathsf{boolean} \tag{4.86}$$

$$\begin{aligned}\mathsf{StructuralReconfiguration}_s^{t_{new}} :=& \mathsf{CurrentConfigurationChange}_s^{t_{new}} \vee \\ & \mathsf{BindingChange}_s^{t_{new}} \\ & \forall t_{new} \in T, \forall s \in S\end{aligned} \tag{4.87}$$

A Dependable System Infrastructure is expected to react to these reconfiguration triggers and reconfigure a Dependable Dynamic Adaptive System immediately. Thus, we assume, that no Dependability Checkpoints are between recognizing a reconfiguration trigger and reacting to it by reconfiguration. Instead we assume, that these two steps are performed at the same Dependability Checkpoint.

Structural reconfiguration → structural change.

Considering this, we can state, that each reconfiguration requires, that a reconfiguration trigger is true. However, not each active reconfiguration trigger causes a reconfiguration. For example, a Dependable Dynamic Adaptive Component may enter a Dependable Dynamic Adaptive System, which is not bound to any other Component in this system due to syntactical incompatibilities.

This is covered by the following implication which is expected to be true at each Dependability Checkpoint.

$$\mathsf{StructuralReconfiguration}_s^{t_{new}} \Rightarrow \mathsf{StructuralChange}_s^{t_{new}} \quad \forall t_{new} \in T, \forall s \in S \tag{4.88}$$

4.11 Summary

In this section you can find the formal definition of the structural model concentrated at a single place. Thus, we will repeat the formulas from the previous sections top-down. Whenever definitions were revised, you will find the latest definition in this summary.

$$\mathsf{ApplicationComponents}_S^T =_{def} T \times S \rightarrow \mathcal{P}(\mathsf{DependableDynamicAdaptiveComponent}) \tag{4.89}$$

$$\begin{aligned}\mathsf{Contains}_S^T =_{def}\ & \mathsf{DependableDynamicAdaptiveComponent} \times T \times S \rightarrow \\ & \mathcal{P}(\mathsf{DependableComponentConfiguration}) \ : \\ & \mathsf{configuration} \in \mathsf{Contains}_s^t(\mathsf{component}_1) \wedge \\ & \mathsf{configuration} \in \mathsf{Contains}_s^t(\mathsf{component}_2) \Rightarrow \\ & \mathsf{component}_1 = \mathsf{component}_2 \\ & \forall \mathsf{configuration} \in \mathsf{DependableComponentConfiguration}, \\ & \forall s \in S, \forall t \in T, \forall \mathsf{component}_1, \mathsf{component}_2 \in \\ & \mathsf{ApplicationComponents}_s^t\end{aligned} \tag{4.90}$$

$$\begin{aligned}\geq_S^T =_{def}\ & \mathsf{DependableComponentConfiguration} \times \\ & \mathsf{DependableComponentConfiguration} \times T \times S \rightarrow \mathsf{Boolean} \ : \\ & ((\mathsf{c}_1 \geq_s^t \mathsf{c}_2) \wedge (\mathsf{c}_2 \geq_s^t \mathsf{c}_3) \Rightarrow \mathsf{c}_1 \geq_s^t \mathsf{c}_3) \ \wedge (\mathsf{c}_1 \geq_s^t \mathsf{c}_1) \wedge \\ & (((\mathsf{c}_1 \geq_s^t \mathsf{c}_2) \wedge (\mathsf{c}_2 \geq_s^t \mathsf{c}_1)) \Leftrightarrow (\mathsf{c}_1 = \mathsf{c}_2)) \\ & \forall \mathsf{c} \in \mathsf{ApplicationComponents}_s^t, \\ & \forall \mathsf{c}_1, \mathsf{c}_2, \mathsf{c}_3 \in \mathsf{Contains}_s^t(\mathsf{c}), \forall s \in S, \forall t \in T\end{aligned} \tag{4.91}$$

$$\begin{aligned}\mathsf{Current}_S^T =_{def}\ & \mathsf{DependableDynamicAdaptiveComponent} \times T \times S \rightarrow \\ & \mathsf{DependableComponentConfiguration} \cup \{\emptyset\}\ : \\ & (\mathsf{Current}_s^t(\mathsf{c}) \in \mathsf{Contains}_s^t(\mathsf{c}) \cup \{\emptyset\}) \wedge \\ & (\mathsf{Current}_s^t(\mathsf{c}) \neq \emptyset\ \Rightarrow \mathsf{Uses}_s^t(\mathsf{ref}) \neq \emptyset) \forall s \in S, \forall t \in T, \\ & \forall \mathsf{c} \in \mathsf{ApplicationComponents}_s^t, \forall \mathsf{ref} \in \mathsf{Declares}_s^t(\mathsf{Current}(\mathsf{c}))\end{aligned} \tag{4.92}$$

$$\begin{aligned}\mathsf{Provides}_S^T =_{def}\ & \mathsf{DependableComponentConfiguration} \times T \times S \rightarrow \\ & \mathcal{P}(\mathsf{DependableService})\ : \\ & \mathsf{c}_1 \neq \mathsf{c}_2 \Rightarrow \mathsf{Provides}_s^t(\mathsf{conf}_1) \cap \mathsf{Provides}_s^t(\mathsf{conf}_2) = \emptyset \\ & \forall \mathsf{c}_1, \mathsf{c}_2 \in \mathsf{DependableDynamicAdaptiveComponent}, \\ & \forall \mathsf{conf}_1 \in \mathsf{Contains}_s^t(\mathsf{c}_1), \forall \mathsf{conf}_2 \in \mathsf{Contains}_s^t(\mathsf{c}_2), \\ & \forall s \in S, \forall t \in T\end{aligned} \tag{4.93}$$

$$\begin{aligned}\mathsf{Declares}_S^T =_{def}\ & \mathsf{DependableComponentConfiguration} \times T \times S \rightarrow \\ & \mathcal{P}(\mathsf{DependableServiceReference})\ : \\ & \mathsf{c}_1 \neq \mathsf{c}_2 \wedge \mathsf{conf}_1 \in \mathsf{Contains}_s^t(\mathsf{c}_1) \wedge \forall \mathsf{conf}_2 \in \mathsf{Contains}_s^t(\mathsf{c}_2) \Rightarrow \\ & \mathsf{Declares}_s^t(\mathsf{conf}_1) \cap \mathsf{Declares}_s^t(\mathsf{conf}_2) = \emptyset \\ & \forall \mathsf{c}_1, \mathsf{c}_2 \in \mathsf{DependableDynamicAdaptiveComponent}, \\ & \forall s \in S, \forall t \in T\end{aligned} \tag{4.94}$$

$$\begin{aligned}\mathsf{Uses}_S^T =_{def}\ & \mathsf{DependableServiceReference} \times T \times S \rightarrow \\ & \mathsf{DependableService} \cup \{\emptyset\}\ : \\ & \mathsf{Uses}_s^t(\mathsf{ref}) \neq \emptyset \Rightarrow \mathsf{ref} \simeq_{\mathsf{Syntactical}} \mathsf{Uses}_s^t(\mathsf{ref}) \wedge \\ & \exists \mathsf{c}_{\mathsf{prov}} \in \mathsf{ApplicationComponents}_s^t\ \mid \\ & \mathsf{c}_{\mathsf{prov}} \neq \mathsf{c}_{\mathsf{req}} \wedge \mathsf{Uses}_s^t(\mathsf{ref}) \in \mathsf{Provides}_s^t(\mathsf{Current}_s^t(\mathsf{c}_{\mathsf{prov}})) \\ & \forall \mathsf{ref} \in \mathsf{Declares}_s^t(\mathsf{Current}_s^t(\mathsf{c}_{\mathsf{req}})), \\ & \forall \mathsf{c}_{\mathsf{req}} \in \mathsf{ApplicationComponents}_s^t, \forall s \in S, \forall t \in T\end{aligned} \tag{4.95}$$

$$\mathsf{Implements} =_{def} \mathsf{DependableService} \rightarrow \mathsf{ServiceInterface} \tag{4.96}$$

$$\mathsf{RefersTo} =_{def} \mathsf{DependableServiceReference} \rightarrow \mathsf{ServiceInterface} \tag{4.97}$$

$$\begin{aligned}\text{Methods} =_{def} \text{ServiceInterface} \rightarrow \mathcal{P}(\text{MethodDeclaration}) :\\ ((\text{m}_1 \simeq_{\text{MethodSyntactical}} \text{m}_2) \wedge ((\text{m}_2 \simeq_{\text{MethodSyntactical}} \text{m}_1)) \Rightarrow \\ \text{m}_1 = \text{m}_2 \; \forall \text{m}_1, \text{m}_2 \in \text{Methods}(\text{i}), \forall \text{i} \in \text{ServiceInterface}\end{aligned} \tag{4.98}$$

$$\begin{aligned}\text{Attributes} =_{def} \text{ServiceInterface} \rightarrow \mathcal{P}(\text{AttributeDeclaration}) :\\ \text{a}_1 \simeq_{\text{AttributeSyntactical}} \text{a}_2 \Rightarrow \text{a}_1 = \text{a}_2 \\ \forall \text{a}_1, \text{a}_2 \in \text{Methods}(\text{i}), \forall \text{i} \in \text{ServiceInterface}\end{aligned} \tag{4.99}$$

$$\text{MethodName} =_{def} \text{MethodDeclaration} \rightarrow \text{String} \tag{4.100}$$

$$\text{ReturnType} =_{def} \text{MethodDeclaration} \rightarrow \text{Type} \tag{4.101}$$

$$\begin{aligned}\text{Parameters} =_{def} \text{MethodDeclaration} \rightarrow \mathcal{P}(\text{AttributeDeclaration}) :\\ \text{Parameters}(\text{m}) \cap \text{Attributes}(\text{c}) = \emptyset \\ \text{m}_1 \neq \text{m}_2 \Rightarrow \text{Parameters}(\text{m}_1) \cap \text{Parameters}(\text{m}_2) = \emptyset \\ \forall \text{m}, \text{m}_1, \text{m}_2 \in \text{MethodDeclaration}, \\ \forall \text{c} \in \text{DependableDynamicAdaptiveComponent}\end{aligned} \tag{4.102}$$

$$\text{AttributeName} =_{def} \text{AttributeDeclaration} \rightarrow \text{String} \tag{4.103}$$

$$\text{AttributeType} =_{def} \text{AttributeDeclaration} \rightarrow \text{Type} \tag{4.104}$$

$$\begin{aligned}\simeq_{\text{Syntactical}} =_{def} \text{DependableServiceReference} \times \text{DependableService} \rightarrow \\ \text{Boolean} \; : \text{ref} \simeq_{\text{Syntactical}} \text{serv} \Rightarrow \\ \text{RefersTo}(\text{ref}) \simeq_{\text{InterfaceSyntactical}} \text{Implements}(\text{serv}) \\ \forall \text{ref} \in \text{DependableServiceReference}, \\ \forall \text{serv} \in \text{ServiceInterface}\end{aligned} \tag{4.105}$$

$$\begin{aligned}\simeq_{\text{InterfaceSyntactical}} =_{def} \text{ServiceInterface} \times \text{ServiceInterface} \rightarrow \text{Boolean} \; : \\ \text{if}_{\text{req}} \simeq_{\text{InterfaceSyntactical}} \text{if}_{\text{prov}} \Rightarrow \\ ((\forall \text{m}_{\text{req}} \in \text{Methods}(\text{if}_{\text{req}}) \exists \text{m}_{\text{prov}} \in \text{Methods}(\text{if}_{\text{prov}}) | \\ \text{m}_{\text{req}} \simeq_{\text{MethodSyntactical}} \text{m}_{\text{prov}}) \wedge \\ (\forall \text{a}_{\text{req}} \in \text{Attributes}(\text{if}_{\text{req}}) \exists \text{a}_{\text{prov}} \in \text{Attributes}(\text{if}_{\text{prov}}) | \\ \text{a}_{\text{req}} \simeq_{\text{AttributeSyntactical}} \text{a}_{\text{prov}}) \\ \forall \text{if}_{\text{req}}, \text{if}_{\text{prov}} \in \text{ServiceInterface}\end{aligned} \tag{4.106}$$

$$\begin{aligned}\simeq_{\text{MethodSyntactical}} =_{def} & \text{MethodDeclaration} \times \text{MethodDeclaration} \rightarrow \\ & \text{Boolean} \ : \text{m}_{\text{req}} \simeq_{\text{MethodSyntactical}} \text{m}_{\text{prov}} \Rightarrow \\ & \text{MethodName}(\text{m}_{\text{req}}) = \text{MethodName}(\text{m}_{\text{prov}}) \wedge \\ & \text{ReturnType}(\text{m}_{\text{req}}) \geq \text{ReturnType}(\text{m}_{\text{prov}}) \wedge \\ & \exists \text{p}_{\text{prov}} \in \text{Parameters}(\text{m}_{\text{prov}}) | \\ & \text{p}_{\text{req}} \simeq_{\text{ParameterSyntactical}} \text{p}_{\text{prov}} \wedge \\ & |\text{Parameters}(\text{m}_{\text{req}})| = |\text{Parameters}(\text{m}_{\text{prov}})| \\ & \forall \text{p}_{\text{req}} \in \text{Parameters}(\text{m}_{\text{req}}), \\ & \forall \text{m}_{\text{req}}, \text{m}_{\text{prov}} \in \text{MethodDeclaration}\end{aligned} \tag{4.107}$$

$$\begin{aligned}\simeq_{\text{ParameterSyntactical}} =_{def} & \text{AttributeDeclaration} \times \text{AttributeDeclaration} \rightarrow \\ & \text{Boolean} \ : \text{p}_{\text{req}} \simeq_{\text{ParameterSyntactical}} \text{p}_{\text{prov}} \Rightarrow \\ & \text{AttributeName}(\text{p}_{\text{req}}) = \text{AttributeName}(\text{p}_{\text{prov}}) \wedge \\ & \text{AttributeType}(\text{p}_{\text{req}}) \leq \text{AttributeType}(\text{p}_{\text{prov}}) \\ & \forall \text{p}_{\text{req}}, \text{p}_{\text{prov}} \in \text{AttributeDeclaration}\end{aligned} \tag{4.108}$$

$$\begin{aligned}\simeq_{\text{AttributeSyntactical}} =_{def} & \text{AttributeDeclaration} \times \text{AttributeDeclaration} \rightarrow \\ & \text{Boolean} \ : \text{a}_{\text{req}} \simeq_{\text{AttributeSyntactical}} \text{a}_{\text{prov}} \Rightarrow \\ & \text{AttributeName}(\text{a}_{\text{req}}) = \text{AttributeName}(\text{a}_{\text{prov}}) \wedge \\ & \text{AttributeType}(\text{a}_{\text{req}}) = \text{AttributeType}(\text{a}_{\text{prov}}) \\ & \forall \text{a}_{\text{req}}, \text{a}_{\text{prov}} \in \text{AttributeDeclaration}\end{aligned} \tag{4.109}$$

Now that we have specified our structural model for Dependable Dynamic Adaptive Systems we need to add behavioral aspects to it. in order to achieve not only syntactical Compatibility but semantical Compatibility as well, when binding Dependable Dynamic Adaptive Systems. Thus, we will look at these behavioral aspects of our system model in the following.

It would seem that perfection is attained not when no more can be added, but when no more can be removed.

Antoine de Saint Exupéry

5

Behavioral Model for Dependable Dynamic Adaptive Systems

We already defined syntactical Compatibility within the previous chapter. However, this is not sufficient [VHT00] for Dependable Dynamic Adaptive Systems – we also need to consider the behavior of Dependable Dynamic Adaptive Components bound within these systems.

In this chapter we will, therefore, investigate the behavior of Dependable Dynamic Adaptive Systems. Thus, we will introduce behavioral aspects in our system model and use them to describe the behavior of the specific Dependable Dynamic Adaptive System from our application example.

In contrary to the introduction of structural aspects in Section 4, we will describe the behavior bottom-up ending at the system level. We describe it bottom-up, since our model defines semantical Compatibility between Dependable Services and Dependable Service References. All specifications on top-levels in our behavioral system model only contain aggregations of specification from the lower level.

Not the behavior itself is described, but state spaces of equivalent provided respectively expected behavior.

Our model does *not* describe the *behavior* of a system respectively its Components *itself*. Instead we describe only *classes of equivalent behavior* for its Components.

The idea behind these Behavior Equivalence Classes is, to describe state spaces, where a Component provides respectively requires an equivalent behavior. At runtime a Dependable System Infrastructure, therefore, is capable of monitoring the Behavior Equivalence Classes of bound Dependable Dynamic Adaptive Components and can execute Compliance Test Cases whenever such a Behavior Equiva-

lence Class changes.

A graphical view of our behavioral model introduced in the following is depicted in Figure 5.1 – this Figure helps you, to understand our system model. The structural parts of the model are depicted as well but with a lower saturation. This highlights our behavioral additions to the structural model.

The margins depict model elements from Figure 5.1.

We will use the colors from this Figure during the following sections when we describe the system from our application example using our system model. As a reminder, we will depict corresponding model elements from Figure 5.1 in the margins, whenever a model element is described in the following sections.

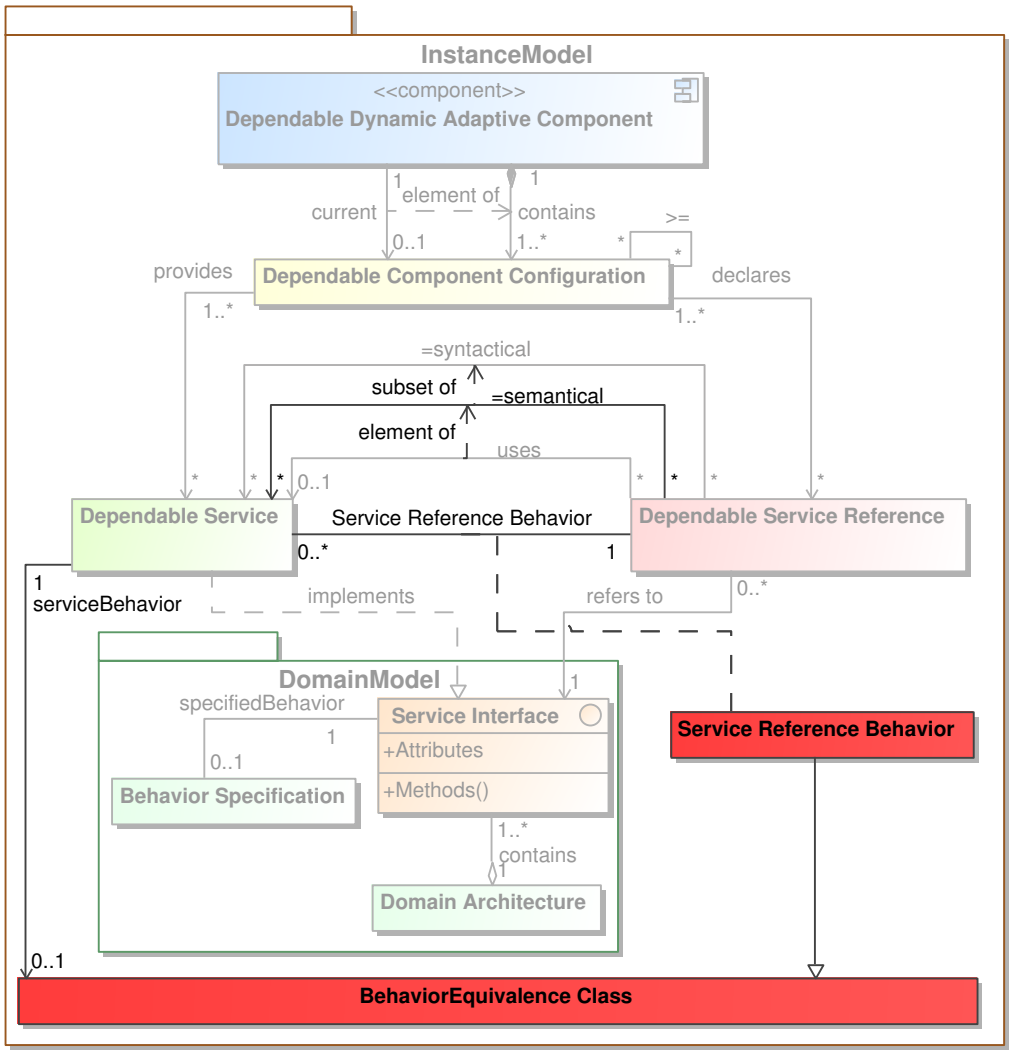


Figure 5.1: Graphical View of our Behavioral System Model.

5.1 Looking Back at the Application Example

Our model needs to be capable of describing the Emergency Assistance System introduced in section 3.

Considering system s_{ae} from our application example, we could investigate the behavior of this Dependable Dynamic Adaptive System at Dependability Checkpoints t_0 to $t_0 + 6$ which have been introduced in section 4.1. However, regarding our aim of establishing Dependable Dynamic Adaptive Systems, this situation is not very interesting.

Until $t_0 + 6$ semantical Compatibility could have been proven in advance at development time.

Those Components bound together at $t_0 + 6$ were developed by the same vendor. They may have been explicitly designed to interoperate with each other. The interaction, therefore, could have been easily tested respectively verified in advance by the Component vendor.

To demonstrate the benefit of our approach we will, therefore, consider the following situation: Later, at Dependability Checkpoint t_n ($t_n > t_0 + 6$) a Dutch medic equips a further casualty with a Dutch C-Unit. Thus, now a second Dutch C-Unit is present in our Dependable Dynamic Adaptive System. The structure and behavior of this second Dutch C-Unit does not differ from the first one. To distinguish it from the first one, we add the postfix *2*, whenever we consider this C-Unit using our system model.

The Dutch medic does not deploy a P-Unit at this casualty, as his situation is not critical yet. Most interesting regarding our approach are the following two Dependability Checkpoints $t_n + 1$ and $t_n + 2$ as depicted in Figure 5.2. They are defined as follows:

$t_n + 1$ and $t_n + 2$ denote Dependability Checkpoints, when a Dutch C-Unit is bound to a German P-Unit.

$t_n + 1$ denotes the Dependability Checkpoint, when this casualty is equipped with a German P-Unit to monitor his vital condition in the following. This results from the decision of a German medic treating this casualty, that his vital condition got critical and, therefore, live monitoring is indicated. The structure and behavior of this second German P-Unit does not differ from the first one. To distinguish it from the first one, we add the postfix *2*, whenever we consider this P-Unit using our system model.

$t_n + 2$ describes the situation, where the fingerclip of this second German P-Unit measuring the pulse rate of this casualty slips off. Thus, this casualty has a measured pulse rate of zero, while his blood pressure is still present.

$t_n + 2$: semantical incompatibility occurs.

In the following we will investigate, which additional basic sets are required in our formal model in order to capture behavioral aspects of Dependable Dynamic Adaptive Systems.

Figure 5.2: Dependability Checkpoints Interesting for Semantical Compatibility in our Example.

5.2 Basic Sets

All Behavior Equivalence Classes.

As for the structural model, we first of all need to define some basic sets, which we refer to, when talking about behavioral aspects of a Dependable Dynamic Adaptive System. On the behavioral side we only have the set *BehaviorEquivalenceClass* which contains all Behavior Equivalence Classes, we can think of.

Now that we defined the basic sets, we can start describing the behavior (respectively the classes of equivalent behavior) bottom-up in the following.

5.3 Dependable Service and Dependable Service Reference Behavior Class

Behavior Equivalence Classes imply equivalent behavior of a Service. Behavior Equivalence Classes are defined by component vendors at development time.

First of all we will investigate, how Dependable Services respectively Dependable Service References behave in a Dependable Dynamic Adaptive System.

A Dependable Service may behave differently depending on the internal state of a Dependable Dynamic Adaptive Component providing this service. Thus, we introduce the concept of Behavior Equivalence Classes. A Dependable Service is associated with a set of such Behavior Equivalence Classes.

At each point during runtime, a single Behavior Equivalence Class is active for each Dependable Service, depending on the state of the provider of this service. The meaning of a Behavior Equivalence Class for a Dependable Service is, that

the Dependable Service will behave equivalently as long as the active Behavior Equivalence Class does not change.

Behavior Equivalence Classes of a Dependable Service are defined by a Component vendor at development time of a Dependable Dynamic Adaptive Component providing the specific Dependable Service. The vendor can derive them from the control flow of the implementation by considering the different paths.

A starting point would be considering each path as a separate Behavior Equivalence Class. However, a Component vendor may need to classify several paths under a single Behavior Equivalence Class. For example, if different paths are taken depending on input passed to a Dependable Service, the Behavior Equivalence Class would be the same for all these paths.

Within our formal model, we assume, that the currently active Behavior Equivalence Class of a Dependable Service is given by a relation ServiceBehavior depending on the internal state of a providing Component[1]. It is defined as follows.

$$\text{ServiceBehavior}_S^T =_{def} \text{DependableService} \times T \times S \rightarrow \text{BehaviorEquivalenceClass} \tag{5.1}$$

Considering system s_{ae} from our application example, we need to investigate the behavior of the Dependable Service provided by a Dutch P-Unit and the behavior of a Dependable Service Reference declared by a Dutch C-Unit, which are bound at $t_0 + 6$. However, regarding our aim of establishing Dependable Dynamic Adaptive Systems, this situation is not very interesting, as the Components bound at $t_0 + 6$ were designed by the same vendor. Thus, they could have been designed to interoperate with each other. In our example the interaction could have been easily tested respectively verified in advance by the Dutch vendor of these two Components.

To demonstrate the benefit of our approach we will, therefore, consider $t_n + 1$ which is another Dependability Checkpoint, where a Dutch C-Unit is bound to a German P-Unit. This situation can occur within our example, when a German medic treats a casualty, which has been equipped with a Dutch C-Unit in advance.

If this German medic decides, that the vital condition of this casualty is critical, he may equip him with an additional P-Unit to monitor his vital condition in the following. In this case, we will face exactly this situation.

In our example, the vendor of the German P-Units may have defined, that these P-Units behave equivalently regardless of their internal state. He calls this Behavior Equivalence Class $\text{pUnitServiceUsualOperation}_{\text{German}}$. Thus, the Behavior Equivalence Class of the provided Dependable Service $\text{pUnitService}_{\text{German}}$ is

[1] The internal state is not considered in our formal system model. When implementing our system model, a mechanism is needed, which updates the Behavior Equivalence Class, whenever the internal state of a providing Component changes.

BehaviorEquivalence Class

$\text{pUnitServiceUsualOperation}_{\text{German}}$ at any time during system execution in our example system. Therefore, we will realize the following at $t_n + 1$.

$$\text{ServiceBehavior}_{s_{ae}}^{t_n+1}(\text{pUnitService}_{\text{German}}2) = \text{pUnitServiceUsualOperation}_{\text{German}} \tag{5.2}$$

A Service User expects equivalent behavior as long as the Behavior Equivalence Class does not change.

Next to Behavior Equivalence Classes of Dependable Services, we also apply this concept to Dependable Service References. This enables us to describe state spaces, where a Service User assumes equivalent behavior of a Service Provider.

For Dependable Service References Behavior Equivalence Classes are used to express, that a specific equivalent behavior is expected for the provider of the Dependable Service as long as the Behavior Equivalence Class does not change. This specific equivalent behavior needs to be specified by the Service User.

In our formal model this is done by the relation $\simeq_{\text{Semantical}}$[2] in Section 5.4.

A currently active Behavior Equivalence Class of a Dependable Service Reference is given by the relation ServiceReferenceBehavior. It is a relation, which assigns a Behavior Equivalence Class to a pair of Dependable Service and Dependable Service Reference.

Service Reference Behavior

This Behavior Equivalence Class depends on the internal state of a Dependable Dynamic Adaptive Component containing the given Dependable Service Reference as well as on an externally visible state[3] of the given syntactically compatible Dependable Service. The relation is, therefore, defined as follows.

$$\begin{aligned}\text{ServiceReferenceBehavior}_S^T =_{def}\ & \text{DependableServiceReference} \times \\ & \text{DependableService} \times T \times S \rightarrow \\ & \text{BehaviorEquivalenceClass} \cup \{\emptyset\}\ : \\ & \text{ServiceReferenceBehavior}_s^t(\text{ref}, \text{serv}) = \emptyset \Leftrightarrow \\ & \text{ref} \not\simeq_{\text{Syntactical}} \text{serv} \\ & \forall \text{ref} \in \text{DependableServiceReference}, \\ & \forall \text{serv} \in \text{DependableService}, \forall t \in T, \forall s \in S\end{aligned} \tag{5.3}$$

ServiceReferenceBehavior is defined for any pair of syntactically compatible instances of Dependable Service Reference and Dependable Service. This relation, therefore, enables a Dependable System Infrastructure to reason about a currently active ServiceReferenceBehavior of such a pair, even if they are not bound within the system at the moment.

[2] In our reference implementation of a Dependable System Infrastructure in Chapter 6 this relation is realized by runtime testing.

[3] The externally visible state is each part of state information, which can be queried by calling state-preserving methods or by reading attributes of a specific Dependable Service.

If we consider system s_{ae} from our application example, the vendor of the Dutch C-Unit might have specified, that he can think of two Behavior Equivalence Classes regarding a required P-Unit:

1. $\text{pUnitReferenceUsualOperation}_{\text{Dutch}}$ where this C-Unit would calculate the Triage Class of a casualty to T I, T II, T III, or T IV based on values provided by the P-Unit. In this case, the C-Unit expects the P-Unit to return values for pulse as well as blood pressure, which are in usual ranges (e.g. $0 <$ return value < 300).

2. $\text{pUnitReferenceDeadOperation}_{\text{Dutch}}$ where this C-Unit would calculate the Triage Class of a casualty to EX based on values provided by the P-Unit, since the pulse of the casualty is equal to zero. In this case, the C-Unit expects the P-Unit to return values for pulse as well as blood pressure, which are equal to zero.

If we assume, that a German P-Unit is attached to a casualty with a pulse of 70, and a blood pressure of (120, 80) at $t_n + 1$, this Dutch C-Unit would calculate the Triage Class to T I. Thus, it is in Behavior Equivalence Class $\text{pUnitReferenceUsualOperation}_{\text{Dutch}}$ at this Dependability Checkpoint. This is specified as follows.

$$\begin{aligned}&\text{ServiceReferenceBehavior}_{s_{ae}}^{t_n+1}(\text{pUnitReference}_{\text{Dutch}}2,\\ &\quad \text{pUnitService}_{\text{German}}2) = \text{pUnitReferenceUsualOperation}_{\text{Dutch}}\end{aligned} \tag{5.4}$$

In this Behavior Equivalence Class it expects pulse as well as blood pressure of the P-Unit to be between zero and 300 in order to be semantical compatible.

An excerpt of the situation at $t_n + 1$ showing only the second German P-Unit and the second Dutch C-Unit is depicted in Figure 5.3.

If we consider Dependability Checkpoint $t_n + 2$, where the fingerclip of the associated casualty slips off, the Dutch C-Unit would calculate the Triage Class to EX, as it does not consider blood pressure in addition. This is due to the implicit assumption made by the Dutch vendor, that the blood pressure *must* be equal to (0, 0) in this case as well.

Thus, the Behavior Equivalence Class of the pair of the second Dutch C-Unit and the second German P-Unit changes at $t_n + 2$ towards $\text{pUnitReferenceDeadOperation}_{\text{Dutch}}$. The C-Unit now expects the P-Unit to behave differently. Instead of values between zero and 300 it now expects pulse as well as blood pressure of the P-Unit to be equal to zero in order to be semantical compatible.

$$\begin{aligned}&\text{ServiceReferenceBehavior}_{s_{ae}}^{t_n+2}(\text{pUnitReference}_{\text{Dutch}}2,\\ &\quad \text{pUnitService}_{\text{German}}2) = \text{pUnitReferenceDeadOperation}_{\text{Dutch}}\end{aligned} \tag{5.5}$$

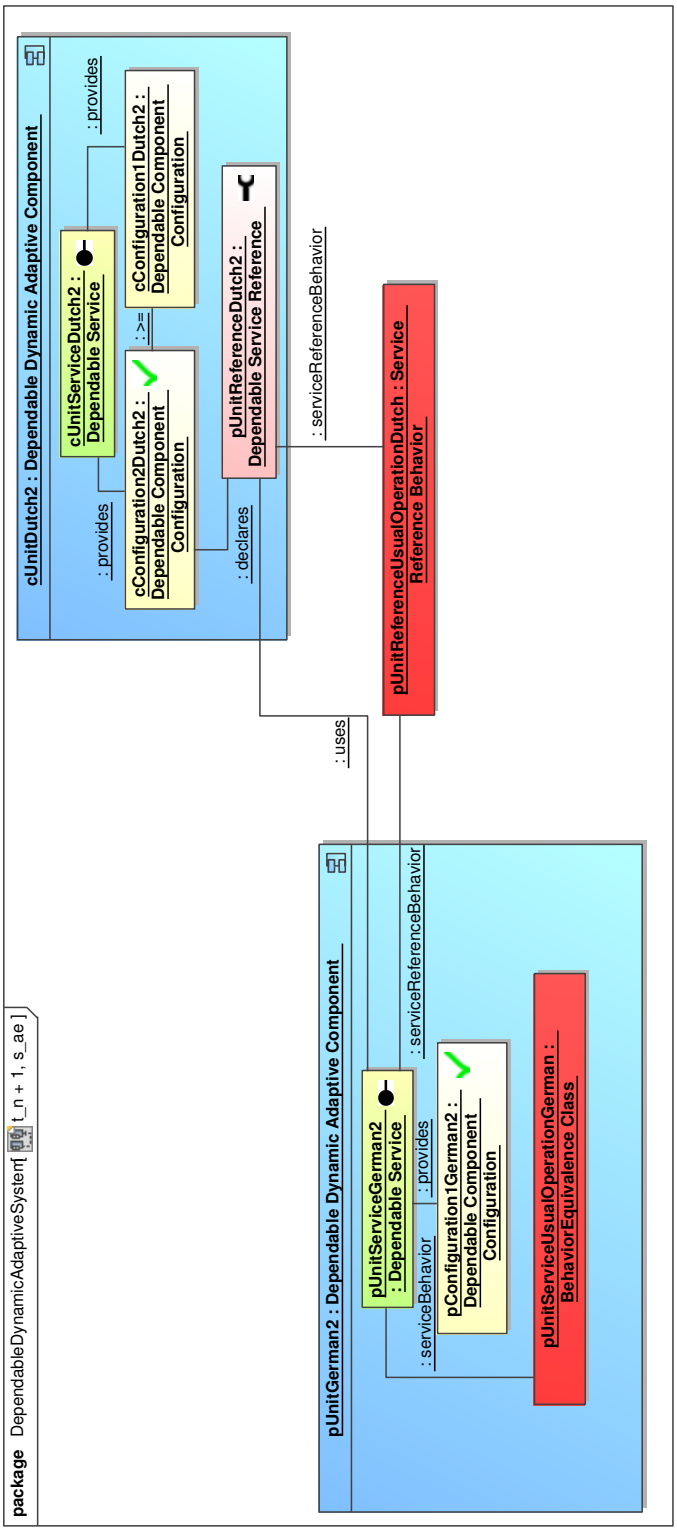


Figure 5.3: The Behavior Equivalence Classes of a Dependable Service and a Dependable Service Reference in Our Application Example at $t_n + 1$.

In the following we will analyze, how we can reason about semantical Compatibility at runtime by using our behavioral model defined before.

5.4 Semantical Compatibility

Now, that we considered classes of equivalent behavior within our model, we can describe, whether a Dependable Service is semantically compatible with a Dependable Service Reference. Therefore, we specify the $\simeq_{\text{Semantical}}$ relation between Dependable Service Reference and Dependable Service.

Important about this specification is, that it is expected to return the same value[4] for semantical Compatibility as long as the Behavior Equivalence Classes of the involved Dependable Service and Dependable Service Reference do not change. Thus, if a Dependable Service Reference ref has been semantical compatible with Dependable Service serv at Dependability Checkpoint t, they are semantical compatible at every other Dependability Checkpoint, when the same Behavior Equivalence Classes are active for ref and serv, as well. The same applies for semantical incompatibilities.

Semantical Compatibility of a binding is not expected to change, unless its Combined Behavior Equivalence Class changes.

This is reflected in the specification of $\simeq_{\text{Semantical}}$ as follows.

$$
\begin{aligned}
\simeq_{\text{Semantical}}{}^{T}_{S} =_{def}\ & \text{DependableServiceReference} \times \text{DependableService} \times \\
& T \times S \rightarrow \text{Boolean} \ : \\
& (\text{ServiceBehavior}_s^{t_1}(\text{serv}) = \text{ServiceBehavior}_s^{t_2}(\text{serv}) \wedge \\
& \text{ServiceReferenceBehavior}_s^{t_1}(\text{ref}, \text{serv}) = \\
& \text{ServiceReferenceBehavior}_s^{t_2}(\text{ref}, \text{serv})) \Rightarrow \\
& ((\text{ref} \simeq_{\text{Semantical}}{}^{t_1}_{s} \text{serv} \Leftrightarrow \text{ref} \simeq_{\text{Semantical}}{}^{t_2}_{s} \text{serv}) \wedge \\
& (\text{ref} \not\simeq_{\text{Semantical}}{}^{t_1}_{s} \text{serv} \Leftrightarrow \text{ref} \not\simeq_{\text{Semantical}}{}^{t_2}_{s} \text{serv})) \\
& \forall \text{ref} \in \text{DependableServiceReference}, \\
& \forall \text{serv} \in \text{DependableService}, \forall t_1, t_2 \in T, \forall s \in S
\end{aligned}
\tag{5.6}
$$

The $\simeq_{\text{Semantical}}$ relation is our key to Dependable Dynamic Adaptive Systems. We use it in our Dependable System Infrastructure to decide, whether two Dependable Dynamic Adaptive Components may be bound together (as they are semantically compatible) or not (as they are semantically incompatible).

However, our formal model does not specify this $\simeq_{\text{Semantical}}$ relation in more detail. The specific realization of this relation may, therefore, vary in different Dependable System Infrastructures.

The specific realization of $\simeq_{\text{Semantical}}$ may differ in different Dependable System Infrastructures.

Depending on the degree of required Dependability, you can use different techniques for the realization of this relation. Formal proofs can verify specific

[4] True or false.

properties of a Service Binding like absence of deadlocks while methods like runtime testing verify only a specific execution path. Of course, you can also think of combinations of these options.

Our infrastructure uses runtime testing as a well-known, lightweight approach.

For our reference implementation of the Dependable System Infrastructure, we chose runtime testing as a lightweight mechanism to achieve Dependable Dynamic Adaptive Systems.

In system s_{ae} from our application example, the Dutch C-Unit is semantically compatible with the German P-Unit at $t_n + 1$. However, as soon as the fingerclip slips off at $t_n + 2$, they are not semantically compatible anymore.

This semantical incompatibility results from a changed Behavior Equivalence Class of the second Dutch C-Unit regarding the second German P-Unit. Since the second Dutch C-Unit would calculate the Triage Class to EX due to the pulse rate of zero measured by the second German P-Unit, it now expects a different behavior from this P-Unit.

The second Dutch C-Unit is not semantical compatible with the second German P-Unit due to a failing Compliance Test Case at $t_n + 2$.

It expects, that pulse rate as well as blood pressure are equal to zero now, which is not the case for the second German P-Unit as only the fingerclip slipped off. Thus, the $\simeq_{\text{Semantical}}$ will return false – in our realization this is caused by a failing testcase, detecting this deviation between expected and provided behavior.

This change in semantical Compatibility at $t_n + 2$ is reflected as follows using our model.

$$\text{pUnitReference}_{\text{Dutch}} \simeq_{\text{Semantical}}{}^{t_n+1}_{s_{ae}} \text{pUnitService}_{\text{German}} \tag{5.7}$$

$$\text{pUnitReference}_{\text{Dutch}} \not\simeq_{\text{Semantical}}{}^{t_n+2}_{s_{ae}} \text{pUnitService}_{\text{German}} \tag{5.8}$$

Only semantical compatible Services may be used by a Service Reference

Now that we have defined semantical Compatibility, we can refine the Uses relation in a way, that only semantically compatible Dependable Service References and Dependable Services may be bound together. This is reflected in a refined specification of the Uses relation as follows.

$$\begin{aligned}
\text{Uses}_S^T =_{def}\ & \text{DependableServiceReference} \times T \times S \rightarrow \\
& \text{DependableService} \cup \{\emptyset\}\ : \\
& \text{Uses}_s^t(\text{ref}) \neq \emptyset \Rightarrow \text{ref} \simeq_{\text{Syntactical}} \text{Uses}_s^t(\text{ref}) \wedge \\
& \text{ref} \simeq_{\text{Semantical}} \text{Uses}_s^t(\text{ref}) \wedge \exists \text{c}_{\text{prov}} \in \text{ApplicationComponents}_s^t \mid \\
& \text{c}_{\text{prov}} \neq \text{c}_{\text{req}} \wedge \text{Uses}_s^t(\text{ref}) \in \text{Provides}_s^t(\text{Current}_s^t(\text{c}_{\text{prov}})) \\
& \forall \text{ref} \in \text{Declares}_s^t(\text{Current}_s^t(\text{c}_{\text{req}})), \\
& \forall \text{c}_{\text{req}} \in \text{ApplicationComponents}_s^t, \forall s \in S, \forall t \in T
\end{aligned} \tag{5.9}$$

Semantical Compatibility is only defined for Service Bindings between Dependable Service References and Dependable Services. Since we assume, that Dependable Dynamic Adaptive Components are developed at different points in time and

especially a Dependable Dynamic Adaptive System is not developed as a whole, we cannot assume, that there are additional semantical Compatibility criteria reasoning about semantically Compatibility at the system level. The only criterion we consider are the local semantical Compatibilities between bound Dependable Service References and Dependable Services contained in the Dependable System Configuration. Other criteria like system test criteria are not defined, since we can not predict each specific Dependable System Configuration in advance. However, such system test criteria could be easily added to our system model if necessary for other types of systems.

A Dependable Dynamic Adaptive System in our system model is characterized by the fact, that only semantically compatible Dependable Services and Dependable Service References are bound. This is specified by the isDependable relation as depicted in the following.

$$\text{isDependable}_S^T =_{def} T \times S \rightarrow \text{Boolean} \tag{5.10}$$

$$\begin{aligned} &\text{isDependable}_s^t := true \Leftrightarrow \text{ref} \simeq_{\text{Semantical}} \text{Uses}_s^t(\text{ref}) \\ &\forall \text{ref} \in \text{Declares}_s^t(\text{Current}_s^t(\text{comp})), \forall \text{comp} \in \text{ApplicationComponents}_s^t, \\ &\forall t \in T, \forall s \in S \end{aligned} \tag{5.11}$$

The task of a Dependable System Infrastructure is, to guarantee, that isDependable holds for the Dependable Dynamic Adaptive System at each point during system execution. This is automatically achieved, as long as a Dependable System Infrastructure considers semantical Compatibility during establishing respectively updating the Service Binding as specified in our Uses relation. It must not change Service Bindings except it is indicated by a reconfiguration trigger.

A Dependable Dynamic Adaptive System *must not contain* semantically incompatible bindings.

For completeness we will investigate the combinations of Behavior Equivalence Classes to the behavior class of a binding in the following.

5.5 Binding Behavior Class

The binding behavior class as well as all following behavior specifications is only a combination of the Behavior Equivalence Classes defined for Dependable Services and Dependable Service References. They are described here only for completeness.

Combined Behavior Equivalence Class of a binding is only a combination of the Behavior Equivalence Classes of its Service User and its Service Provider.

The behavior class of a Service Binding is defined by a Dependable Service Reference and a Dependable Service, which are bound. The pair of Dependable Service Reference and Dependable Service defines the Behavior Equivalence Class for this binding from a Service User's point of view, whereas the Dependable Service defines the Behavior Equivalence Class from a Service Provider's point of view.

Thus, the binding behavior relation is defined as follows.

$$\begin{aligned}\text{BindingBehavior}_S^T =_{def} \text{DependableServiceReference} \times T \times S \rightarrow \\ \text{BehaviorEquivalenceClass} \times \\ \text{BehaviorEquivalenceClass} \\ \text{BindingBehavior}_s^t(\text{ref}) := \{\text{ServiceReferenceBehavior}_s^t(\text{ref}, \text{Uses}_s^t(\text{ref})), \\ \text{ServiceBehavior}_s^t(\text{Uses}_s^t(\text{ref}))\} \\ \forall \text{ref} \in \text{DependableServiceReference}, \\ \forall t \in T, \forall s \in S\end{aligned} \tag{5.12}$$

Considering system s_{ae} from our application example, we can observe a behavior class of the binding between the second Dutch C-Unit and the second German P-Unit at $t_n + 1$ as follows.

$$\begin{aligned}&\text{BindingBehavior}_{s_{ae}}^{t_n+1}(\text{pUnitReference}_{\text{Dutch}}2) = \\ &\{\text{ServiceReferenceBehavior}_{s_{ae}}^{t_n+1}(\text{pUnitReference}_{\text{Dutch}}2, \\ &\text{pUnitService}_{\text{German}}2), \\ &\text{ServiceBehavior}_{s_{ae}}^{t_n+1}(\text{pUnitService}_{\text{German}}2)\} = \\ &\{\text{pUnitReferenceUsualOperation}_{\text{Dutch}}, \\ &\text{pUnitServiceUsualOperation}_{\text{German}})\}\end{aligned} \tag{5.13}$$

In the following we will investigate, how the behavior class of a Dependable Component Configuration is defined within our formal system model.

5.6 Dependable Component Configuration Behavior Class

The behavior class of a Dependable Component Configuration consists of two parts: one part containing the Behavior Equivalence Classes of all provided Dependable Services from a Service Provider's point of view, the other part contains a Behavior Equivalence Class for each declared Dependable Service Reference from a Service User's point of view.

Since we already stated before, that a Behavior Equivalence Class of a Dependable Service Reference may depend not only on the state of the declaring Dependable Dynamic Adaptive Component but as well on the externally visible state of a specific Dependable Dynamic Adaptive Component providing a syntactically compatible Dependable Service, we define the behavior class only for Current Configurations, where the specific provider of a Dependable Service is

already known for each Dependable Service Reference. The behavior class of a Current Configuration, therefore, is defined as depicted in the following.

$$\begin{aligned}\text{ConfigurationBehavior}_S^T =_{def}\ & \text{DependableComponentConfiguration} \times \\ & T \times S \to \mathcal{P}(\text{BehaviorEquivalenceClass}) \times \\ & \mathcal{P}(\text{BehaviorEquivalenceClass}) \\ \text{ConfigurationBehavior}_s^t(c) :=\ & \{\text{ProvidedBehavior}_s^t(c), \\ & \text{DeclaredBehavior}_s^t(c)\} \\ & \forall c \in \text{DependableComponentConfiguration}, \\ & \forall t \in T, \forall s \in S\end{aligned} \tag{5.14}$$

This is a simplification, resulting in a specification, which is easier to read. However, it does not limit the capabilities of an Dependable System Infrastructure, as it could reason directly about semantical Compatibility of each syntactically compatible combination of Dependable Service and Dependable Service Reference by referring to their specific Behavior Equivalence Classes.

The ProvidedBehavior respectively DeclaredBehavior relation is defined by the union of the Behavior Equivalence Classes of provided Dependable Services respectively declared Dependable Service References of a specific Dependable Component Configuration. The Behavior Equivalence Class of each Dependable Service Reference is calculated regarding the Dependable Service it is currently bound to.

$$\begin{aligned}\text{ProvidedBehavior}_S^T =_{def}\ & \text{DependableComponentConfiguration} \times \\ & T \times S \to \mathcal{P}(\text{BehaviorEquivalenceClass}) \\ \text{ProvidedBehavior}_s^t(c) :=\ & \bigcup_{\text{serv} \in \text{Provides}_s^t(c)} \text{ServiceBehavior}_s^t(\text{serv}) \\ & \forall c \in \text{DependableComponentConfiguration}, \\ & \forall t \in T, \forall s \in S\end{aligned} \tag{5.15}$$

$$\begin{aligned}\text{DeclaredBehavior}_S^T =_{def}\ & \text{DependableComponentConfiguration} \times \\ & T \times S \to \mathcal{P}(\text{BehaviorEquivalenceClass}) \\ \text{DeclaredBehavior}_s^t(c) :=\ & \bigcup_{\text{ref} \in \text{Declares}_s^t(c)} \text{ServiceReferenceBehavior}_s^t(\text{ref}, \\ & \quad \text{Uses}_s^t(\text{ref})) \\ & \forall c \in \text{DependableComponentConfiguration}, \\ & \forall t \in T, \forall s \in S\end{aligned} \tag{5.16}$$

Looking at system s_{ae} from our application example, therefore, the behavior classes of those Dependable Component Configurations which are Current Configurations within our system at $t_n + 1$ are as shown in the following.

$$\begin{aligned}
&\mathsf{ConfigurationBehavior}_{s_{ae}}^{t_n+1}(\mathsf{cConfiguration2}_{\mathsf{Dutch}}2) = \\
&\quad \{\mathsf{ProvidedBehavior}_{s_{ae}}^{t_n+1}(\mathsf{cConfiguration2}_{\mathsf{Dutch}}2), \\
&\quad \mathsf{DeclaredBehavior}_{s_{ae}}^{t_n+1}(\mathsf{cConfiguration2}_{\mathsf{Dutch}}2)\} \\
&\mathsf{ConfigurationBehavior}_{s_{ae}}^{t_n+1}(\mathsf{pConfiguration1}_{\mathsf{German}}2) = \\
&\quad \{\mathsf{ProvidedBehavior}_{s_{ae}}^{t_n+1}(\mathsf{pConfiguration1}_{\mathsf{German}}2), \\
&\quad \mathsf{DeclaredBehavior}_{s_{ae}}^{t_n+1}(\mathsf{pConfiguration1}_{\mathsf{German}}2)\}
\end{aligned} \tag{5.17}$$

$$\begin{aligned}
&\mathsf{ProvidedBehavior}_{s_{ae}}^{t_n+1}(\mathsf{cConfiguration2}_{\mathsf{Dutch}}2) = \\
&\quad \{\mathsf{ServiceBehavior}_{s_{ae}}^{t_n+1}(\mathsf{cUnitService}_{\mathsf{Dutch}}2)\} \\
&\mathsf{ProvidedBehavior}_{s_{ae}}^{t_n+1}(\mathsf{pConfiguration1}_{\mathsf{German}}2) = \\
&\quad \{\mathsf{ServiceBehavior}_{s_{ae}}^{t_n+1}(\mathsf{pUnitService}_{\mathsf{German}}2)\}
\end{aligned} \tag{5.18}$$

$$\begin{aligned}
&\mathsf{DeclaredBehavior}_{s_{ae}}^{t_n+1}(\mathsf{cConfiguration2}_{\mathsf{Dutch}}2) = \\
&\quad \{\mathsf{ServiceReferenceBehavior}_{s_{ae}}^{t_n+1}(\mathsf{pUnitReference}_{\mathsf{Dutch}}2, \\
&\quad \mathsf{pUnitService}_{\mathsf{German}}2)\} \\
&\mathsf{DeclaredBehavior}_{s_{ae}}^{t_n+1}(\mathsf{pConfiguration1}_{\mathsf{German}}2) = \emptyset
\end{aligned} \tag{5.19}$$

Now that we investigated behavior classes of Current Configurations, we can have a look at the behavior classes of Dependable Dynamic Adaptive Components in the following.

5.7 Dependable Dynamic Adaptive Component Behavior Class

As a Dependable Dynamic Adaptive Component can be active in different Dependable Component Configurations its behavior depends on its Current Configuration. Thus, the behavior class of a Dependable Dynamic Adaptive Component at a Dependability Checkpoint is simply defined as follows:

$$\begin{aligned}
\mathsf{ComponentBehavior}_S^T =_{def} \mathsf{DependableDynamicAdaptiveComponent} \times \\
T \times S \rightarrow \mathcal{P}(\mathsf{BehaviorEquivalenceClass}) \times \\
\mathcal{P}(\mathsf{BehaviorEquivalenceClass}) \\
\mathsf{ComponentBehavior}_s^t(c) := \mathsf{ConfigurationBehavior}_s^t(\mathsf{Current}_s^t(c)) \\
\forall c \in \mathsf{ApplicationComponents}_s^t, \forall t \in T, \forall s \in S
\end{aligned} \tag{5.20}$$

Looking at system s_{ae} from our application example, the behavior class of those Dependable Dynamic Adaptive Components present within the system at $t_n + 1$ is as specified in the following.

$$\begin{aligned}&\mathsf{ComponentBehavior}_{s_{ae}}^{t_n+1}(\mathsf{mUnit}_{\mathsf{German}}) = \\ &\quad \mathsf{ConfigurationBehavior}_{s_{ae}}^{t_n+1}(\mathsf{mConfiguration2}_{\mathsf{German}})\end{aligned} \tag{5.21}$$

$$\begin{aligned}&\mathsf{ComponentBehavior}_{s_{ae}}^{t_n+1}(\mathsf{cUnit}_{\mathsf{German}}) = \\ &\quad \mathsf{ConfigurationBehavior}_{s_{ae}}^{t_n+1}(\mathsf{cConfiguration2}_{\mathsf{German}})\end{aligned} \tag{5.22}$$

$$\begin{aligned}&\mathsf{ComponentBehavior}_{s_{ae}}^{t_n+1}(\mathsf{pUnit}_{\mathsf{German}}) = \\ &\quad \mathsf{ConfigurationBehavior}_{s_{ae}}^{t_n+1}(\mathsf{pConfiguration1}_{\mathsf{German}})\end{aligned} \tag{5.23}$$

$$\begin{aligned}&\mathsf{ComponentBehavior}_{s_{ae}}^{t_n+1}(\mathsf{pUnit}_{\mathsf{German}}2) = \\ &\quad \mathsf{ConfigurationBehavior}_{s_{ae}}^{t_n+1}(\mathsf{pConfiguration1}_{\mathsf{German}}2)\end{aligned} \tag{5.24}$$

$$\begin{aligned}&\mathsf{ComponentBehavior}_{s_{ae}}^{t_n+1}(\mathsf{mUnit}_{\mathsf{Dutch}}) = \\ &\quad \mathsf{ConfigurationBehavior}_{s_{ae}}^{t_n+1}(\mathsf{mConfiguration2}_{\mathsf{Dutch}})\end{aligned} \tag{5.25}$$

$$\begin{aligned}&\mathsf{ComponentBehavior}_{s_{ae}}^{t_n+1}(\mathsf{cUnit}_{\mathsf{Dutch}}) = \\ &\quad \mathsf{ConfigurationBehavior}_{s_{ae}}^{t_n+1}(\mathsf{cConfiguration2}_{\mathsf{Dutch}})\end{aligned} \tag{5.26}$$

$$\begin{aligned}&\mathsf{ComponentBehavior}_{s_{ae}}^{t_n+1}(\mathsf{cUnit}_{\mathsf{Dutch}}2) = \\ &\quad \mathsf{ConfigurationBehavior}_{s_{ae}}^{t_n+1}(\mathsf{cConfiguration2}_{\mathsf{Dutch}}2)\end{aligned} \tag{5.27}$$

$$\begin{aligned}&\mathsf{ComponentBehavior}_{s_{ae}}^{t_n+1}(\mathsf{pUnit}_{\mathsf{Dutch}}) = \\ &\quad \mathsf{ConfigurationBehavior}_{s_{ae}}^{t_n+1}(\mathsf{pConfiguration1}_{\mathsf{Dutch}})\end{aligned} \tag{5.28}$$

Now that we have defined the behavior class of a Dependable Dynamic Adaptive Component by referring to the behavior class of its Current Configuration, we will specify the behavior class of a Dependable Dynamic Adaptive System in the following.

5.8 Dependable Dynamic Adaptive System Behavior Class

It is important to understand, that our approach does not require a specification of the behavior of a Dependable Dynamic Adaptive System, as we only investigate the semantical Compatibility of Dependable Dynamic Adaptive Components to be bound locally based on Behavior Equivalence Classes of Dependable Services and Dependable Service References. The Behavior Equivalence Class of the Dependable Service Reference is associated with a specific expected behavior.

Thus, we can compare this expected behavior[5] of a Dependable Service Reference to a provided behavior of a syntactically compatible Dependable Service at runtime in order to decide, whether the corresponding Dependable Dynamic Adaptive Components are semantically compatible or not. This comparison needs to be performed in the implementation of the $\simeq_{\text{Semantical}}$ relation, which is introduced in the following section.

No vendor is required to specify the behavior of a Dependable Dynamic Adaptive System as this would contradict our assumption, that a Dependable Dynamic Adaptive System is not designed as a whole.

Therefore, at the system level, the behavior class of a Dependable Dynamic Adaptive System at a specific Dependability Checkpoint is defined simply by the union of behavior classes of all Dependable Dynamic Adaptive Components present in the system. This is specified by relation SystemBehavior in the following.

$$\begin{aligned} \text{SystemBehavior}_S^T &=_{def} T \times S \rightarrow \mathcal{P}(\mathcal{P}(\text{BehaviorEquivalenceClass}) \times \\ &\quad \mathcal{P}(\text{BehaviorEquivalenceClass})) \\ \text{SystemBehavior}_s^t &:= \bigcup_{c \in \text{ApplicationComponents}_s^t} \text{ComponentBehavior}_s^t(c) \\ &\quad \forall t \in T, \forall s \in S \end{aligned} \tag{5.29}$$

Looking at system s_{ae} from our application example, therefore, the behavior class of the system at $t_n + 1$ is a set containing the behavior classes of all Depend-

[5] In our reference implementation the expected behavior is defined by Compliance Test Cases.

able Dynamic Adaptive Components as specified in the following.

$$\begin{aligned}\text{SystemBehavior}_{s_{ae}}^{t_n+1} = \{&\text{ComponentBehavior}_{s_{ae}}^{t_n+1}(\text{mUnit}_{\text{German}}),\\ &\text{ComponentBehavior}_{s_{ae}}^{t_n+1}(\text{cUnit}_{\text{German}}),\\ &\text{ComponentBehavior}_{s_{ae}}^{t_n+1}(\text{pUnit}_{\text{German}}),\\ &\text{ComponentBehavior}_{s_{ae}}^{t_n+1}(\text{pUnit}_{\text{German}}2),\\ &\text{ComponentBehavior}_{s_{ae}}^{t_n+1}(\text{mUnit}_{\text{Dutch}}),\\ &\text{ComponentBehavior}_{s_{ae}}^{t_n+1}(\text{cUnit}_{\text{Dutch}}),\\ &\text{ComponentBehavior}_{s_{ae}}^{t_n+1}(\text{cUnit}_{\text{Dutch}}2),\\ &\text{ComponentBehavior}_{s_{ae}}^{t_n+1}(\text{pUnit}_{\text{Dutch}})\}\end{aligned} \tag{5.30}$$

In the following we will investigate, which behavioral triggers exist, which can cause a reconfiguration of a Dependable Dynamic Adaptive System.

5.9 Behavioral Reconfiguration Triggers

In section 4.10 we already investigated, which structural triggers may cause a reconfiguration of the system. We identified *AppComponentsChange*, *ConfigurationChange*, *ConfigOrderChange*, *ServiceChange*, and *ReferenceChange* as structural triggers for the reconfiguration of a Dependable Dynamic Adaptive System. Now we will have a look, how a change in the behavior classes of a Dependable Dynamic Adaptive System respectively a change in the behavior classes of its Dependable Dynamic Adaptive Components may trigger reconfiguration.

If we look at the lowest level of our behavioral model – the behavior of Dependable Service References and Dependable Services – we can think of two behavioral changes, that might affect the System Configuration.

First of all, we consider an *IncompatibilityChange*. We call a situation an IncompatibilityChange, when a Dependable Service Reference uses a Dependable Service and this binding becomes semantically incompatible and, therefore, needs to be removed from the Dependable Dynamic Adaptive System.

A semantically compatible binding becomes incompatible.

An IncompatibilityChange *always* causes a reconfiguration as the incompatible Service Binding is removed[6] from the Dependable System Configuration as reflected in the following equation.

$$\begin{aligned}&\text{IncompatibilityChange}_s^{t_{new}} \Rightarrow \text{BindingChange}_s^{t_{new}}\\ &\forall t_{new} \in T, \forall s \in S\end{aligned} \tag{5.31}$$

[6] It may be replaced by a semantically compatible Service Binding involving a different Service Provider.

In addition Current configurations of Dependable Dynamic Adaptive Components are changed as well, iff a Current configuration is not runnable anymore due to this detected incompatibility.

An IncompatibilityChange is specified as follows.

$$\mathsf{IncompatibilityChange}_S^T =_{def} T \times S \rightarrow \mathsf{boolean} \tag{5.32}$$

$$\begin{aligned}\mathsf{IncompatibilityChange}_s^{t_{new}+1} :=& \exists \mathsf{c} \in (\mathsf{ApplicationComponents}_s^{t_{new}+1} \cap \\ & \mathsf{ApplicationComponents}_s^{t_{new}}) \wedge \\ & \exists \mathsf{ref} \in (\mathsf{Declares}_s^{t_{new}}(\mathsf{Current}_s^{t_{new}}(\mathsf{c})) \cap \\ & \mathsf{Declares}_s^{t_{new}+1}(\mathsf{Current}_s^{t_{new}}(\mathsf{c})))| \\ & \mathsf{ref} \simeq_{\mathsf{Semantical}_s^{t_{new}}} \mathsf{Uses}_s^{t_{new}}(\mathsf{ref}) \wedge \\ & \mathsf{ref} \not\simeq_{\mathsf{Semantical}_s^{t_{new}+1}} \mathsf{Uses}_s^{t_{new}}(\mathsf{ref}) \wedge \\ & \mathsf{Uses}_s^{t_{new}+1}(\mathsf{ref}) \neq \mathsf{Uses}_s^{t_{new}}(\mathsf{ref}) \\ & \forall t_{new} \in T, \forall s \in S\end{aligned} \tag{5.33}$$

A semantically incompatible binding becomes compatible.

Next to IncompatibilityChanges we consider C*ompatibilityChanges* as behavioral reconfiguration triggers. We call a situation a CompatibilityChange, when a Dependable Service Reference becomes semantically compatible with a previously incompatible Dependable Service and, therefore, a Service Binding between this reference and this service could be established within the Dependable Dynamic Adaptive System.

A CompatibilityChange is specified as follows.

$$\begin{aligned}\mathsf{CompatibilityChange}_s^{t_{new}+1} :=& \exists \mathsf{c_{req}}, \mathsf{c_{prov}} \in \\ & (\mathsf{ApplicationComponents}_s^{t_{new}+1} \cap \\ & \mathsf{ApplicationComponents}_s^{t_{new}}) \wedge \\ & \exists \mathsf{conf_{req}} \in (\mathsf{Contains}_s^{t_{new}+1}(\mathsf{c_{req}}) \cap \\ & \mathsf{Contains}_s^{t_{new}}(\mathsf{c_{req}})) \wedge \\ & \exists \mathsf{conf_{prov}} \in (\mathsf{Contains}_s^{t_{new}+1}(\mathsf{c_{prov}}) \cap \\ & \mathsf{Contains}_s^{t_{new}}(\mathsf{c_{prov}})) \wedge \\ & \exists \mathsf{ref} \in (\mathsf{Declares}_s^{t_{new}}(\mathsf{conf_{req}}) \cap \\ & \mathsf{Declares}_s^{t_{new}+1}(\mathsf{conf_{req}})) \wedge \\ & \exists \mathsf{serv} \in (\mathsf{Provides}_s^{t_{new}}(\mathsf{conf_{prov}}) \cap \\ & \mathsf{Provides}_s^{t_{new}+1}(\mathsf{conf_{prov}}))| \\ & \mathsf{ref} \not\simeq_{\mathsf{Semantical}_s^{t_{new}}} \mathsf{serv} \wedge \\ & \mathsf{ref} \simeq_{\mathsf{Semantical}_s^{t_{new}+1}} \mathsf{serv} \\ & \forall t_{new} \in T, \forall s \in S\end{aligned} \tag{5.34}$$

A CompatibilityChange *may* cause a reconfiguration resulting in a change of the Service Binding as well as a change of Current configurations of Dependable Dynamic Adaptive Components. However, it does not necessarily imply these changes, as better Service Providers may be available which are used instead of this newly semantically compatible Dependable Service.

These are the only behavioral reconfiguration triggers we consider in our system model. You can think of other behavioral reconfiguration triggers like changing context (e.g. a Dependable Dynamic Adaptive Component hosted on a device for which the battery is running low). We do not consider them here, as we focus on behavioral reconfiguration triggers that are related to Dependability of the Dependable Dynamic Adaptive System. Therefore, other triggers are out of the scope of this thesis.

Thus, we can specify a BehavioralChange as CompatibilityChange or IncompatibilityChange as specified in the following.

$$\text{BehavioralChange}_S^T =_{def} T \times S \rightarrow \text{boolean} \tag{5.35}$$

$$\begin{aligned} \text{BehavioralChange}_s^{t_{new}} := {} & \text{IncompatibilityChange}_s^{t_{new}} \vee \\ & \text{CompatibilityChange}_s^{t_{new}} \\ & \forall t_{new} \in T, \forall s \in S \end{aligned} \tag{5.36}$$

Now that we have specified BehavioralChange, StructuralChange as well as StructuralReconfiguration, we can define Change as follows.

$$\text{Change}_S^T =_{def} T \times S \rightarrow \text{boolean} \tag{5.37}$$

$$\begin{aligned} \text{Change}_s^{t_{new}} := {} & \text{StructuralChange}_s^{t_{new}} \vee \\ & \text{BehavioralChange}_s^{t_{new}} \vee \\ & \text{StructuralReconfiguration}_s^{t_{new}} \\ & \forall t_{new} \in T, \forall s \in S \end{aligned} \tag{5.38}$$

In our system model, Change is expected to be true for each $t \in T$, as we identified these to be the only points in time relevant for Dependability of a Dependable Dynamic Adaptive System.

Changes in the Dependable System Configuration can not only be triggered by changes of structure, but by changes of a Component's behavior, as well. Thus, our implication from Equation 4.88 needs to be revised as follows.

Each structural reconfiguration implies that a structural change or a behavioral change is present.

$$\begin{aligned} \text{StructuralReconfiguration}_s^{t_{new}} \Rightarrow {} & \text{StructuralChange}_s^{t_{new}} \vee \\ & \text{BehavioralChange}_s^{t_{new}} \\ & \forall t_{new} \in T, \forall s \in S \end{aligned} \tag{5.39}$$

In the following we will give a short summary of our behavioral system model, before we will explain, how we applied our runtime testing approach for Dependable Dynamic Adaptive Systems.

5.10 Summary

In this section you can find the formal definition of our behavioral model concentrated at a single place. Thus, we will repeat the formulas from the previous sections top-down. Whenever definitions were revised, you will find the latest definition in this summary.

$$\begin{aligned}\text{SystemBehavior}_S^T =_{def}\ & T \times S \rightarrow \mathcal{P}(\mathcal{P}(\text{BehaviorEquivalenceClass}) \times \\ & \mathcal{P}(\text{BehaviorEquivalenceClass})) \\ \text{SystemBehavior}_s^t :=\ & \bigcup_{c\ \in\ \text{ApplicationComponents}_s^t} \text{ComponentBehavior}_s^t(c) \\ & \forall t \in T, \forall s \in S\end{aligned} \tag{5.40}$$

$$\begin{aligned}\text{ComponentBehavior}_S^T =_{def}\ & \text{DependableDynamicAdaptiveComponent} \times \\ & T \times S \rightarrow \mathcal{P}(\text{BehaviorEquivalenceClass}) \times \\ & \mathcal{P}(\text{BehaviorEquivalenceClass}) \\ \text{ComponentBehavior}_s^t(c) :=\ & \text{ConfigurationBehavior}_s^t(\text{Current}_s^t(c)) \\ & \forall c \in \text{ApplicationComponents}_s^t, \forall t \in T, \forall s \in S\end{aligned} \tag{5.41}$$

$$\begin{aligned}\text{ConfigurationBehavior}_S^T =_{def}\ & \text{DependableComponentConfiguration} \times \\ & T \times S \rightarrow \mathcal{P}(\text{BehaviorEquivalenceClass}) \times \\ & \mathcal{P}(\text{BehaviorEquivalenceClass}) \\ \text{ConfigurationBehavior}_s^t(c) :=\ & \{\text{ProvidedBehavior}_s^t(c), \\ & \text{DeclaredBehavior}_s^t(c)\} \\ & \forall c \in \text{DependableComponentConfiguration}, \\ & \forall t \in T, \forall s \in S\end{aligned} \tag{5.42}$$

$$\begin{aligned}\text{ProvidedBehavior}_S^T =_{def}\ & \text{DependableComponentConfiguration}\times \\ & T \times S \to \mathcal{P}(\text{BehaviorEquivalenceClass}) \\ \text{ProvidedBehavior}_s^t(c) :=\ & \bigcup_{\text{serv} \in \text{Provides}_s^t(c)} \text{ServiceBehavior}_s^t(\text{serv}) \\ & \forall c \in \text{DependableComponentConfiguration}, \\ & \forall t \in T, \forall s \in S\end{aligned} \tag{5.43}$$

$$\begin{aligned}\text{DeclaredBehavior}_S^T =_{def}\ & \text{DependableComponentConfiguration}\times \\ & T \times S \to \mathcal{P}(\text{BehaviorEquivalenceClass}) \\ \text{DeclaredBehavior}_s^t(c) :=\ & \bigcup_{\text{ref} \in \text{Declares}_s^t(c)} \text{ServiceReferenceBehavior}_s^t(\text{ref}, \\ & \quad \text{Uses}_s^t(\text{ref})) \\ & \forall c \in \text{DependableComponentConfiguration}, \\ & \forall t \in T, \forall s \in S\end{aligned} \tag{5.44}$$

$$\begin{aligned}\text{BindingBehavior}_S^T =_{def}\ & \text{DependableServiceReference} \times T \times S \to \\ & \text{BehaviorEquivalenceClass}\times \\ & \text{BehaviorEquivalenceClass} \\ \text{BindingBehavior}_s^t(\text{ref}) :=\ & \{\text{ServiceReferenceBehavior}_s^t(\text{ref}, \text{Uses}_s^t(\text{ref})), \\ & \text{ServiceBehavior}_s^t(\text{Uses}_s^t(\text{ref}))\} \\ & \forall \text{ref} \in \text{DependableServiceReference}, \\ & \forall t \in T, \forall s \in S\end{aligned} \tag{5.45}$$

$$\begin{aligned}\text{ServiceBehavior}_S^T =_{def}\ & \text{DependableService} \times T \times S \to \\ & \text{BehaviorEquivalenceClass}\end{aligned} \tag{5.46}$$

$$\begin{aligned}\text{ServiceReferenceBehavior}_S^T =_{def}\ & \text{DependableServiceReference}\times \\ & \text{DependableService} \times T \times S \to \\ & \text{BehaviorEquivalenceClass} \cup \{\emptyset\}\ : \\ & \text{ServiceReferenceBehavior}_s^t(\text{ref}, \text{serv}) = \emptyset \Leftrightarrow \\ & \text{ref} \not\approx_{\text{Syntactical}} \text{serv} \\ & \forall \text{ref} \in \text{DependableServiceReference}, \\ & \forall \text{serv} \in \text{DependableService}, \forall t \in T, \forall s \in S\end{aligned} \tag{5.47}$$

$$\begin{aligned}\simeq_{\text{Semantical}}{}_S^T =_{def}\ & \text{DependableServiceReference} \times \text{DependableService} \times \\ & T \times S \rightarrow \text{Boolean}\ : \\ & (\text{ServiceBehavior}_s^{t_1}(\text{serv}) = \text{ServiceBehavior}_s^{t_2}(\text{serv}) \wedge \\ & \text{ServiceReferenceBehavior}_s^{t_1}(\text{ref}, \text{serv}) = \\ & \text{ServiceReferenceBehavior}_s^{t_2}(\text{ref}, \text{serv})) \Rightarrow \\ & ((\text{ref} \simeq_{\text{Semantical}}{}_s^{t_1} \text{serv} \Leftrightarrow \text{ref} \simeq_{\text{Semantical}}{}_s^{t_2} \text{serv}) \wedge \\ & (\text{ref} \not\simeq_{\text{Semantical}}{}_s^{t_1} \text{serv} \Leftrightarrow \text{ref} \not\simeq_{\text{Semantical}}{}_s^{t_2} \text{serv})) \\ & \forall \text{ref} \in \text{DependableServiceReference}, \\ & \forall \text{serv} \in \text{DependableService}, \forall t_1, t_2 \in T, \forall s \in S\end{aligned} \tag{5.48}$$

$$\begin{aligned}\text{Uses}_S^T =_{def}\ & \text{DependableServiceReference} \times T \times S \rightarrow \\ & \text{DependableService} \cup \{\emptyset\}\ : \\ & \text{Uses}_s^t(\text{ref}) \neq \emptyset \Rightarrow \text{ref} \simeq_{\text{Syntactical}} \text{Uses}_s^t(\text{ref}) \wedge \\ & \text{ref} \simeq_{\text{Semantical}} \text{Uses}_s^t(\text{ref}) \wedge \exists \text{c}_{\text{prov}} \in \text{ApplicationComponents}_s^t \mid \\ & \text{c}_{\text{prov}} \neq \text{c}_{\text{req}} \wedge \text{Uses}_s^t(\text{ref}) \in \text{Provides}_s^t(\text{Current}_s^t(\text{c}_{\text{prov}})) \\ & \forall \text{ref} \in \text{Declares}_s^t(\text{Current}_s^t(\text{c}_{\text{req}})), \\ & \forall \text{c}_{\text{req}} \in \text{ApplicationComponents}_s^t, \forall s \in S, \forall t \in T\end{aligned} \tag{5.49}$$

$$\text{isDependable}_S^T =_{def} T \times S \rightarrow \text{Boolean} \tag{5.50}$$

$$\begin{aligned}& \text{isDependable}_s^t := true \Leftrightarrow \text{ref} \simeq_{\text{Semantical}} \text{Uses}_s^t(\text{ref}) \\ & \forall \text{ref} \in \text{Declares}_s^t(\text{Current}_s^t(\text{comp})), \forall \text{comp} \in \text{ApplicationComponents}_s^t, \\ & \forall t \in T, \forall s \in S\end{aligned} \tag{5.51}$$

Now that we have specified our formal model for Dependable Dynamic Adaptive Systems, we will explain in the following how we implemented it in our Dependable System Infrastructure DAiSI and how Component vendors can benefit from it in order to achieve Dependable Dynamic Adaptive Systems.

By the time you've sorted out a complicated idea into little steps that even a stupid machine can deal with, you've learned something about it yourself.

Douglas Adams

6

Realization of an Infrastructure for Dependable Dynamic Adaptive Systems

The realization of our application example, serving as development guidelines, can be found in Appendix B.

This chapter describes the realization of our Dependable System Infrastructure DAiSI, which includes an implementation of our formal system model introduced before. You can find the implementation of our application example in Appendix B. It serves as a development guideline for Component vendors, wanting to provide Dependable Dynamic Adaptive Components.

The chapter serves as a validation of our approach, as it provides an implementation of our formal system model and shows, that Dependable Dynamic Adaptive Systems can be bound automatically at runtime using our system model.

Dependable Dynamic Adaptive Systems consist of layers.

In [NKA+07], we already described, that Dependable Dynamic Adaptive Systems consist of the following four layers as depicted in Figure 6.1:

- Application Layer: This layer contains application specific Dependable Dynamic Adaptive Components.
- Infrastructure Layer: It consists of infrastructure Components providing general services for Dependable Dynamic Adaptive Systems like automatic establishment of a Dependable System Configuration or lifecycle management.
- System Layer: Within this layer general services like a communication infrastructure or supported programming languages are provided, that are

required by the infrastructure layer.

- Physical Layer: This layer defines the supported target hardware of Dependable Dynamic Adaptive Systems running on top of this Dependable System Infrastructure.

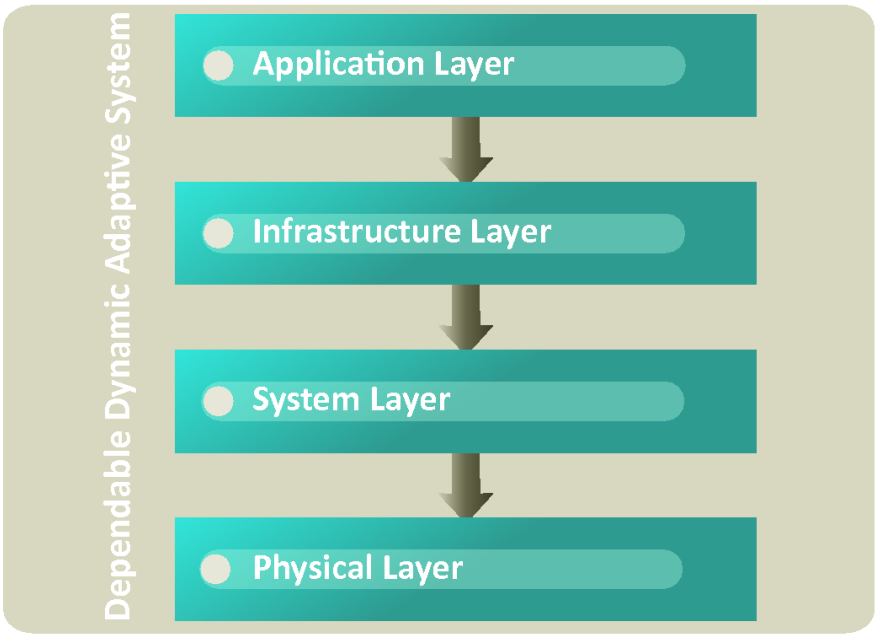


Figure 6.1: Layers of a Dependable Dynamic Adaptive System.

Infrastructure layer = DAiSI.

Within this thesis, we contribute to the infrastructure layer and to the application layer. Our contribution to the infrastructure layer is our reference implementation of a *D*ependable Dynamic *A*daptive *S*ystem *I*nfrastructure called DAiSI. It refers to our system model from Chapter 4 and Chapter 5 in order to establish and update a Dependable System Configuration in Dependable Dynamic Adaptive Systems automatically at runtime.

Application layer = component framework.

As our contribution to the application layer, we provide a Component framework for vendors of Dependable Dynamic Adaptive Components. This framework consists of abstract classes, which provide a reference implementation of our Component model from Chapter 4 and Chapter 5.

In the following sections we will describe these contributions in detail. For the two lowest layers we use existing standards like IIOP or Ethernet. Thus, we do not provide own contributions to these layers.

6.1 Dependable Dynamic Adaptive System Infrastructure

Our reference implementation DAiSI consists of several infrastructure Components to achieve automatic Dependable System Configuration. We will focus on two of these infrastructure Components – the Node Component and DAiSI's heartbeat: the Dependable Configuration Component – in the following, as they are required in each Dependable Dynamic Adaptive System based on DAiSI.

Infrastructure components = Dependable Configuration Component + Node Component + Event Component + Device Bay Component.

Next to them, infrastructure Components like an Event Component or a Device Bay Component are optional infrastructure Components in Dependable Dynamic Adaptive Systems. The Event Component provides asynchronous communication while the Device Bay Component enables integration of Dependable Dynamic Adaptive Components hosted on resource-constrained devices like smartphones.

These infrastructure Components are not described within this thesis. You can find out more about them in [KNW06] and [AKNR09]. An overview of DAiSI's infrastructure Components is depicted in Figure 6.2.

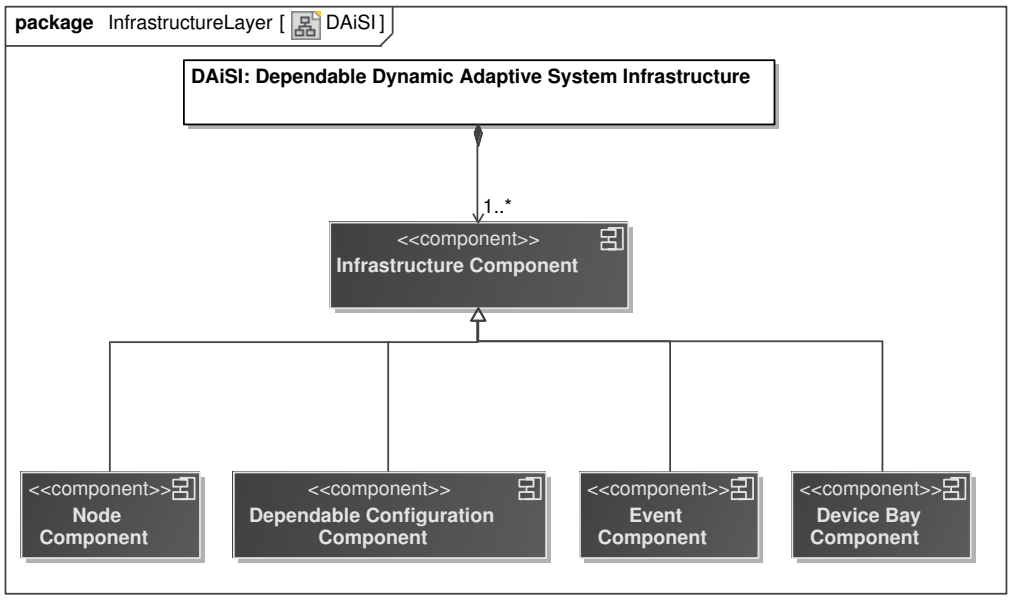


Figure 6.2: An Overview of DAiSI's Infrastructure Components.

DAiSI is based on CORBA [Gro04b]. Thus, Service Interfaces provided respectively required by Dependable Dynamic Adaptive Components need to be specified in IDL. The public interfaces provided by our infrastructure Components are consequently specified in IDL as well. Thus, they can be accessed remotely by other

Service Interfaces as well as interfaces of infrastructure Components are specified in IDL due to underlying CORBA.

infrastructure Components easily.

We implemented DAiSI in Java, nevertheless you could implement a Dependable System Infrastructure in other programming languages as well – previous versions of DAiSI have been implemented in C++ as proof of concept [KNW06].

6.1.1 Node Component

A Node Component is responsible for lifecycle management of Components.

The Node Component is a specific infrastructure Component present at each physical node[1] within a Dependable Dynamic Adaptive System. It manages the lifecycle of Dependable Dynamic Adaptive Components executed on this node. Thus, it is responsible for starting up Dependable Dynamic Adaptive Components or shutting them down.

The relationship between a physical node, a Node Component and Dependable Dynamic Adaptive Components is depicted in Figure 6.3. A physical node may instantiate multiple Node Components. Each Node Component may manage several Dependable Dynamic Adaptive Components.

A node model specifies, which Components should be started by a Node Component.

A Node Component requires a specification of Dependable Dynamic Adaptive Components that should be hosted at this specific node. A *node model* specifies these Dependable Dynamic Adaptive Components.

The Node Component parses a given node model, starts up those specified Dependable Dynamic Adaptive Components and registers them at DAiSI's Dependable Configuration Component, which is responsible for establishing a Dependable System Configuration.

Usage of DAiSI's Node Component

DAiSI's Node Component implements no remotely accessible interface. Thus, a Node Component can only be used by handing over a node model specifying Dependable Dynamic Adaptive Components and infrastructure Components that it should start up or by using its graphical user interface. This can be done by passing a node model XML file as an argument to the Node Component during its startup.

A node model XML file looks as depicted in Listing 6.1.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <daisi:Node xmi:version="2.0" xmlns:xmi="http://www.omg.org/
    XMI" xmlns:daisi="de.tuc.ifi.sse.daisi">
3  <infrastructureComponent name="
     DependableConfigurationComponent"/>
4  <dependableDynamicAdaptiveComponent name="de.tuc.ifi.sse.
     daisi.applicationExample.germanVendor.cUnit.CUnitImpl"
     requestRun="true" />
```

[1] A physical node here means a device, e.g. a smartphone or a laptop, hosting Dependable Dynamic Adaptive Components.

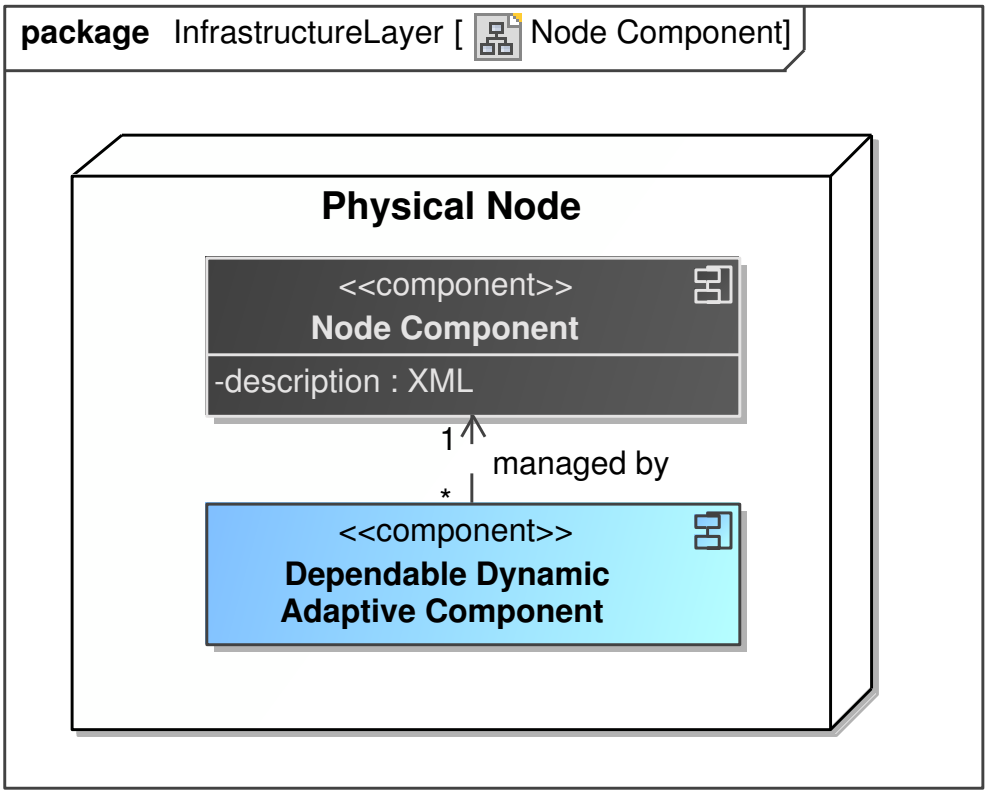


Figure 6.3: Relationship between Physical Node, Node Component, and Dependable Dynamic Adaptive Components.

```
5 </daisi:Node>
```

Listing 6.1: Node Model for a Physical Node that Should Host a German C-Unit.

Within a node model XML file, you specify Dependable Dynamic Adaptive Components, that should be started by a node, by adding child nodes called *dependableDynamicAdaptiveComponent* to the XML root node *daisi:node*. This is depicted in line 4 of Listing 6.1.

You use attribute *name* to specify, which Dependable Dynamic Adaptive Component should be started up by this Node Component. It contains a full classifier to the Dependable Dynamic Adaptive Component's Java class.

***requestRun* specifies, whether a Dependable Dynamic Adaptive Component should be activated, if it is not used by other Components.**

Attribute *requestRun* influences *how* this Dependable Dynamic Adaptive Component is registered at the Dependable Configuration Component. It describes, whether this Dependable Dynamic Adaptive Component is a Component, that provides a feature to a user of a Dependable Dynamic Adaptive System on its own.

Dependable Dynamic Adaptive Components, which specify this as *true*, will be activated by DAiSI's Dependable Configuration Component even if no other Dependable Dynamic Adaptive Component requires any Dependable Service provided by this Component.

Those Dependable Dynamic Adaptive Components, for which *false* has been specified, will not be activated in this case. This can be used to save power for sensor Components as they only need to be active, if their values are queried by other Dependable Dynamic Adaptive Components.

In our application example, German C-Units may be used by medics, which are not equipped with M-Units. These medics set the Triage Class of a casualty by pushing a C-Unit's button. Thus, attribute requestRun is specified to true in this example, as C-Units in our example need to be activated even if no other Dependable Dynamic Adaptive Component uses them.

Next to Dependable Dynamic Adaptive Components, a Node Component may start up *infrastructure Components* . For example, a Node Component will start up a Dependable Configuration Component, if none is available within this Dependable Dynamic Adaptive System yet.

Additional infrastructure Components that need to be started up by a specific node are specified in the node model XML file by adding a child node called *infastructureComponent* to the XML root node. Within this child node, only the name of the infrastructure Component is specified.

Valid names of infrastructure Components within our DAiSI implementation are *DependableConfigurationComponent*, *DeviceBayComponent*, and *EventComponent*[2]. The corresponding infrastructure Component will be started up by the Node Component.

The specification of a Dependable Configuration Component within a node model is optional: as a Dependable Configuration Component is mandatory within Dependable Dynamic Adaptive Systems it will always be started by a Node Component, if it has not been started before by another Node Component. Since DAiSI's Dependable Configuration Component is implemented as a singleton, a node will not start a Dependable Configuration Component if one is already present in the Dependable Dynamic Adaptive System, although it may be specified in this node model.

Summed up, a Node Component behaves as depicted in Figure 6.4. The registration of a Dependable Dynamic Adaptive Component itself is only included as a reference in this diagram, as it is discussed in section 6.1.2 when explaining DAiSI's Dependable Configuration Component.

[2]Currently there is no implementation of an Event Component. There is only a concept, described in [AKNR09].

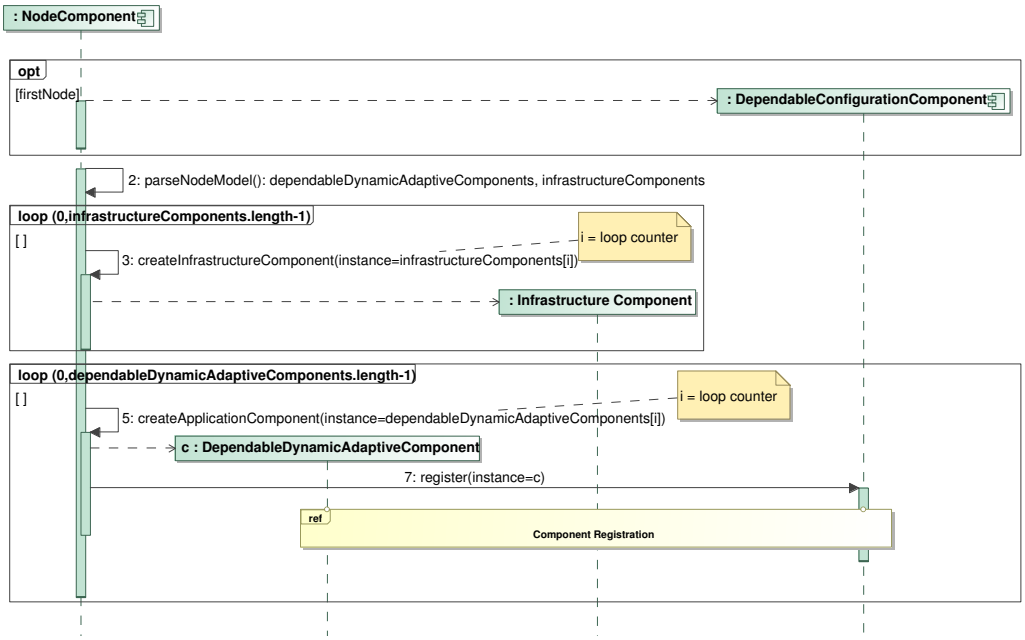


Figure 6.4: Behavior of a Node Component During Startup.

Graphical User Interface

If no argument is handed over to a Node Component during startup, a dialog as depicted in Figure 6.5 will pop up, asking for a node model XML file. This dialog allows you to easily select an appropriate node model, specifying, which Dependable Dynamic Adaptive Components or infrastructure Components need to be started by this node.

After startup, a second graphical user interface pops up, displaying all Dependable Dynamic Adaptive Components and infrastructure Components, that have been started by this Node Component. This user interface is depicted in Figure 6.6. On the top you can see a list of all Dependable Dynamic Adaptive Components started by this Node Component. At the bottom an analogous list of infrastructure Components is displayed.

In Figure 6.6, the list of infrastructure Components has a yellow background. This indicates, that DAiSI's Dependable Configuration Component has been started up by this Node Component. Thus, you should not shut down this Node Component before all other Node Components of this Dependable Dynamic Adaptive System have been shut down. Otherwise this system will not be capable of updating its Dependable System Configuration due to active Change triggers anymore.

By pushing the bottommost buttons of a Node Component's user interface, you

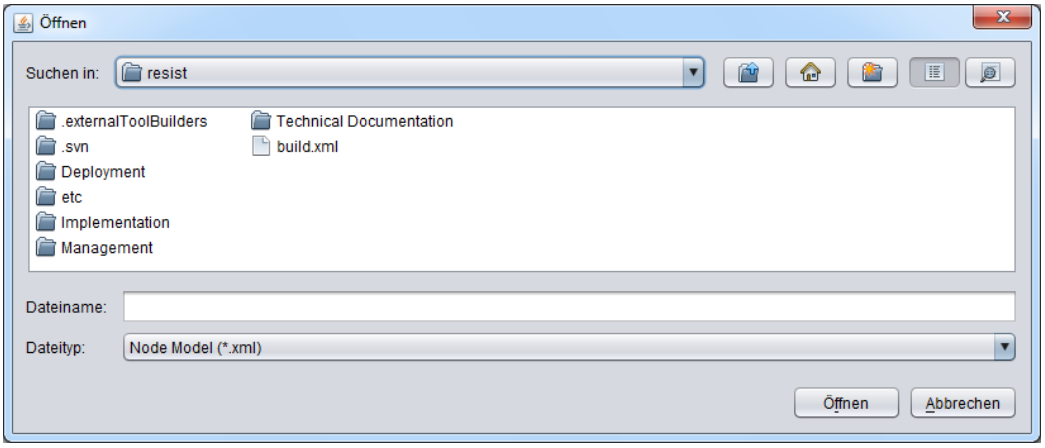


Figure 6.5: Dialog Asking for a Node Model XML File During Startup of the Node Component.

The Node Component's user interface enables us to unregister Components.

can refresh the lists[3] or shut down selected Dependable Dynamic Adaptive Components respectively infrastructure Components started by this Node Component. The latter requires you to select those Components in these lists before pushing the "Shutdown selected Components" button.

Another option of shutting down Components started by a specific node is shutting down a whole Node Component by closing its window using the cross-button in the upper right corner.

Whenever a Node Component shuts down a Dependable Dynamic Adaptive Component, it will unregister this specific Component at the Dependable Configuration Component to notify it, that this Component is not available anymore. This enables the Dependable Configuration Component to update the Dependable System Configuration accordingly.

As Node Components notify the Dependable Configuration Component of entering or leaving Dependable Dynamic Adaptive Components, this realizes the AppComponentsChange trigger of our formal system model.

6.1.2 Dependable Configuration Component

The Dependable Configuration Component is DAiSI's heartbeat regarding Dependable Dynamic Adaptive Systems. It establishes and updates the System Configuration of a Dependable Dynamic Adaptive System at runtime, ensuring that only Dependable System Configurations will be established.

To achieve this, the Dependable Configuration Component manages the state

[3] For example, if a Dependable Dynamic Adaptive Component has been shut down.

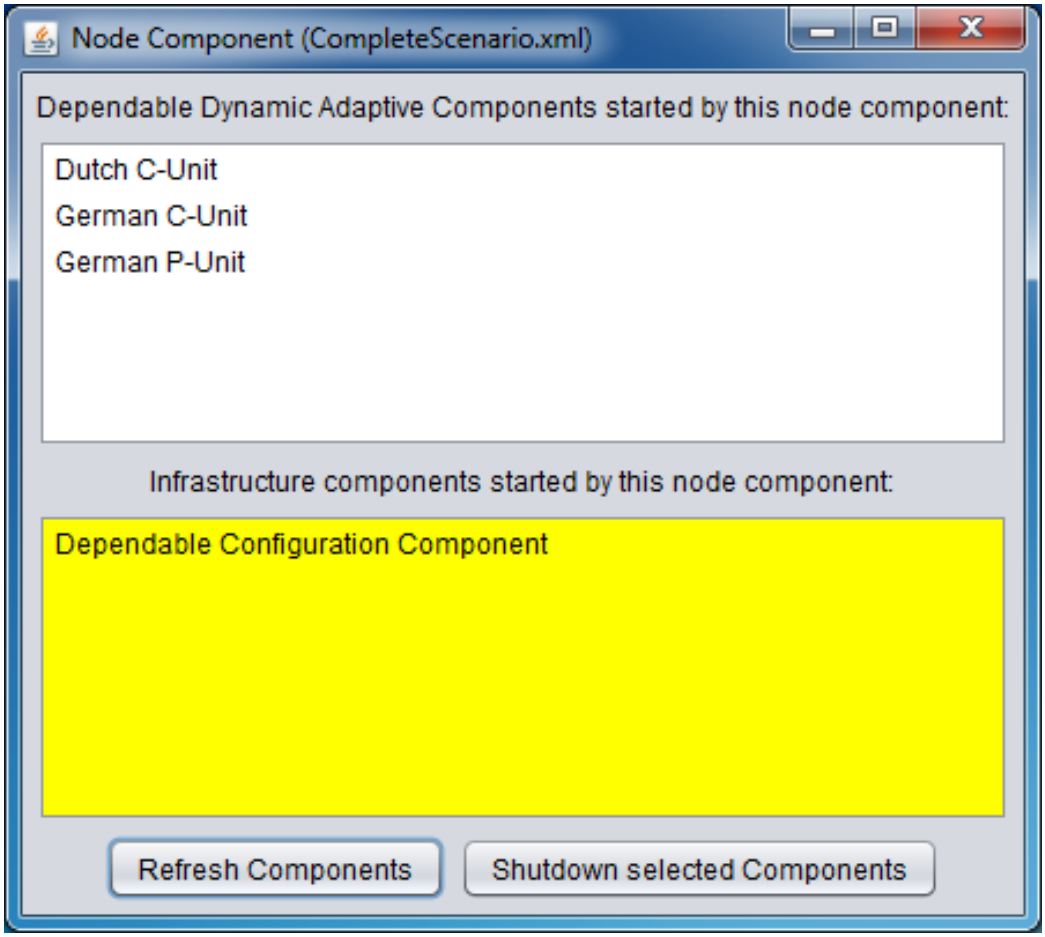


Figure 6.6: The Graphical User Interface Allowing Interaction with Components Started by a Specific Node Component.

States of a Dependable Dynamic Adaptive Component.

of Dependable Dynamic Adaptive Components acting as Service Partners. During runtime, each Service Partner can be in one out of four different states. These states are introduced in the following. A Service Partner can be...

- ... *registered*: This is the starting state of a Service Partner. A Service Partner is registered, as soon as he enters a Dependable Dynamic Adaptive System. He remains in this state, until a semantically compatible Service Provider is available for each declared Dependable Service References of one of his Dependable Component Configurations.
- ... *runnable*: This state indicates, that for at least one of its Dependable Component Configurations all required Service Interfaces are offered by Unknown Service Partners, which are runnable respectively running as well.

The Service Partner remains in this state, if no other Service Partner requires his offered Dependable Services and the Service Partner is declared as a passive Service Partner. This means, that this Service Partner should only run, iff at least one other Service Partner exists, which requires a Dependable Service, offered by this passive Service Partner[4].

The testing mode is required, to prevent side-effects of test case execution.

- ... in *testing mode*: A Service Partner is in testing mode, while the interaction with other Service Partners is tested. This is similar to the sandbox-principle which has been applied to different domains in the past [WLAG93,GWTB96, GMPS97, KDF04]. The goal of sandboxing is the capability of integrating untrusted third-party code into your system. Since you want to avoid negative side-effects of this integration on your system you transfer the third-party code in a special environment for execution. This environment guarantees, that the execution does not harm your system and, therefore, is free of side-effects.

  For our approach the trigger for sandboxing is, when we put a Service Partner into testing mode. All changes regarding the internal state of a Service Partner in testing mode have to be rolled back as soon, as this Service Partner leaves this state – this is a difference to the classical sandboxing approach, where you do not allow any temporary modifications of the system state. Instead you would provide a simulated system environment for interaction with sandboxed objects.

  Rolling back the state of Service Partners is necessary in order to avoid side-effects of a semantical Compatibility test on a running system. Since you allow changes of the system state, you need to make sure, that you can rollback all of these changes. If testing interaction with a Service Partner includes irreversible changes in state (like an explosion of an airbag caused by an airbag controller Component), a Service Partner, therefore, needs to replace these irreversible state changes by steps simulating these state changes whenever it is in testing mode.

- ... *running*: In this state a Service Partner interacts with other Service Partners by providing respectively using provided Dependable Services. He leaves this state, if its Behavior Equivalence Class or the Behavior Equivalence Class of a Service Partner he is bound to by a Dependable Component Binding changes towards an untested combination of Behavior Equivalence Classes. In this case we need to reevaluate the semantical Compatibility of this Dependable Component Binding. Thus, this Service Partner enters the testing

[4]The P-Unit from our application example could be such a passive Service Partner: It does not make much sense to execute this Dependable Dynamic Adaptive Component unless there is a Service Partner using its Dependable Service. Otherwise we would just drain the battery of the P-Unit.

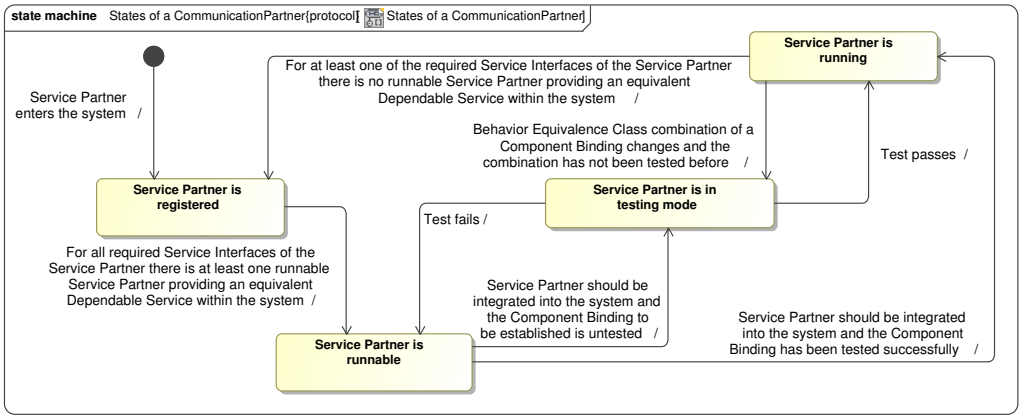


Figure 6.7: Different States of a Service Partner and Their Transitions.

mode state and test cases are executed for the Dependable Component Binding.

An overview, showing these four states and transitions between them is depicted in Figure 6.7.

From an abstract point of view, the Dependable Configuration Component behaves as depicted in Figure 6.8. It deals with registering Components while it monitors changes in Behavior Equivalence Classes of Dependable Services or Dependable Service References in parallel to recognizing active CompatibilityChange triggers and reacting to them by changing the Dependable System Configuration.

The Dependable Configuration Component monitors for change triggers in parallel to registration.

During registration of a Dependable Dynamic Adaptive Component it derives the syntactical Compatibility to other Components. In the following it activates this registering Component in its best runnable Dependable Component Configuration and binds it to other Components, if the registering Component has requested to run or its Dependable Services are required by other running Components in the Dependable Dynamic Adaptive System.

Summed up, it realizes the Change trigger from our formal model in order to recognize each threat to Dependability and to react to them appropriately. In general there are two strategies, how these triggers can be realized:

The Change **trigger can be realized by a polling or observer strategy.**

1. Polling Strategy: The Dependable Configuration Component queries each Dependable Dynamic Adaptive Component in the Dependable Dynamic Adaptive System periodically, to check whether a Change trigger is active.

2. Observer Strategy: Dependable Dynamic Adaptive Components notify the Dependable Configuration Component whenever a Change trigger is active.

We chose the observer strategy due to less overhead and a quicker recognition of active triggers.

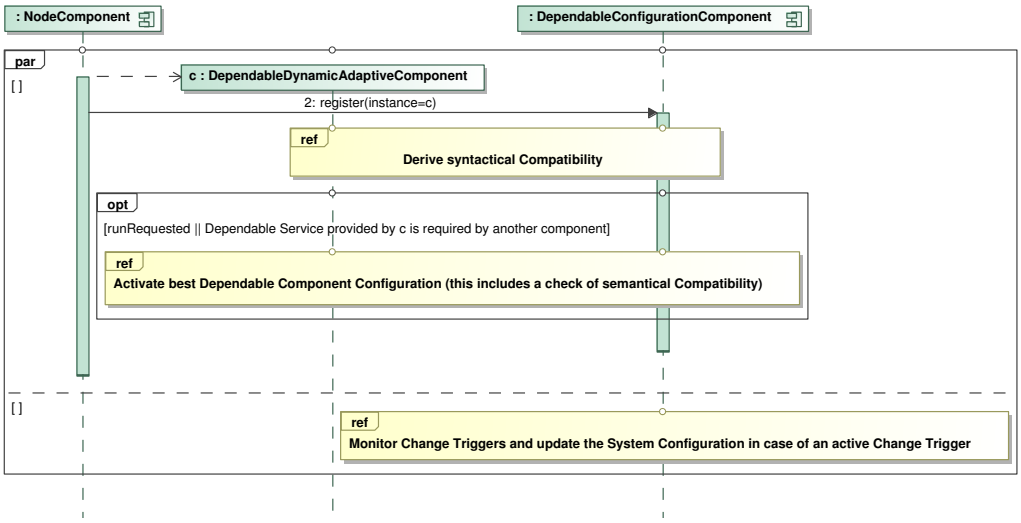


Figure 6.8: Behavior of the Dependable Configuration Component from an Abstract Point of View.

We decided to realize the Change trigger by using the observer strategy. This strategy has two main advantages: it *causes less overhead*, since we do not need to poll periodically while no Change trigger is active and we *recognize active* Change *triggers immediately* instead of recognizing them in the next polling period.

Assumption: Dependable Dynamic Adaptive Components notify the Dependable Configuration Component if triggers are active.

However, by this realization we assume, that each Dependable Dynamic Adaptive Component immediately notifies the Dependable Configuration Component, whenever a Change trigger is active. If we do not rely on this notification, only the polling strategy would be applicable.

The Dependable Configuration Component is automatically started up together with the first Node Component in a Dependable Dynamic Adaptive System. It is realized as a singleton, however, one could realize it in a distributed way as well having several Dependable Configuration Components, which cooperate to establish a Dependable System Configuration.

Simplification: DAiSI does not consider structural changes of a Component after registration.

DAiSI's Node Component registers each Dependable Dynamic Adaptive Component at the Dependable Configuration Component during startup of the Components. As a simplification, DAiSI does not consider structural changes of Components after their registration.

Thus, the Dependable Configuration Component only updates its view of the Dependable Dynamic Adaptive System by querying the structure of Components *during registration*. It will query, which Dependable Component Configurations a Component contains, and which Dependable Services are provided respectively which Dependable Service References are declared within these configurations.

Based on this structural information it can derive syntactical Compatibility to other Components present in the Dependable Dynamic Adaptive System. As a simplification of our formal system model, a Dependable Service and a Dependable Service Reference are treated as syntactically compatible, iff they implement respectively refer to the same type. We do not compare methods and attributes of the corresponding Service Interfaces.

Simplification: syntactical Compatibility is reduced to identical types.

This enables a much less complicated implementation of DAiSI. It would not be impossible to implement syntactical Compatibility as defined in our formal model, but the Uses relation from our formal model could not be easily set by reflection, since Java would blame the incorrect type of this assignment at runtime.

Using its structural view of the Dependable Dynamic Adaptive Components present in the system, the Dependable Dynamic Adaptive System tries to activate the *best* activatable Dependable Component Configuration of each Component. Thus, in our reference implementation DAiSI, the $\geq$ relation from our formal system model is realized based on an integer quality value.

$\geq$ is based on integer quality values of the configurations. If this value is not set, the order of adding configurations is considered.

If this quality value has not been set, it considers the order of adding Dependable Component Configurations to a Component. The configuration added first is treated as best configuration. A more sophisticated $\geq$ relation could be easily implemented, if, for example, internal properties of a Component should be considered to derive an order of configurations.

Since we want to consider Dependability during establishing a Dependable System Configuration, the Dependable Configuration Component also considers the $\simeq_{\text{Semantical}}$ relation from our system model. This relation can be implemented in many different ways using those verification and validation techniques discussed in section 2.2.

Since we don't want to restrict the specification language and want to retain a good system performance, our reference implementation DAiSI uses runtime testing to achieve a Dependable Dynamic Adaptive System – thus, the $\simeq_{\text{Semantical}}$ relation is realized by executing Compliance Test Cases and reasoning about their results. A more detailed view, how DAiSI's Dependable Configuration Component is realized – including those behaviors referred to in Figure 6.8 – can be found in Subsection *Realization*.

$\simeq_{\text{Semantical}}$ is realized by the well-known and lightweight mechanism of runtime testing.

Usage of DAiSI's Configuration Component

There are mainly three different use cases, how someone wants to use the Dependable Configuration Component.

Three different use cases.

1. He wants to register or unregister Dependable Dynamic Adaptive Components in a Dependable Dynamic Adaptive System.

2. He wants to notify the Dependable Configuration Component, that a Change

trigger is active due to a change of the internal state of a Dependable Dynamic Adaptive Component since this Component may behave differently now.

3. He wants to monitor a Dependable Dynamic Adaptive System.

Each of these use cases is addressed by the interface of the Dependable Configuration Component, which is depicted in Figure 6.9.

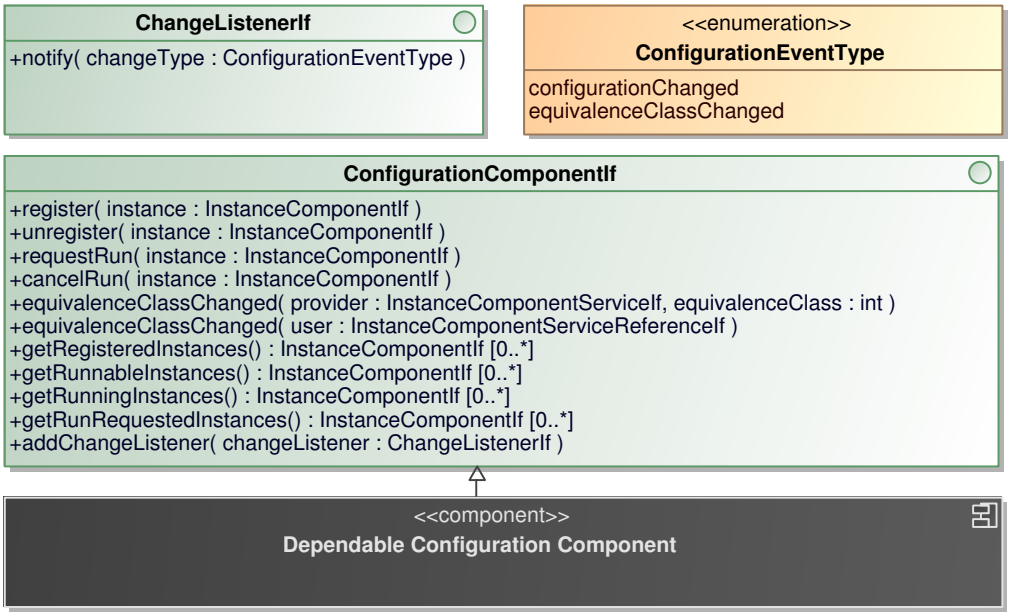


Figure 6.9: Interface of DAiSI's Dependable Configuration Component.

Four methods deal with registration or unregistration of Dependable Dynamic Adaptive Components. By calling the `register` method, you register a Dependable Dynamic Adaptive Component within a Dependable Dynamic Adaptive System. This causes, that the Dependable Configuration Component updates the Dependable System Configuration and may bind this registering Component to existing Dependable Dynamic Adaptive Components.

Simplification: We sequenced the registration / unregistration of Dependable Dynamic Adaptive Components.

If a Dependable Dynamic Adaptive Component wants to leave a Dependable Dynamic Adaptive System it calls the `unregister` method. This causes that the Dependable Configuration Component once again updates the Dependable System Configuration – it will remove this Component and all Dependable Service Bindings incorporating this Component.

These two methods are a simplified implementation of the AppComponentsChange trigger of our formal system model. Since Depend-

able Dynamic Adaptive Components are expected to call these methods when entering or leaving a Dependable Dynamic Adaptive System, each call of one of these methods implies this trigger. As a simplification, we sequenced this trigger in our realization – thus it is not possible, that x Components join at Dependability Checkpoint t causing a new Dependability Checkpoint $t + 1$; instead they register one after another leading to $t + 1$ to $t + x$.

Crashing Components can be detected by applying a watchdog mechanism.

This implementation does not recognize crashing Components. This could be added by applying a watchdog mechanism [CFHL07], where the liveness of all Components is periodically checked and, therefore, crashes can be detected.

Next to the `register` and `unregister` methods, method `requestRun` can be used to request, that this Dependable Dynamic Adaptive Component should be activated by DAiSI's Dependable Configuration Component even, if no other Component requires any Dependable Service provided by this Component.

By default these Components are not activated unless they are required by another Component, which is reasonable as long as a Component does not provide any functionality to an end user directly. Calling method `cancelRun` restores this default behavior for Dependable Dynamic Adaptive Components if `requestRun` has been called for them before.

If the internal state of a Dependable Dynamic Adaptive Component has changed, it needs to notify the Dependable Configuration Component, as its provided Dependable Services may behave differently now or as he may expect a different behavior of used Dependable Services. This notification is done by calling the `equivalenceClassChanged` method. It is available to notify the Dependable Configuration Component of a change for Dependable Services (changed *provided* behavior) as well as for Dependable Service References (changed *expected* behavior).

In the first case, it has two parameters: the Dependable Service, whose behavior may be different now, and the Behavior Equivalence Class, which the Service Provider now calculates for this Dependable Service. In the later case, it only has the Dependable Service Reference as parameter, as the Behavior Equivalence Class depends on the Dependable Service, for which the semantical Compatibility should be checked.

This notification causes, that DAiSI's Dependable Configuration Component recalculates the Behavior Equivalence Classes for all Dependable Service Bindings including this Dependable Service respectively Dependable Service Reference. In case of changed Behavior Equivalence Classes the Dependable Configuration Component will execute test cases for the corresponding Dependable Service Bindings to evaluate their semantical Compatibility and change the Dependable System Configuration in case of a detected incompatibility.

Besides these use cases how Dependable Dynamic Adaptive Components interact with the Dependable Configuration Component, its interface also supports

Monitoring Components visualize a running Dependable Dynamic Adaptive System.

monitoring Components, that visualize the Dependable Dynamic Adaptive System including its Components in order to provide a graphical debug view of the system[5]. One reference implementation of such a monitoring Component is described in the following subsection.

Monitoring Components can add themselves as listener to the Dependable Configuration Component by calling its `addChangeListener` method. This causes, that the `notify` method of these monitoring Components will be called by the Dependable Configuration Component each time, the Dependable System Configuration or a Behavior Equivalence Class of a Dependable Service Binding changes.

DAiSI's Dependable Configuration Component offers four getter methods to query the currently established Dependable System Configuration. The methods `getRegisteredInstances()`, `getRunnableInstances()`, `getRunningInstances()`, and `getRunRequestedInstances()` return different subsets of all Dependable Dynamic Adaptive Components present in the system. `getRegisteredInstances()` returns each Dependable Dynamic Adaptive Component, while `getRunningInstances()` returns only those Components, which currently run – meaning that they have a Current Configuration.

Method `getRunRequestedInstances()` returns all Components, for which a run request has been submitted by calling `requestRun()`. As Components, which have not requested to run, are not activated by DAiSI's Dependable Configuration Component unless they are required by other Components, `getRunnableInstances()` returns all Components, which could be activated by DAiSI – this includes all running Components.

These methods are not independent of each other – the subset relations `getRegisteredInstances()` $\supseteq$ `getRunnableInstances()` $\supseteq$ `getRunningInstances()` and `getRegisteredInstances()` $\supseteq$ `getRunRequestedInstances()` apply to the return sets of these methods. In addition, all instances appearing in the result sets of `getRunnableInstances()` *and* in the result set of `getRunRequestedInstances()` have to appear in the result set of `getRunningInstances()` as well. These relations are depicted in Figure 6.10

[5] This graphical view was especially useful during development and debugging of DAiSI. Moreover it helped Component vendors in research projects where DAiSI was applied to understand why DAiSI establishes a specific Dependable System Configuration during system execution.

However, it is not essential for the execution of Dependable Dynamic Adaptive Systems – this is the reason, why we decoupled this graphical view by introducing a listener concept for our Dependable Configuration Component. This enables us, to exchange this graphical view with a new one or even to disable it.

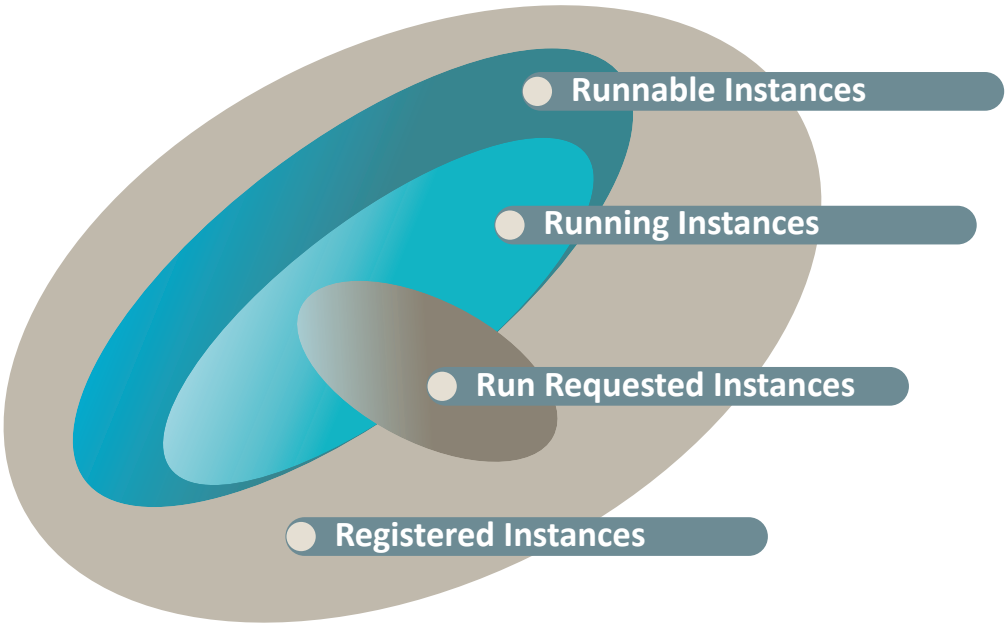


Figure 6.10: Relations Between Result Sets of those Monitoring Methods Returning Information about the Dependable System Configuration.

Next to these four methods another method provides monitoring access to DAiSI's Dependable Configuration Component: `getRegisteredTypes()`. This method returns the types of Dependable Configuration Components present in the Dependable Dynamic Adaptive System.

DAiSI's Dependable Configuration Component derives these types at runtime from the Component instances by comparing the structure of entering Components to the structure of the existing Components. As only structure is considered for the extraction of type information, in our application example, the C-Units provided by the two vendors are considered to be of the same type by DAiSI's Dependable Configuration Component.

Extracting type information in our reference implementation enables the Dependable Configuration Component to reason about syntactical Compatibility. As type information is only retrieved once during registration of a Component, performance is increased.

DAiSI's Dependable Configuration Component does not have to query each Dependable Dynamic Adaptive Component to reason about syntactical Compatibility during each change of the Dependable System Configuration but can extract this information from their types, which means local method calls instead of remote calls. However, this means, that structural changes occurring after registration, which are

possible using our system model, are not considered by DAiSI.

Graphical User Interface

The Configuration Component Browser displays a running Dependable Dynamic Adaptive System for reasons of debugging.

As a prototypical implementation of a monitoring Component, DAiSI features a so called *Configuration Component Browser*. It visualizes the Dependable Dynamic Adaptive System and its Dependable System Configuration graphically. It represents the concepts from our formal system model as follows:

- Dependable Dynamic Adaptive Components are displayed as blue rectangles.
- Dependable Component Configurations are represented by yellow bars within a Component.
- Dependable Services are depicted by green circles, which are connected to providing Dependable Component Configurations.
- Dependable Service References correspond to red semicircles, which are connected to declaring Dependable Component Configurations.
- Uses relations are displayed as connections between Dependable Services and Dependable Service References. A label aligned across this connection indicates, which Service Interface the corresponding Dependable Service implements respectively the corresponding Dependable Service Reference refers to.

A view of a Dependable Dynamic Adaptive System displayed by this Configuration Component Browser is depicted in Figure 6.11. You can see a situation from our application example, where a German and a Dutch C-Unit are bound to a German P-Unit. Both Dependable Service Bindings have been established using Service Interface `PUnitServiceIf`. If you hover a Dependable Service respectively a Dependable Service Reference, the footing of this window will display the Service Interface, it implements respectively refers to – in Figure 6.11 we hovered the Dependable Service provided by the Dutch C-Unit.

You can change the view using a dropdown list, where you can choose between showing only running Dependable Dynamic Adaptive Components– which is the default view – or any other option[6] provided by the monitoring methods of DAiSI's Dependable Configuration Component.

[6] Showing only registered, runnable, or run requested Dependable Dynamic Adaptive Components.

In order to enable you to improve the layout of this view, the Configuration Component Browser supports dragging and dropping of Components, services and service references. Thus, you can arrange the Components to your own needs.

Besides this, you can flip a Component by clicking the green double arrow in the lower right corner of a Component. This causes, that afterwards service references and services are displayed on the opposite side of the Component. This gives you better layouting opportunities. Finally, the Configuration Component Browser supports zooming by the buttons on top of the window.

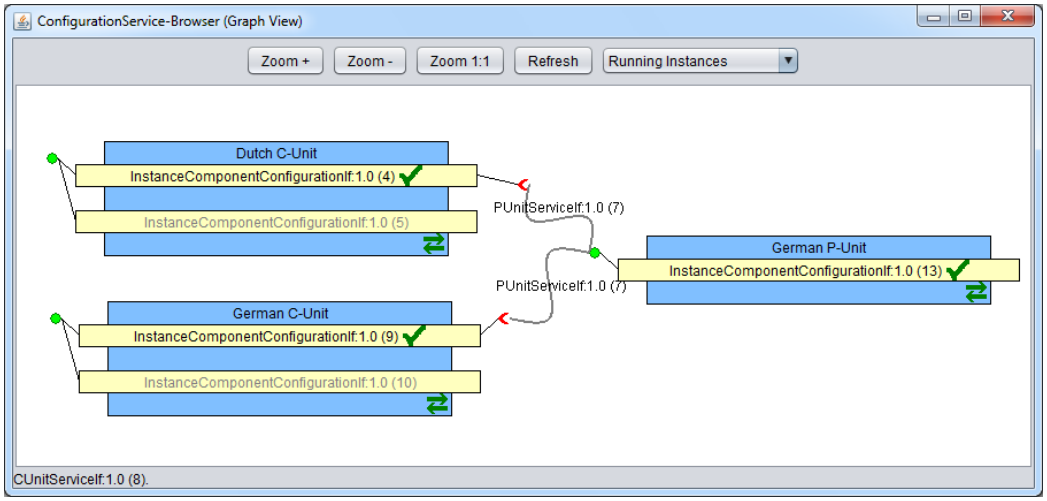


Figure 6.11: A View of our System from the Application Example Using the Configuration Component Browser.

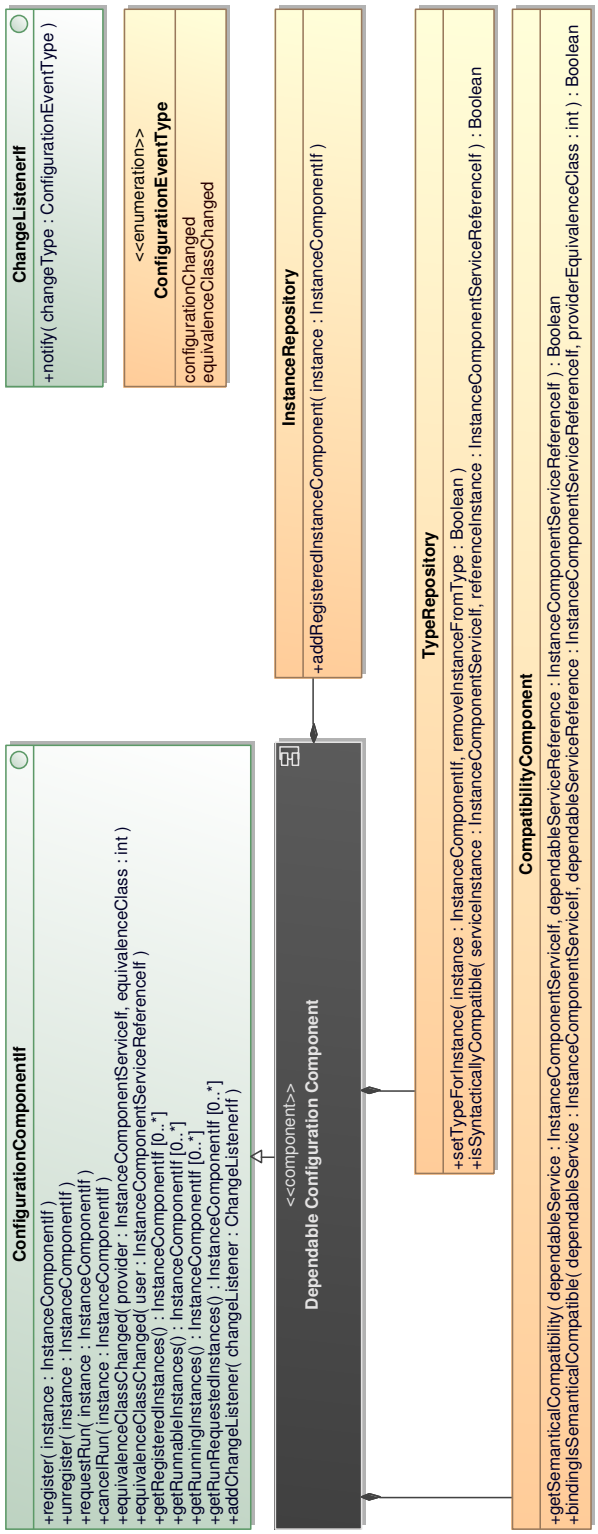


Figure 6.12: Internal Structure of DAiSI's Dependable Configuration Component.

As the Dependable Configuration Component is DAiSI's heartbeat and realizes our approach to Dependable Dynamic Adaptive Systems, we will describe its realization in more detail in the following by looking at some important methods from its public interface.

Realization

The realization of DAiSI's Dependable Configuration Component contains three important parts: An Instance Repository, a Type Repository, and a Compatibility Component, as depicted in Figure 6.12.

The Instance Repository contains all Component instances registered in a Dependable Dynamic Adaptive System.

The Instance Repository is used to store all instances of Dependable Dynamic Adaptive Components, which registered at the Dependable Configuration Component. It stores them in different sets like registered, runnable, running, or run requested instances. These sets can be queried at the Instance Repository separately. Next to this, it provides methods to retrieve a Component, containing a given Dependable Service respectively Dependable Service Reference.

The Type Repository contains types of these instances and realizes the $\simeq_{\text{Syntactical}}$ relation.

The Type Repository manages runtime type information extracted from registering Dependable Dynamic Adaptive Components. It is used by DAiSI's Dependable Configuration Component to reason about syntactical Compatibility of Dependable Dynamic Adaptive Components. The Type Repository, therefore, realizes the $\simeq_{\text{Syntactical}}$ relation of our formal model.

It enables DAiSI's Dependable Configuration Component to find out, which types of Dependable Dynamic Adaptive Components offer a specific Service Interface and to retrieve all instances of them without querying all instances present in the system. This also increases DAiSI's performance during establishing a Dependable System Configuration.

The Compatibility Component triggers the execution of Compliance Test Cases if a Behavior Equivalence Class has changed.

Using its Compatibility Component, the Dependable Configuration Component can query, whether a Dependable Service and a Dependable Service Reference are semantically compatible.

Based on our formal model, the Compatibility Component only calls method `isSemanticallyCompatible` at a Dependable Service Reference, if the Behavior Equivalence Class of this given Dependable Service or of the given Dependable Service Reference has changed. Otherwise it will return the result from the last semantical Compatibility check of this Dependable Service Binding, as the behavior has not changed.

By accessing these three parts, DAiSI's Dependable Configuration Component realizes its public interface. We will have a closer look at two central methods of this interface: `register` and `equivalenceClassChanged`.

They are considered here, as DAiSI's Dependable Configuration Component checks Dependability of a System Configuration within these methods. A call to the `register` method corresponds to trigger *AppComponentsChange* while a

call to the `equivalenceClassChanged` requires us to check, whether trigger CompatibilityChange is active and, therefore, the Dependable System Configuration needs to be changed.

Method `register` is realized as follows: First of all, the registering Dependable Dynamic Adaptive Component is added to the registered instances at the Instance Repository and a type is set for it at the Type Repository.

By doing this, the syntactical Compatibility to existing Components is also derived at the Type Repository. Figure 6.8 referred to this by reference "*Derive syntactical Compatibility*".

register = set a type, figure out a runnable configuration, and activate it.

Then the Dependable Configuration Component checks, whether the registering Component is runnable (meaning that at least one of its Dependable Component Configurations can be activated). If this is the case, it is additionally added to the set of runnable Components at the Instance Repository, before the Dependable System Configuration is updated by calling `updateSystemConfiguration()`. This corresponds to reference "*Activate best Dependable Component Configuration (this includes a check of semantical Compatibility)*" in Figure 6.8.

The overall registration process is depicted in Figure 6.13.

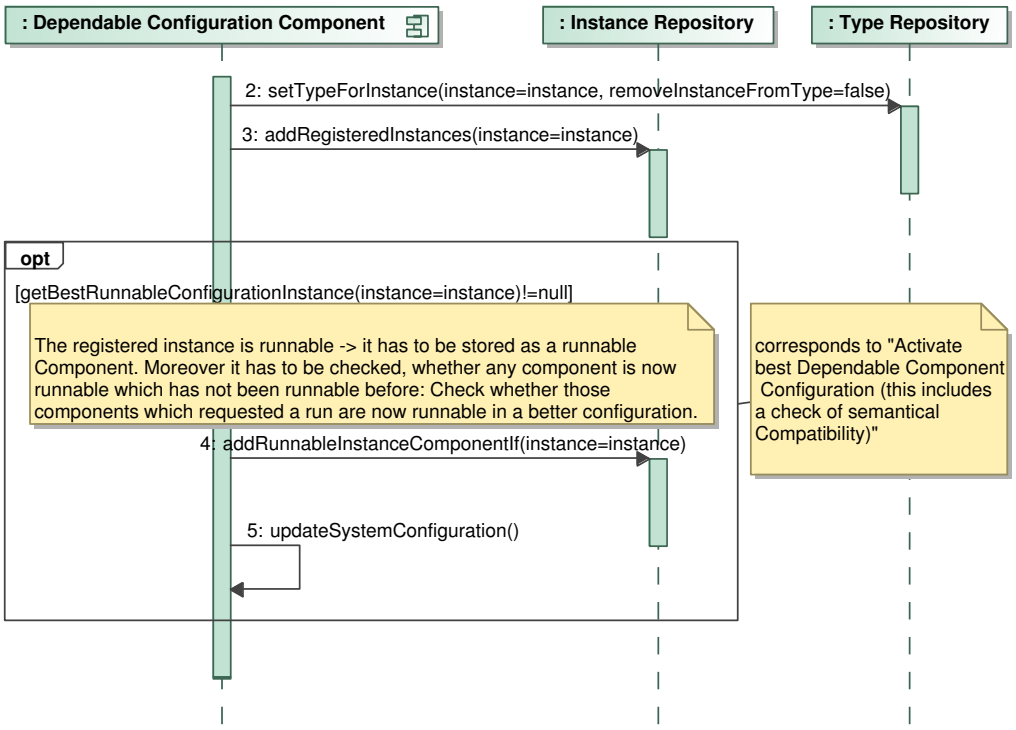


Figure 6.13: Realization of the `register` Method by DAiSI's Dependable Configuration Component.

Method `getBestRunnableConfigurationInstance` checks, which is the best runnable Dependable Component Configuration. It does this by checking for each configuration whether a syntactically compatible Dependable Service is provided by another Component in a runnable configuration for each of the configuration's Dependable Service References. It then returns the best runnable configuration.

Method `updateSystemConfiguration()` is realized as follows: First of all the Dependable Configuration Component checks, whether any Dependable Dynamic Adaptive Component, which has not been runnable before, is runnable now. It marks them as runnable, so they can be started up when updating the Dependable System Configuration at the end.

Then it checks, whether any Dependable Dynamic Adaptive Component which requested to run is runnable now. If it finds such a Component, it immediately activates it. As a consequence it may also activate further runnable but not yet running Dependable Dynamic Adaptive Components which are required by this Component.

In the following it updates the Dependable Component Configurations of running Components by calling `adaptComponentConfiguration()`, as now better configurations may be activatable. As other Components may not be used anymore, due to this reconfiguration, in the following unused Components are stopped[7] by calling `stopUnusedComponents`. Finally, all listeners are notified, that the Dependable System Configuration has changed and they, therefore, need to update their views.

Summed up, the realization of this method is depicted in Figure 6.14.

unregister = stop Component and update the System Configuration.

The `unregister` method calls `updateSystemConfiguration()` as well, after it has stopped the unregistering Component and removed all Component Bindings, where this component was involved.

Whenever DAiSI's Dependable Configuration Component establishes a new Dependable Service Binding, it calls the `equivalenceClassChanged` method, passing the Dependable Service and its current Behavior Equivalence Class. Next to this, this method is called by Dependable Dynamic Adaptive Components, when their internal state changes and, therefore, they may behave differently now.

To understand, how Dependability is addressed within DAiSI's Dependable Configuration Component, we will look at the realization of this method. This method is called, whenever a BehavioralChange trigger may be active, as a Behavior Equivalence Class has changed, or a new Component Binding has been established. We assume, that Dependable Dynamic Adaptive Components notify the Dependable Configuration Component of internal state changes – thus, each active BehavioralChange trigger can be recognized within this method. The realization of

[7] As long as they did not call `requestRun()`.

Figure 6.14: Realization of the `updateSystemConfiguration` Method by DAiSI's Dependable Configuration Component.

this method is depicted in Figure 6.15 – it represents reference "*Monitor changes and update the System Configuration in case of an active Change Trigger*" depicted in Figure 6.8.

Whenever a Behavior Equivalence Class has changed, the semantical Compatibility needs to be updated – the System Configuration needs to be updated if a (In-)Compatibility has been detected.

Whenever the `equivalenceClassChanged` method is called, DAiSI's Dependable Configuration Component queries all instances of Dependable Service References, which refer to the Service Interface implemented by the given Dependable Service (interactions 2 – 7 of Figure 6.15).

In the following it uses the CompatibilityComponent to evaluate for each of these Dependable Service References, whether they have been semantically compatible before the Behavior Equivalence Class has changed. It then updates the semantical Compatibility by calling the `updateSemanticalCompatibility` method at the CompatibilityComponent and compares the newly calculated semantical Compatibility to the previous Compatibility.

This enables DAiSI's Dependable Configuration Component to recognize,

whether a previously semantically incompatible set of Dependable Service and Dependable Service Reference is semantically compatible now or vice versa. Thus, it is a realization of the BehavioralChange trigger from our formal model. If DAiSI's Dependable Configuration Component recognizes, that this trigger is active, it updates the Dependable System Configuration by calling the `updateSystemConfiguration` method.

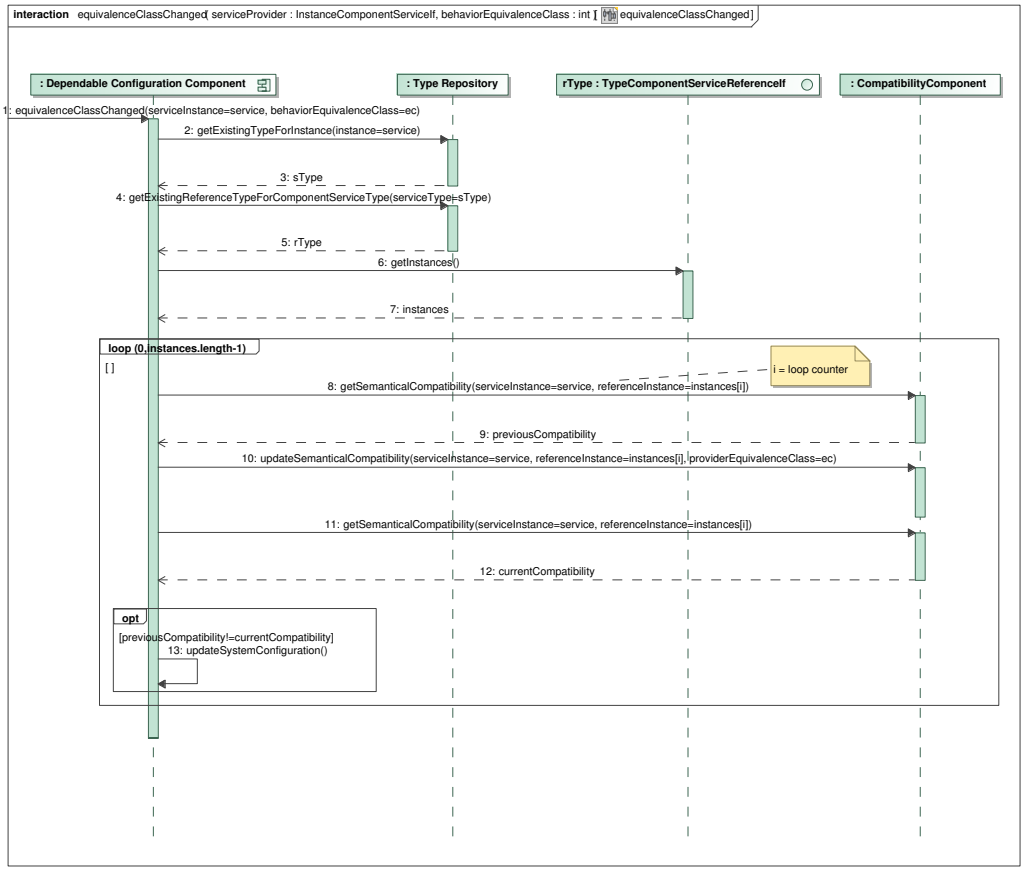


Figure 6.15: Realization of the `equivalenceClassChanged` Method by DAiSI's Dependable Configuration Component.

To update semantical Compatibility, DAiSI's Dependable Configuration Component calls `bindingIsSemanticalCompatible` for the specific Service Binding and checks, whether the result contradicts the previous result. If this is the case, the new result is stored as sketched in Figure 6.16. If a semantical Compatibility is stored, this Dependable Service Binding can be established in the following by DAiSI's Dependable Configuration Component, otherwise it is removed from the

Dependable System Configuration.

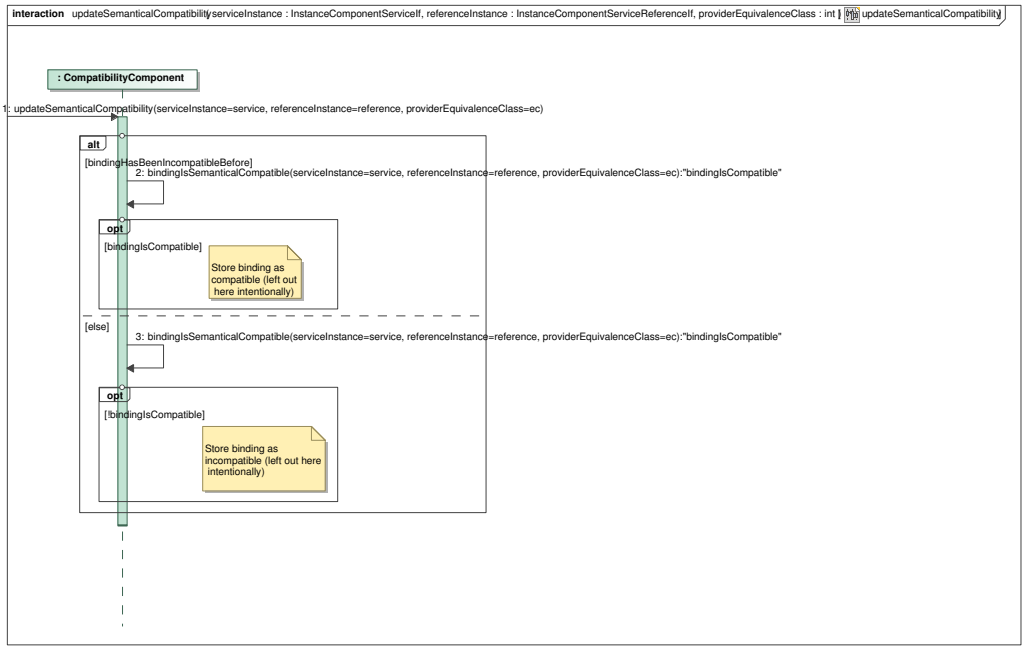


Figure 6.16: Realization of the `updateSemanticalCompatibility` Method by DAiSI's Dependable Configuration Component.

Method `bindingIsSemanticalCompatible` is implemented specially due to our approach of applying runtime tests to decide, whether a Service Binding is semantically compatible: since we need to trigger execution of runtime tests here, we need to we need to take measures against side-effects. This includes side-effects on a running system as well as side-effects on Service Partners involved in the test (this may be Service Partners, which are not part of the running system, yet).

We put affected Service Partners into testing mode to avoid side-effects of test case execution.

To avoid side-effects on Service Partners involved in the test, these Service Partners are put into testing mode, where they can store their internal state. This enables them to recover their state after test case execution.

To avoid side-effects of test execution on a running system, we need to put further Dependable Dynamic Adaptive Components into testing mode. To understand, which Dependable Dynamic Adaptive Components need to be put into testing mode, we will look at a simple scenario in the following.

Let us examine a test case execution from the system of our application example. At $t_0 + 6$ we want to bind the Dutch P-Unit to the Dutch C-Unit. Thus, we need to evaluate, whether Dependable Service $\text{pUnitService}_{\text{Dutch}}$ is semantically compatible to Dependable Service Reference $\text{pUnitReference}_{\text{Dutch}}$. Therefore, we query the

current Behavior Equivalence Classes of these two Service Partners.

In the following we want to execute the test case defined by the Dutch C-Unit for its current Behavior Equivalence Class regarding the Dependable Component Binding. Before we can do this, we need to put the two Service Partners into testing mode. Besides them we have to put all other Service Partners in testing mode, which are bound to them in a direct or indirect way.

Affected = bound directly or transitively.

Two Service Partners *A* and *B* are directly bound, iff A uses a Dependable Service provided by B or vice versa. An indirect binding means, that they are bound transitively by multiple Service Partners in a row. All these bound Service Partners need to be put into testing mode, since they may be affected by test case execution and, therefore, need to recover their internal state when the test has finished.

Let's consider the system from our application example to get a deeper understanding. Figure 6.17 depicts the situation at $t_0 + 6$ again. If we test the Dependable Component Binding between $\text{pUnitService}_{\text{Dutch}}$ and $\text{pUnitReference}_{\text{Dutch}}$, we first of all need to put the two Dependable Dynamic Adaptive Components into testing mode, which provide Dependable Service $\text{pUnitService}_{\text{Dutch}}$ respectively declare Dependable Service Reference $\text{pUnitReference}_{\text{Dutch}}$. In our example these are pUnitDutch respectively cUnitDutch.

Next to these two Dependable Dynamic Adaptive Components, we need to put all Dependable Dynamic Adaptive Components into testing mode, which are directly or indirectly bound to them. We can derive them by following the uses relations in Figure 6.17.

When considering the pUnitDutch, we find out, that the cUnitDutch as well as the mUnitDutch are bound to it. When considering the cUnitDutch, we find out that the pUnitDutch and the mUnitDutch are bound to it.

Summing up, next to pUnitDutch and cUnitDutch we need to put mUnitDutch into testing mode, as it is bound to at least one Dependable Dynamic Adaptive Component under test as depicted in Figure 6.18. We call these Dependable Dynamic Adaptive Components bound to Components under test *affected Components* in the following.

Test cases are executed as depicted in Figure 6.19. Putting affected Components into testing mode is illustrated in this figure by the call *setTestMode(true)*[8]. After we prepared the test by putting the affected Components into testing mode, we proceed with testing the specific behavior of this Dependable Service.

Test case execution includes method calls at the Dependable Service under test as well as internal calls at the Requesting Service Partner, which may be used to reason about the semantical Compatibility. This enables us to decide, whether the

[8] We leave out putting transitively affected Components into testing mode here for better readability of the sequence chart.

test has passed or failed.

Idea: Compliance Test Cases are specified in a test specification language; Compliance Test Cases' code is generated from this specification.

Regarding our application example, at $t_n + 1$ we want to bind the second Dutch C-Unit to the second German P-Unit. Thus, Dependable Service $\text{pUnitService}_{\text{German}}2$ provided by the second German P-Unit is initially tested by executing Compliance Test Cases, which have been defined by the second Dutch C-Unit for the declared Dependable Service Reference $\text{pUnitReference}_{\text{Dutch}}2$. In the following we are looking at this test case, which is executed at $t_n + 1$.

Let us consider a test case specification like the one depicted as TTCN-3 specification in Listing 6.2 respectively as UML Testing Profile specification in Figure 6.20.

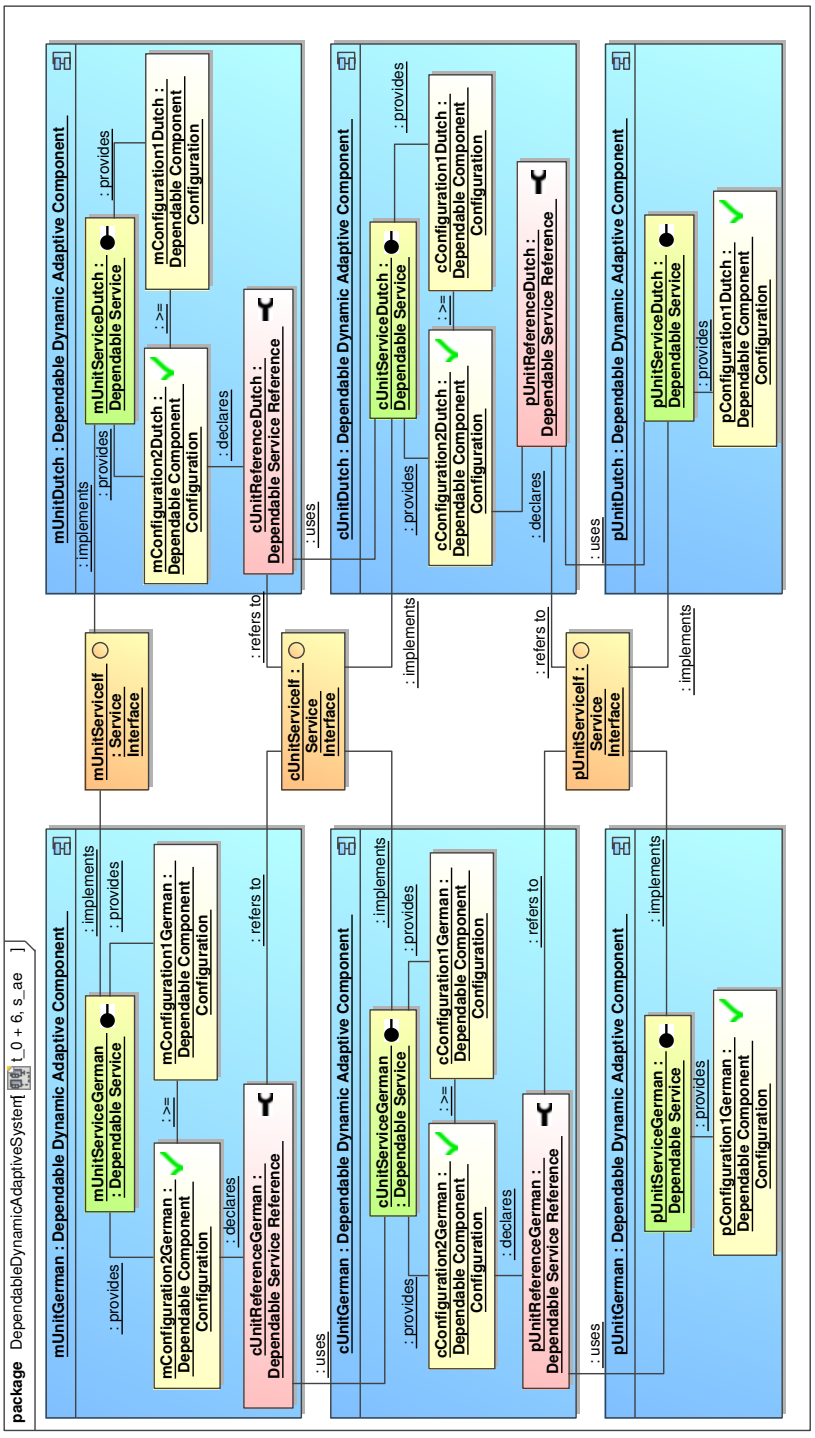


Figure 6.17: Looking Back at the Bindings from our Application Example at $t_0 + 6$.

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

Listing 6.2: TTCN-3 Specification of a Testcase for P-Units as Defined by the Dutch C-Unit.

To be executable, this test case specification needs to be transformed into executable code. All code examples in this thesis are Java-based. Thus, Listing 6.3 depicts, how a Java fragment, generated from these specifications, may look like. This can be generated directly into the code of the Dependable Dynamic Adaptive Component acting as Requesting Service Partner[9] and can be executed by our Dependable System Infrastructure to reason about semantical Compatibility of a Dependable Component Binding involving the specific Requesting Service Partner.

```
1 public boolean testSemanticalCompatibility(EquivalenceClass
    equivalenceClass, PeripheralUnitIf pUnit) {
2  if(equivalenceClass == TriageClass.EX) {
3    int pulseRate = pUnit.getPulseRate();
```

[9] In this case this is the second Dutch C-Unit

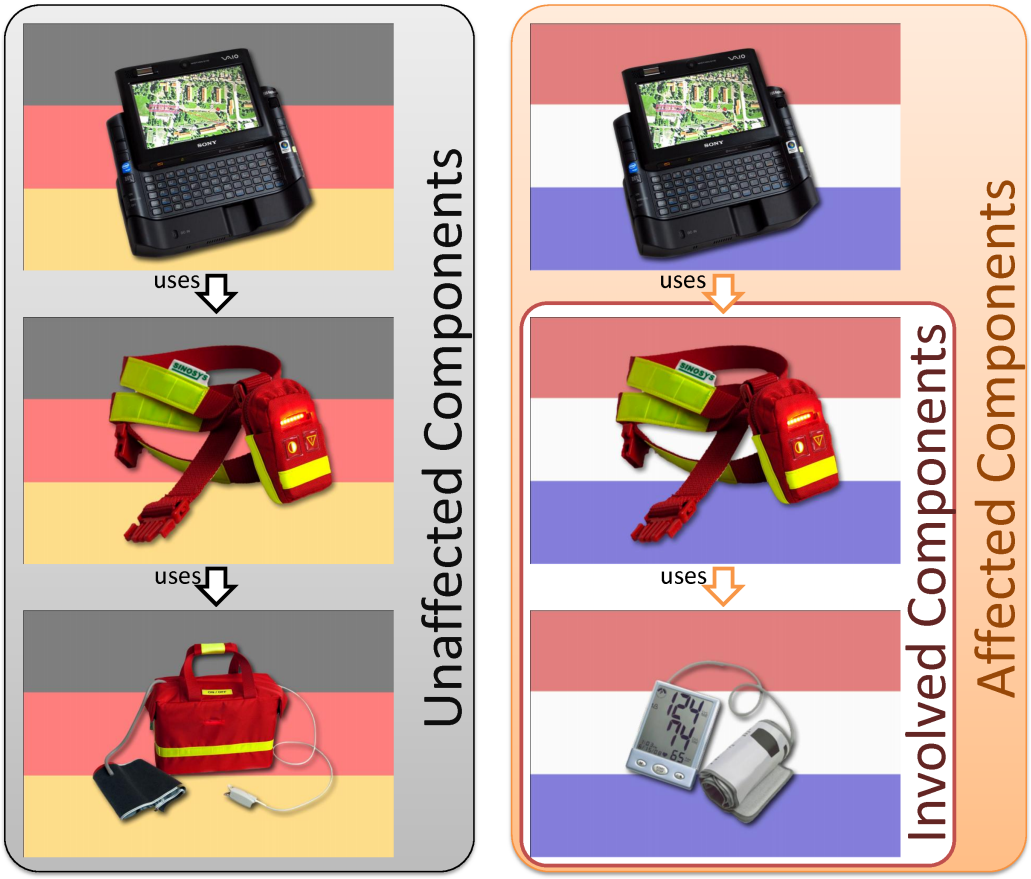


Figure 6.18: A View on a System Under Test at $t_0 + 6$. We Can Distinguish *Affected* From *Unaffected* Components Based on the Components Involved in the Test.

```
4     int systolicBloodPressure = pUnit.getSystolicBloodPressure
          ();
5     int diastolicBloodPressure = pUnit.
          getDiastolicBloodPressure();
6     if (pulseRate != 0 || systolicBloodPressure != 0 ||
          diastolicBloodPressure != 0) {
7       return false;
8     } else {
9       return true;
10    }
11  } else {
12    if(equivalenceClass == TriageClass.T_I) {
13      int pulseRate = pUnit.getPulseRate();
14      int systolicBloodPressure = pUnit.
            getSystolicBloodPressure();
15      int diastolicBloodPressure = pUnit.
```

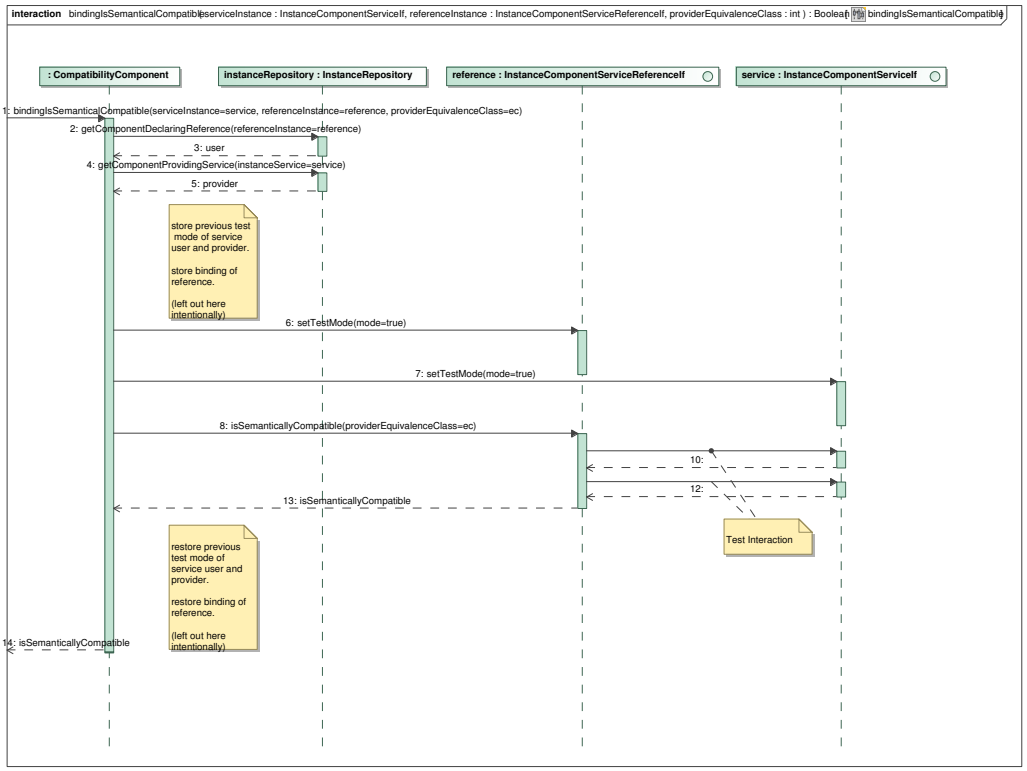


Figure 6.19: Realization of the `bindingIsSemanticalCompatible` Method by DAiSI's Dependable Configuration Component.

```
            getDiastolicBloodPressure();
16        if (0<pulseRate<300 && 0<systolicBloodPressure<300 && 0<
            diastolicBloodPressure<300) {
17          return true;
18        } else {
19          return false;
20        }
21      } else {
22        [...]
23      }
24    }
25  }
```

Listing 6.3: Java-Code fragment generated from a Testcase Specification of a Testcase for P-Units as Defined by the Dutch C-Unit.

If we assume, that the Triage Class of a casualty is calculated to *T_I* at $t_n + 1$, the second option for the test case is executed. Thus, the blood pressure as well as pulse rate are queried and it is checked, whether they are between 0 and 300.

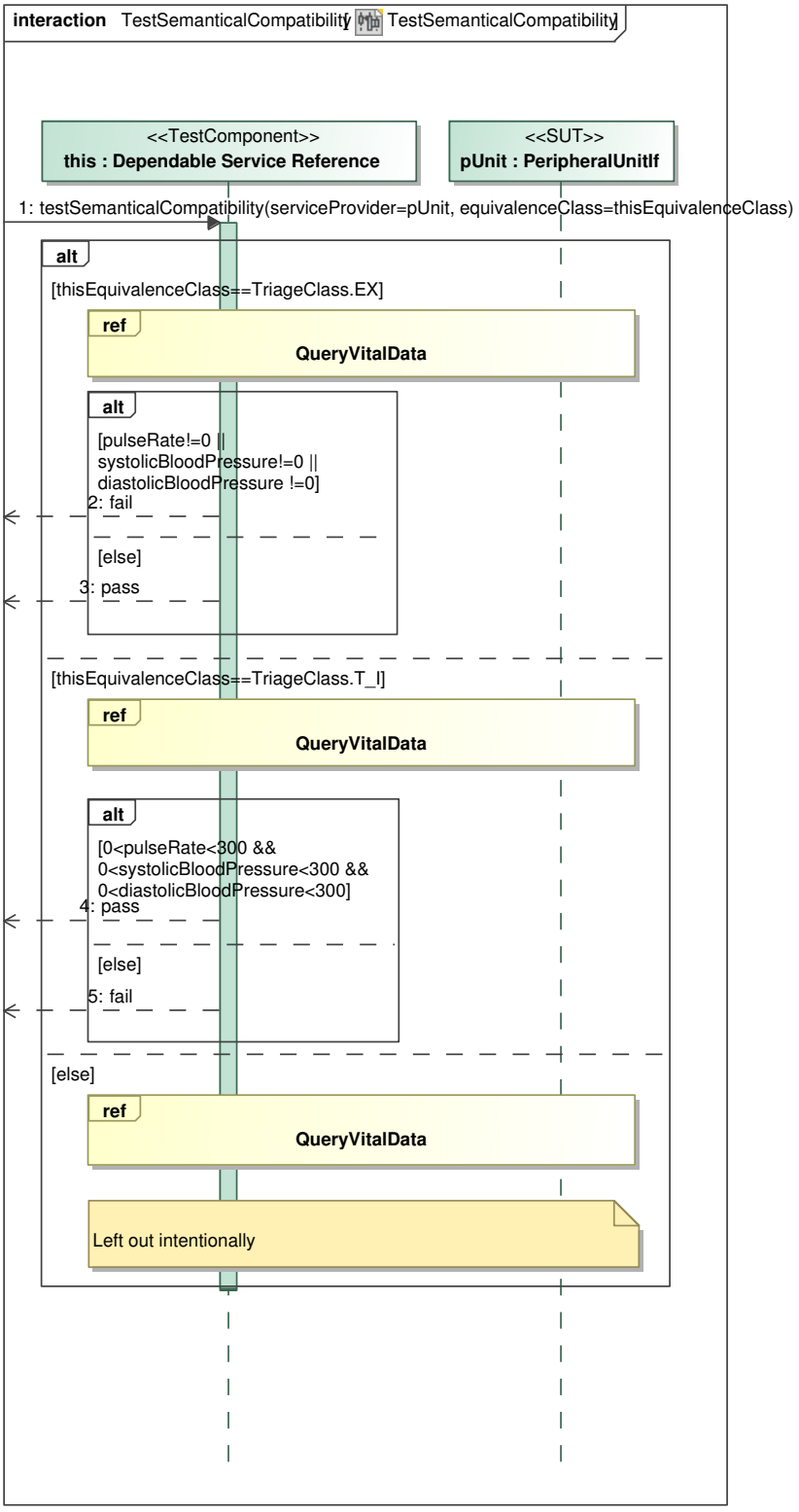


Figure 6.20: UML Testing Profile Specification of a Testcase for P-Units as Defined by the Dutch C-Unit.

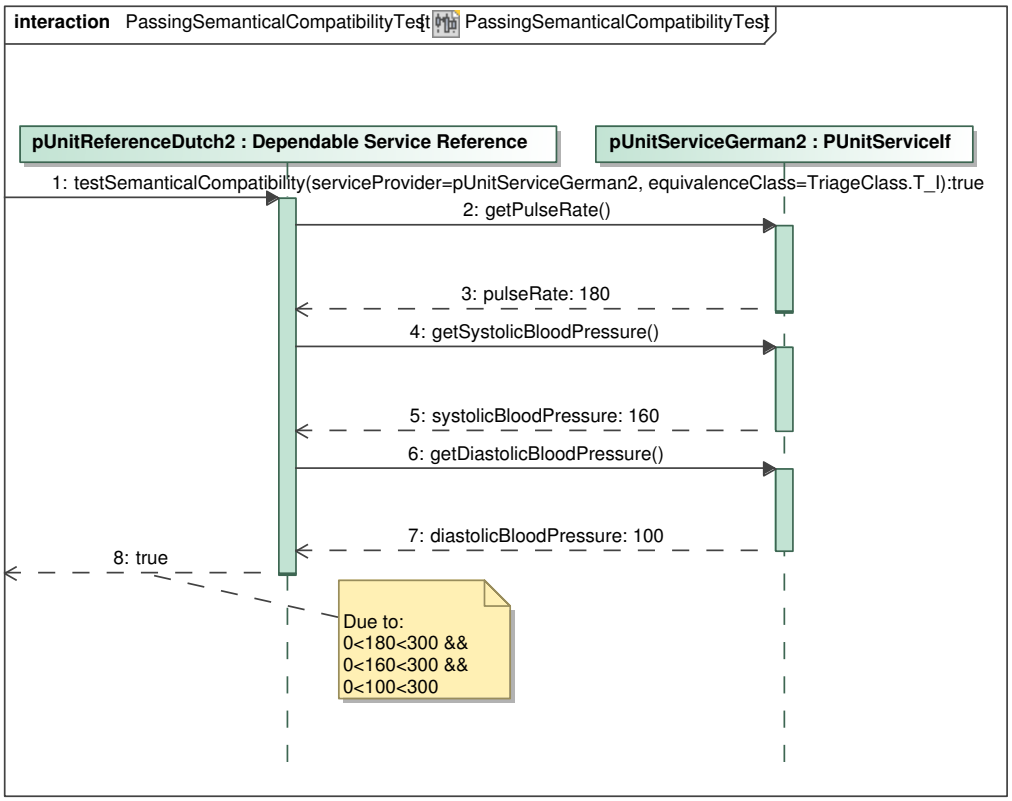


Figure 6.21: Execution of Test Cases to Reason About Semantical Compatibility Before Binding the second Dutch C-Unit to the second German P-Unit at $t_n + 1$.

If the casualty has, for example, a pulse rate of 180 bpm and a blood pressure of (160, 100) as depicted in Figure 6.21, therefore, the test passes and these two Service Partners are bound by the Dependable Configuration Component.

Semantical Compatibility + monitoring of Behavior Equivalence Classes = BehavioralChange trigger.

Summed up, this semantical Compatibility test in combination with monitoring the Behavior Equivalence Classes of all Service Partners within a Dependable Dynamic Adaptive System is our realization of the BehavioralChange trigger from our formal model: We detect changes in Behavior Equivalence Classes by monitoring and determine, whether they influence Dependability by runtime testing.

If we look at our application example at $t_n + 2$ we are, therefore, able to recognize, that the second Dutch C-Unit and the second German P-Unit are not semantically compatible anymore[10]: Our Dependable System Infrastructure will

[10] $t_n + 2$ denotes the Dependability Checkpoint when the fingerclip measuring the pulse rate slips off.

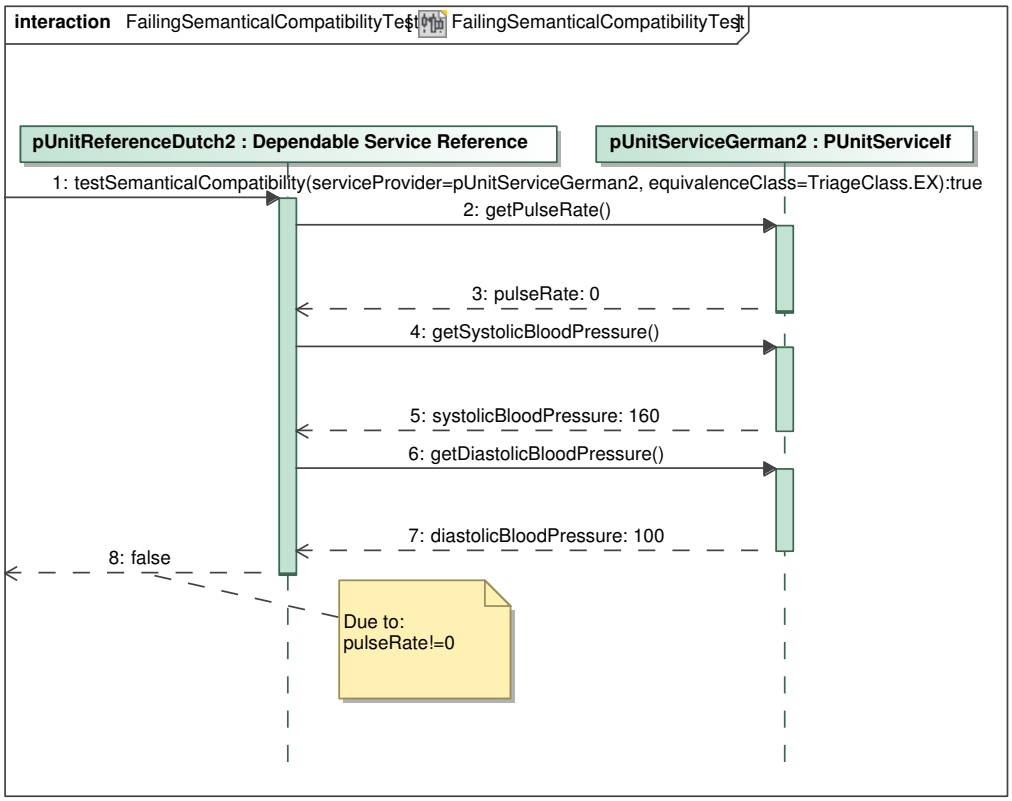


Figure 6.22: Execution of Test Cases to Reason About Semantical Compatibility After Binding the second Dutch C-Unit to the second German P-Unit at $t_n + 2$ when the fingerclip slips off.

recognize a changing Behavior Equivalence Class of the second Dutch C-Unit towards `TriageClass.EX` and, therefore, repeat the semantical Compatibility tests.

We can detect the semantical incompatibility in our application example, using our approach.

The second Dutch C-Unit will query the vital data of the second German P-Unit. It will recognize, that this vital data values – pulse rate equals zero, while blood pressure is above zero – deviate from its expectations. Thus, test case execution fails. In detail, this test case execution will happen as depicted in Figure 6.22.

As a consequence the Dependable Configuration Component will remove this Dependable Component Binding between the second Dutch C-Unit and the German P-Unit. Thus, the casualty will not be classified incorrectly but needs to be classified manually by a medic instead.

6.2 Component Framework

Our component framework enables vendors to focus on Component functionality instead of dealing with specifics of our system model.

To simplify implementation of Dependable Dynamic Adaptive Components, our reference implementation of DAiSI also provides a Component framework. Component vendors can use this framework in order to deal with Component specific implementation details instead of having to deal with details specific to our formal system model.

The Component framework realizes the Component model from our formal system model by following an interface-based approach. Therefore, all elements of the Component model are represented by interfaces as depicted in Figure 6.23.

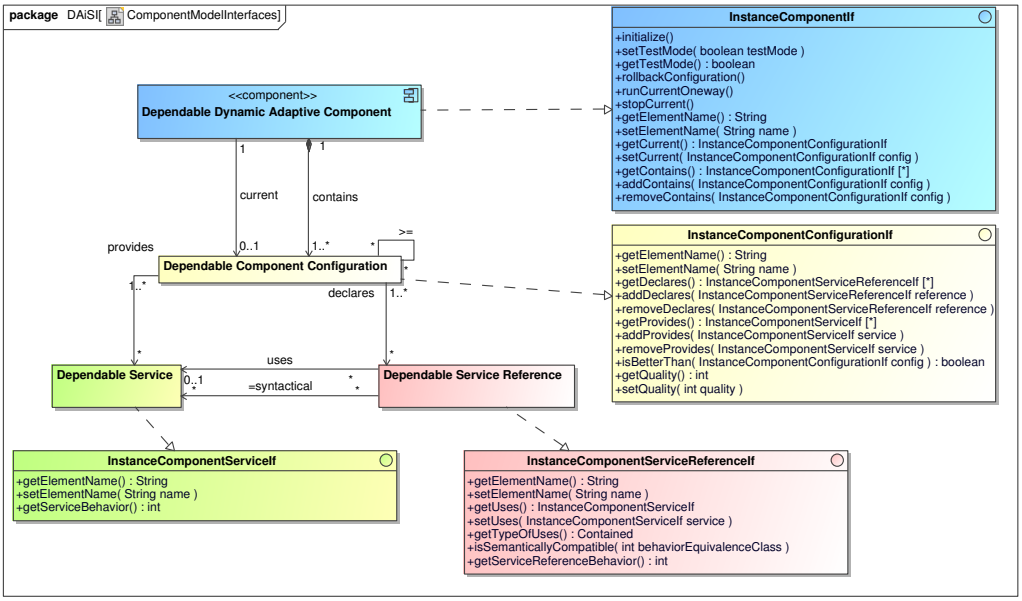


Figure 6.23: Mapping Between the Component Model of our Formal System Model and Interfaces Used by our Reference Implementation DAiSI.

The relations between elements of our Component model are represented in those interfaces as well. You can access all contained Dependable Component Configurations by calling `getContains()` at `interface InstanceComponentIf`, for example. However, there are differences between these interfaces and our Component model. These differences split up into two categories:

- Simplifications, which simplified the implementation of the Dependable System Infrastructure.
- Additions, which added debugging capabilities to the Dependable System

Infrastructure without complicating the implementation of Dependable Dynamic Adaptive Components.

A simplification was to calculate the semantical Compatibility only for active Dependable Service Bindings. This causes, that the Dependable Configuration Component will establish a Service Binding, evaluate the semantical Compatibility and remove it in case of a detected incompatibility (as this means, that it is not a Dependable Service Binding).

Simplification: semantical Compatibility is only calculated for active bindings.

This simplification becomes visible in `interface InstanceComponentServiceReferenceIf` at two places: on the one hand, the `getServiceReferenceBehavior()` method does not take an `InstanceComponentServiceIf` as an argument, as it only returns the Behavior Equivalence Class of the currently used Dependable Service. On the other hand, it is visible at the `isSemanticallyCompatible(int behaviorEquivalenceClass)` method, which only takes the Behavior Equivalence Class of the Dependable Service Reference (regarding the currently used Dependable Service) instead of taking the Dependable Service that should be tested in addition.

As a design decision we realized Behavior Equivalence Classes simply by int values. As our formal model only consider changes of a Behavior Equivalence Class this is no restriction of the formal model but a design decision for the realization of our Dependable System Infrastructure. This is visible at the return values of the `getServiceBehavior()` respectively `getServiceReferenceBehavior()` methods of the interfaces `InstanceComponentServiceIf` respectively `InstanceComponentServiceReferenceIf`.

Behavior Equivalence Classes realized simply by integers.

An addition was, to include a string description for each element of the model in the interfaces, which can be accessed by the `getElementName()` respectively `setElementName(String name)` methods. This enables us to display readable names for the elements in our Dependable Dynamic Adaptive System browser instead of hardly understandable object identifiers.

Addition: String descriptions to enable monitoring Components to generate human readable visualizations.

These interfaces are implemented by helper classes of our Component framework as depicted in Figure 6.24. Next to a pure interface realization the helper classes provide added features enabling Component vendors to realize Dependable Dynamic Adaptive Components more easily.

Using these helper classes a Component vendor can implement a Dependable Dynamic Adaptive Component as depicted in Figure 6.25.

By letting a newly developed Dependable Dynamic Adaptive Component extend the class *AbstractInstanceComponentImpl*, a vendor can benefit from framework methods capable of creating Dependable Component Configurations, Dependable Service References, or Dependable Services and adding them to this

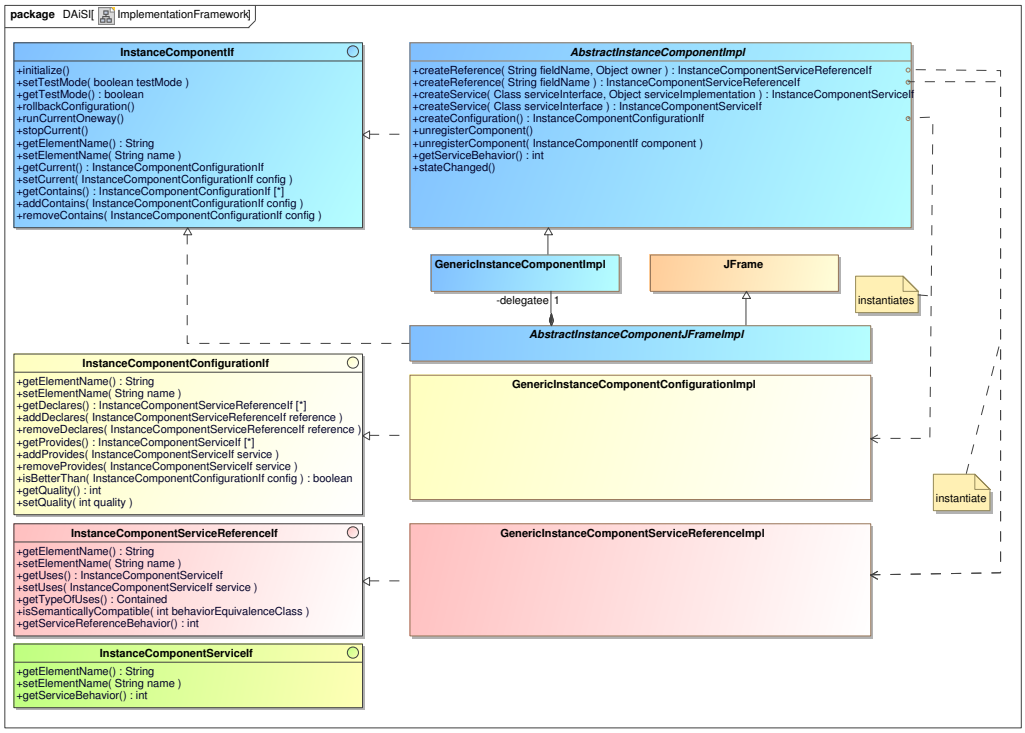


Figure 6.24: Helper Classes of DAiSI's Component Framework Implementing the Interfaces Which Represent our Component Model.

Component. The methods provided by `AbstractInstanceComponentImpl` are briefly sketched in the following.

First of all, it implements helper methods to create remote objects representing the Dependable Dynamic Adaptive Components and their structure. This enables DAiSI's Dependable Configuration Component to call methods on the Dependable Dynamic Adaptive Components and their Dependable Component Configurations respectively provided Dependable Services and declared Dependable Service References remotely.

One of these methods is `createConfiguration()`. It enables us to create a remote object representing an empty Dependable Component Configuration. In the following we can add Dependable Services and Dependable Service References to this configuration and finally add this configuration to the supported configurations of our Dependable Dynamic Adaptive Component by passing it as a parameter to the `addContains` method of this Component.

Another method is `createService`. This method is implemented using two different parameter lists. The first takes a `Class` parameter and an `Object` parameter. The `Class` parameter specifies the Service Interface, for which a

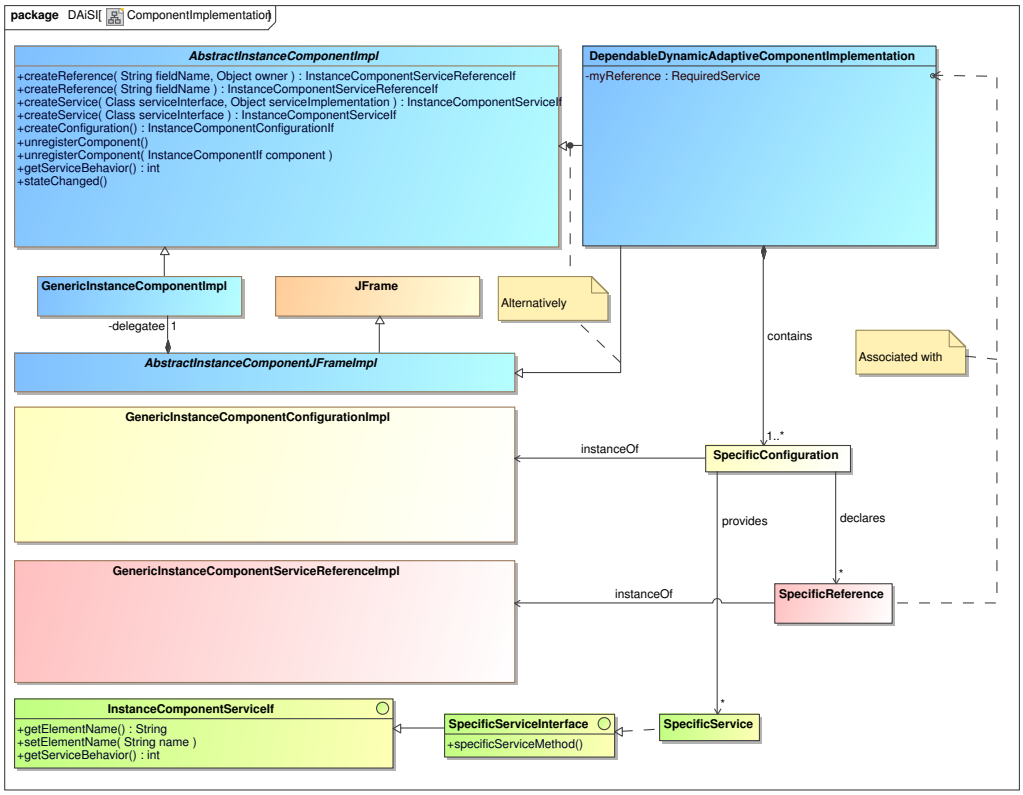


Figure 6.25: Implementation of a Dependable Dynamic Adaptive Component by Using the Component Framework.

remote object should be created (as Dependable Dynamic Adaptive Components may provide multiple Service Interfaces). The `Object` parameter has to be the Object realizing the given Service Interface.

The second `createService` method only takes a `Class` parameter and assumes, that the given Service Interface is realized by the Dependable Dynamic Adaptive Component (`this`) itself.

A third method provided by `AbstractInstanceComponentImpl` is `createReference`. This method creates a remote object for a Dependable Service Reference. It also is provided with two different parameter lists. The first takes a `String` parameter and an `Object` parameter. The `String` parameter specifies the name of the field representing the Dependable Service Reference in our Dependable Dynamic Adaptive Component implementation.

A specific Dependable Service will be assigned to the given field by DAiSI's Dependable Configuration Component using the reflection mechanism. The `Object`

parameter has to be the Object containing the field with the given name[11]. The second `createReference` method only takes a `String` parameter and assumes, that the Dependable Dynamic Adaptive Component (`this`) itself contains a field of the given name.

Next to these methods, three additional methods are provided by `AbstractInstanceComponentImpl`. The method `unregisterComponent` can be used, to unregister a Dependable Dynamic Adaptive Component at DAiSI's Dependable Configuration Component. As a consequence, the Dependable Dynamic Adaptive Component is removed from the Dependable Dynamic Adaptive System by removing existing Dependable Service Bindings containing this Component.

This `unregisterComponent` method also comes with two parameter lists, one taking a `InstanceComponentIf`, which represents the Dependable Dynamic Adaptive Component to be unregistered, the other taking no parameter, assuming that the Dependable Dynamic Adaptive Component (`this`) should be unregistered.

Default implementation of getServiceBehavior for downwards compatibility to prior versions of DAiSI.

Method `getServiceBehavior` is provided for downwards compatibility. As DAiSI has been applied in several research projects and first versions of DAiSI did not take semantical Compatibility into account, we provide a implementation of the `getServiceBehavior` method always returning -1. This enables us to execute Dependable Dynamic Adaptive Components on the latest version of DAiSI without adding the calculation of Behavior Equivalence Classes to their implementation[12].

Component vendors need to override method `getServiceBehavior` method respectively `getServiceReferenceBehavior` to insert their specific calculation of a Behavior Equivalence Class[13].

Only state-preserving methods may be called during calculation of a Behavior Equivalence Class.

First of all, we need to understand, which methods of a Service Interface they may use during calculation of a current Behavior Equivalence Class of a Dependable Service respectively of a Dependable Service Reference. Thus, we can classify methods of a Service Interface regarding the influence of method execution on the internal state of a Service Partner as follows: we can distinguish between *state-preserving* and *state-changing* methods.

Considering our application example, C-Units offer Service Interface `CasualtyUnitIf` as depicted in the Domain Architecture in Figure 6.26. In the following we will classify the methods of this Service Interface according to the previously mentioned dimension in Table 6.1.

[11] This enables us to use, for example, a field from an inner class of the Dependable Dynamic Adaptive Component as Dependable Service Reference.

[12] However, we need to override this method with our own calculation of Behavior Equivalence Classes to fully benefit from our approach to Dependable Dynamic Adaptive Components.

[13] How they can provide own implementations of `getServiceBehavior` respectively `getServiceReferenceBehavior`, is explained in Appendix B.

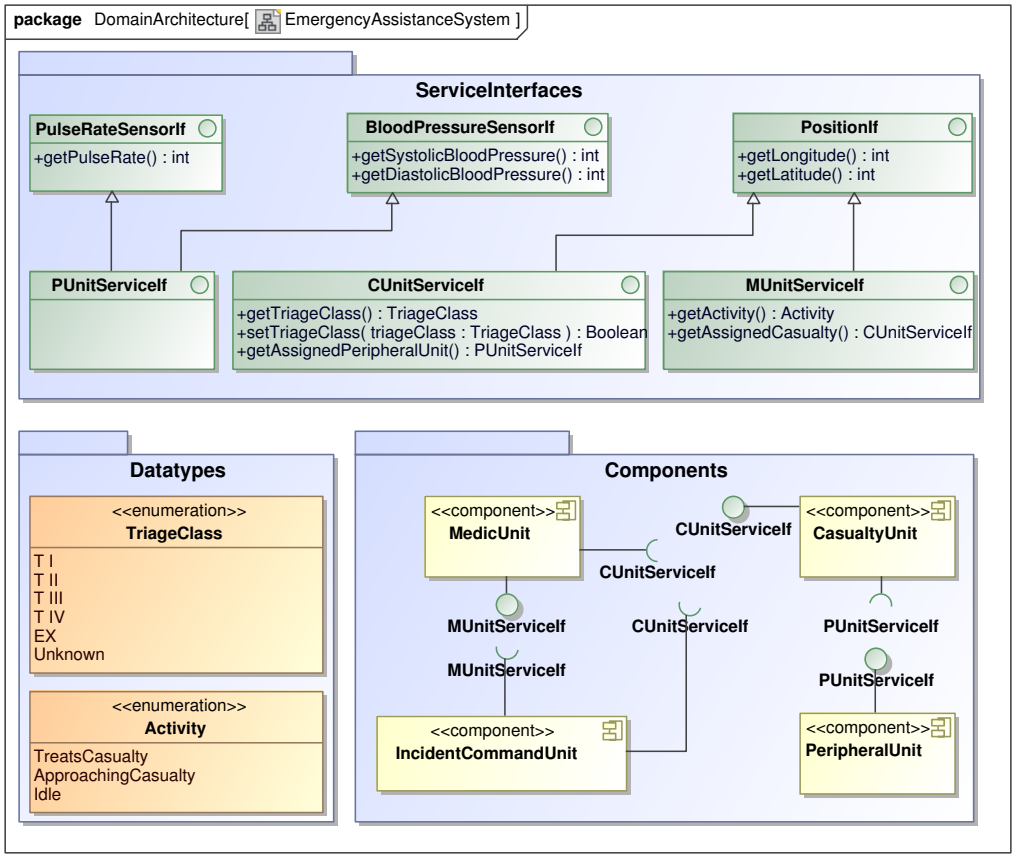


Figure 6.26: Looking Back at our Domain Architecture for Emergency Assistance Systems.

You can see, that getters belong to the category "*state-preserving*" while setters belong to the category "*state-changing*". During calculation of Behavior Equivalence Classes, only state-preserving methods may be called, as otherwise the Behavior Equivalence Class may be changed during calculation. Moreover the calculation would not be free of side-effects otherwise.

When looking at Service Interface `PUnitServiceIf` provided by P-Units in our application example, we find out, that all methods of this Service Interface are state-preserving and, therefore, can be called during calculation of Behavior Equivalence Classes.

Method classification may be part of a Domain Architecture.

Such a classification of methods may be part of a Domain Architecture to give vendors of Dependable Dynamic Adaptive Components an additional clue, how implementations of these methods should behave and which methods they may call during calculation of Behavior Equivalence Classes. Next to this, mechanisms like

influence on internal state	method

state-preserving	`getLongitude`
	`getLatitude`
	`getTriageClass`
	`getAssignedPeripheralUnit`

state-changing	`setTriageClass`

Table 6.1: Classification of Methods Contained in Service Interface `CUnitServiceIf` Along the Dimension *Influence of Method Execution on Internal State*.

marking it by an annotation [Chi05] can enable implementations of a Dependable System Infrastructures to ensure, that only state-preserving methods are called during calculation of Behavior Equivalence Classes.

Considering our framework, method `stateChanged()` notifies DAiSI's Dependable Configuration Component that the state of this Dependable Dynamic Adaptive Component has changed and, therefore, the Behavior Equivalence Classes for all Dependable Service Bindings of this Component need to be recalculated.

In case of changed Behavior Equivalence Classes DAiSI's Dependable Configuration Component will execute Compliance Test Cases for the corresponding Dependable Service Bindings to evaluate their semantical Compatibility.

We assume, that Dependable Dynamic Adaptive Components call this method, whenever their internal state changes. It causes calls of the `equivalenceClassChanged` method at the Dependable Configuration Component for each provided Dependable Service and for each declared Dependable Service Reference of their Current Configuration.

Component vendors need to extend `AbstractInstanceComponentImpl` in order to benefit from these helper methods. Thus, they would be limited, since Java does not support multiple inheritance. To enable Component vendors to provide Dependable Dynamic Adaptive Components extending any specific class, we provide the sample class *AbstractInstanceComponentJFrameImpl*, demonstrating how to do this by extending the specific class *JFrame*.

Inheriting own classes is possible by using the Delegation pattern.

If a vendor wants to provide a Dependable Dynamic Adaptive Component extending a specific class *SuperClass*, he needs to implement an abstract class, extending SuperClass. This class also needs to implement interface *InstanceComponentIf* in order to provide access to the methods from our framework described above.

Component vendors do not need to implement the interface – they can use the Delegation pattern [GHJV94] instead. In this case the abstract class acts as delegator, whereas an internally instantiated object of *GenericInstanceComponentImpl* acts as delegatee. Now they can implement a Dependable Dynamic Adaptive

Component extending SuperClass by extending the newly created abstract class, which extends SuperClass.

You will find a more detailed guideline, how vendors can implement Dependable Dynamic Adaptive Components using our Component framework in Appendix B, where we describe the implementation of specific Components from our application example.

In the following, we will investigate, which tool support exists for developers to speed up the implementation of Dependable Dynamic Adaptive Components and to lower the burden of applying our approach.

6.3 Tool Support During Implementation

IDEs like Eclipse already offer powerful mechanisms to support Component developers during realization of Dependable Dynamic Adaptive Components. One of the most important mechanisms during implementation is code completion. A study in 2005 showed, that it is the 5th most used editing command of the Eclipse IDE [MKF06] exceeded only by the basic commands Delete, Save, Next Word, and Paste. If you are not familiar with code completion you can find an overview of it in [BMM09].

Eclipse code completion templates enable quick implementation of Dependable Dynamic Adaptive Components.

Within the Eclipse IDE, Content Assist is responsible for code completion. We realized templates for Content Assist, providing four new code completions, namely *createConfiguration*, *createReference*, *createService*, and *createTestRelatedMethods*. These templates are depicted in Listing 6.4.

The template *createConfiguration* can be used to create a Dependable Component Configuration and placeholders for required and provided Dependable Services. The created configuration is added to the contained configuration of the currently edited Dependable Dynamic Adaptive Component. The Component vendor can select the Dependable Component Configuration which should be created by using a dropdown list offering all declared attributes of the Type `InstanceComponentConfigurationIf`, which is the representation of a Dependable Component Configuration within DAiSI's framework.

Templates *createService* and *createReference* are used to add provided respectively required Dependable Services to a previously created Dependable Component Configuration. The configuration to which the Dependable Service respectively Dependable Service Reference should be added once again can be chosen using a dropdown list.

In case of *createService* the Component developer needs to specify the Service Interface and a name for the created field representing the Dependable Service[14]. When applying *createReference*, the Component developer can select the

[14]This field can be reused to add the same Dependable Service to further Dependable Compo-

field (which needs to extend `InstanceComponentServiceIf`) representing the Dependable Service Reference from a dropdown list.

The last template *createTestRelatedMethods* can be used to generate a getter for the Behavior Equivalence Class of a Dependable Service Reference and a method calculating the semantical Compatibility for a Dependable Service Reference. Within the latter method a Component developer needs to implement a test case, testing the semantical Compatibility of a bound Dependable Service. The Dependable Service Reference, for which these methods should be generated can be once again chosen from a dropdown list as already sketched for the *createReference* template.

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <templates>
3    <template autoinsert="true" context="java" deleted="false"
        description="Create a Configuration and add it to the
        Component" enabled="true" name="createConfiguration">
4     ${:import(de.tuc.ifi.sse.daisi.componentModel.
        instanceComponent.InstanceComponentConfigurationIf)}
5     ${configurationName:field(de.tuc.ifi.sse.daisi.
        componentModel.instanceComponent.
        InstanceComponentConfigurationIf)} =
        createConfiguration();
6     // Create and add Provided Services here
7     ${cursor}
8     // Create and add Service References for Required
        Services here
9     /** The Configuration is added to the Component.
10     * Note, that the order of adding the configurations
         decides, which
11     * Configuration is treated as "better" by DAiSI.
12     * Configurations added earlier are treated as better.
13     */
14    addContains(${configurationName});
15   </template>
16   <template autoinsert="true" context="java" deleted="false"
       description="Creates a Service Reference for a required
        Service and adds it to a Configuration of this
        Component." enabled="true" name="createReference">
17    ${configurationName:field(de.tuc.ifi.sse.daisi.
        componentModel.instanceComponent.
        InstanceComponentConfigurationIf)}.addDeclares(
        createReference("${referenceName:field(de.tuc.ifi.sse.
        daisi.componentModel.instanceComponent.
        InstanceComponentServiceIf)}"));
18   </template>
19   <template autoinsert="true" context="java" deleted="false"
```

nent Configurations.

```
        description="Creates a Service and adds it to a
        Configuration of a Component" enabled="true" name="
        createService">
20    ${ServiceInterface} ${serviceName} = (${ServiceInterface
        })createService(${ServiceInterface}.class, this);
21    ${configurationName:field(de.tuc.ifi.sse.daisi.
        componentModel.instanceComponent.
        InstanceComponentConfigurationIf)}.addProvides(${
        serviceName});
22  </template>
23  <template autoinsert="true" context="java" deleted="false"
        description="Creates a getter for the behavior of a
        Service Reference and a method testing the semantical
        equivalence of the service reference." enabled="true"
        name="createTestRelatedMethods">
24    public int getServiceReferenceBehavior_${
        referenceName:field(de.tuc.ifi.sse.daisi.
        componentModel.instanceComponent.
        InstanceComponentServiceIf)}(){
25      // TODO: access ${referenceName} to calculate the
          current Behavior Equivalence Class.
26      return 0;
27    }
28    public boolean equivalenceClassTest_${referenceName}(int
        providerEquivalenceClass, int userEquivalenceClass){
29      // TODO: access ${referenceName} to check, whether it is
          semantically equivalent
30      return true;
31    }
32  </template>
33 </templates>
```

Listing 6.4: Code Completion Templates for the Eclipse IDE.

6.4 Summary

Six tables show the relation between realization and formal model and restrictions of the realization.

Within this section we depict the relation between DAiSI and our formal model introduced in the previous sections by six tables. We will relate basic sets, relations and reconfiguration triggers from our model to implementation groups and implementation elements.

While implementation groups specify an area, where this model element is relevant, implementation elements describe, how this model element is realized within this implementation group.

Wherever applicable, we will explicitly state within these tables, which restrictions our reference implementation imposes regarding our formal model.

Model	Reference Implementation	
Model Element	Group	Implementation Element
DependableDynamicAdaptiveComponent	Component Framework	all Instances extending AbstractInstanceComponentImpl
DependableComponentConfiguration	Component Framework	all Instances of GenericInstanceComponentConfigurationImpl
DependableService	Component Framework	all Instances implementing an Interface, which extends InstanceComponentServiceIf
DependableServiceReference	Component Framework	all Instances of GenericInstanceComponentServiceReferenceImpl
ServiceInterface	CORBA	all Interfaces definable in IDL
MethodDeclaration	CORBA	all Methods definable in IDL
AttributeDeclaration	CORBA	all Attributes definable in IDL
Type	CORBA	all Types definable in IDL
String	Programming Language	java.lang.String

Figure 6.27: Mapping Structural Sets Defined in Our Formal System Model to Elements of Our Reference Implementation.

Model	Reference Implementation		
Model Element	Group	Implementation Element	Restrictions
ApplicationComponents	Infrastructure	DependableConfigurationComponent.getRegisteredInstances()	
Contains	Component Framework	AbstractInstanceComponentImpl.getContains()	1
≥	Component Framework	AbstractInstanceComponentImpl	1, 2
Current	Component Framework	AbstractInstanceComponentImpl.getCurrent()	3
Provides	Component Framework	GenericInstanceComponentConfigurationImpl.getProvides()	1
Declares	Component Framework	GenericInstanceComponentConfigurationImpl.getDeclares()	1
Uses	Component Framework	GenericInstanceComponentServiceReferenceImpl.getUses()	3
Implements	Programming Language	Not explicitly defined. Derived by reflection at runtime.	
RefersTo	Component Framework	GenericInstanceComponentServiceReferenceImpl.getTypeOfUses()	
≈Syntactical	Programming Language	Not explicitly defined. Derived by reflection at runtime.	4
Methods	Programming Language	Not explicitly defined. Could be derived by reflection at runtime.	5
Attributes	Programming Language	Not explicitly defined. Could be derived by reflection at runtime.	5
Parameters	Programming Language	Not explicitly defined. Could be derived by reflection at runtime.	5
MethodName	Programming Language	Not explicitly defined. Could be derived by reflection at runtime.	5
ReturnType	Programming Language	Not explicitly defined. Could be derived by reflection at runtime.	5
AttributeName	Programming Language	Not explicitly defined. Could be derived by reflection at runtime.	5
AttributeType	Programming Language	Not explicitly defined. Could be derived by reflection at runtime.	5
≈InterfaceSyntactical	Programming Language	Not explicitly defined. Could be derived by reflection at runtime.	5
≈MethodSyntactical	Programming Language	Not explicitly defined. Could be derived by reflection at runtime.	5
≈ParameterSyntactical	Programming Language	Not explicitly defined. Could be derived by reflection at runtime.	5
≈AttributeSyntactical	Programming Language	Not explicitly defined. Could be derived by reflection at runtime.	5

Legend:	
1	Static - Can not change over time. If a component needs to change it, it needs to unregister, change the relation and register again. Otherwise, DAiSI's DependableConfigurationComponent will not detect the change.
2	Considering only the integer quality value (getQuality()) of configurations, respectively the order of adding configurations, if no quality value has been set.
3	May only be changed by DAiSI's Dependable Configuration Component.
4	Only identical Types of implemented / referred to interfaces are treated as identical - no comparison of methods, attributes, or parameters.
5	Not considered, as the reference implementation treats services and service references as syntactical compatible, iff they implement respectively refer to the same interface.

Figure 6.28: Mapping Structural Relations Defined in Our Formal System Model to Elements of Our Reference Implementation.

Model	Reference Implementation		
Model Element	Group	Implementation Element	Restrictions
AppComponentsChange	Infrastructure	ConfigurationComponent.(un-)register	
ConfigurationChange	n/a	n/a	1
ConfigOrderChange	n/a	n/a	1
CurrentConfigurationChange	Infrastructure	Adaptation of Current Configuration performed by ConfigurationComponent	2
ServiceChange	n/a	n/a	1
ReferenceChange	n/a	n/a	1
BindingChange	Infrastructure	Adaptation of Bindings performed by ConfigurationComponent	2

Legend:	
1	Changes at runtime are not detected unless a component unregisters, changes the relation and registers again.
2	Adaptations, which are not performed by the ConfigurationComponent are not detected.

Figure 6.29: Mapping Structural Reconfiguration Triggers Defined in Our Formal System Model to Elements of Our Reference Implementation.

Model	Reference Implementation	
Model Element	Group	Implementation Element
BehaviorEquivalenceClass	Programming Language	int

Figure 6.30: Mapping Behavioral Sets Defined in Our Formal System Model to Elements of Our Reference Implementation.

Model	Reference Implementation		
Model Element	Group	Implementation Element	Restrictions
SystemBehavior	n/a	n/a	1
ComponentBehavior	n/a	n/a	1
ConfigurationBehavior	n/a	n/a	1
ProvidedBehavior	n/a	n/a	1
DeclaredBehavior	n/a	n/a	1
BindingBehavior	Infrastructure	Combined by the ConfigurationComponent by querying the Behavior of Service and Service Reference.	
ServiceBehavior	Service Implementation	Method getServiceBehavior() - needs to be realized by a Component vendor.	
ServiceReferenceBehavior	Component Implementation	Method getServiceReferenceBehavior_<referenceName>() - needs to be realized by a Component vendor.	2
≈Semantical		Method equivalenceClassTest_<referenceName>(int providerEC, int userEC) - needs to be realized by a Component vendor.	
isDependable	n/a		3

Legend:	
1	Semantical Compatibility is only considered for Service Bindings. Thus only the behavior of a service and a service reference are considered within the reference implementation.
2	Only defined within the reference implementation for a currently established binding instead of passing a Dependable Service as a parameter to the method.
3	Not defined within the reference implementation, as DAiSI will only establish semantically compatible Service Bindings and thus this relation is always true.

Figure 6.31: Mapping Behavioral Relations Defined in Our Formal System Model to Elements of Our Reference Implementation.

Model	Reference Implementation		
Model Element	Group	Implementation Element	Restrictions
IncompatibilityChange	Infrastructure	ConfigurationComponent checks the semantical Compatibility, whenever equivalenceClassChanged(InstanceComponentServiceIf dependableService, int serviceEquivalenceClass) is called	1
CompatibilityChange	Infrastructure	ConfigurationComponent checks the semantical Compatibility, whenever equivalenceClassChanged(InstanceComponentServiceIf dependableService, int serviceEquivalenceClass) is called	1

Legend:	
1	Semantical Compatibility check is triggered by the Service Provider by calling stateChanged; Service User cannot trigger the compatibility check.

Figure 6.32: Mapping Behavioral Reconfiguration Triggers Defined in Our Formal System Model to Elements of Our Reference Implementation.

And now the end is near; and so I face the final curtain.
My friend, I'll say it clear, I'll state my case of which I'm certain.
I've lived a life that's full. I've travelled each and every highway;
and more, much more than this, I did it my way.

Frank Sinatra

7 Summary

Dependable Dynamic Adaptive Systems gain importance.

Nowadays, Dynamic Adaptive Systems are gaining importance rapidly, as you can see by research directions like Ubiquitous Computing, Ultra Large Scale Systems, or IT Ecosystems and numerous conferences or workshops, dealing with these research questions associated with these systems. As indicated by the name, two concepts characterize Dynamic Adaptive Systems: Dynamics and Adaptation.

Dynamics = Components enter or leave the system at runtime.

These systems are dynamic, meaning that Components may enter or leave the system at any time during runtime. To fully benefit from dynamics, Dynamic Adaptive Systems are open system, meaning that external Service Interfaces of their Components are published in a common Domain Architecture enabling Component vendors to provide new Components for these systems.

Components are not known upfront.

These Components need to be integrated into those systems, although they have not necessarily been known upfront. Thus, classical design of an overall system, explicitly specifying the System Configuration containing its Components and their bindings, is not applicable for Dynamic Adaptive Systems anymore.

Systems and Components are adaptive regarding a System Context.

Dynamic Adaptive Systems are adaptive, meaning that they (respectively their Components) may behave differently according to a current context of the system. Thus, they adapt their behavior to fully benefit from entering Components, provide graceful degradation in case of leaving Components, or to adapt to a user's current needs.

A dynamically changing System Configuration is necessary.

As motivated, runtime reconfiguration, which means changing the System Configuration including its Component Bindings, is necessary for Dynamic Adaptive Systems since Components may enter or leave a system at runtime. However, proving

Semantical Compatibility needs to be considered.

the correctness of a Component Binding at runtime is not possible in general. Nevertheless it is crucial to deal with semantical Compatibility of Component Bindings, as (re-)binding Dynamic Adaptive Components at runtime imposes huge threats to a system's Dependability.

7.1 Conclusion

Our system model defines syntactical and semantical Compatibility.

Starting with an application example from the emergency management domain, we analyzed Dependable Dynamic Adaptive Systems to gain a deep understanding of these systems. This resulted in a formal system model, enabling us to reason about Compatibility within these systems. We defined two aspects of Compatibility within our model: syntactical and semantical Compatibility.

Syntactical Compatibility only considers a structural comparison of Dependable Services provided by Components and Dependable Service References specifying required Dependable Services of other Components. Semantical Compatibility compares provided and required behavior of Components instead.

A Dependable Dynamic Adaptive System must not contain semantically incompatible bindings.

We elaborated an approach evaluating the semantical Compatibility of Component Bindings during establishing a System Configuration. This enables a Dependable System Infrastructure to establish only Dependable System Configurations free of semantically incompatible Component Bindings. We call Dynamic Adaptive Systems containing only semantically compatible Component Bindings *Dependable Dynamic Adaptive Systems*.

A Component Binding may consist of several Service Bindings between a Service User and a Service Provider. The semantical Compatibility is evaluated separately for each of these Service Bindings resulting in a composed semantical Compatibility of the Component Binding.

$\simeq_{\text{Semantical}}$ defines semantical Compatibility.

Semantical Compatibility in our formal model is defined by the $\simeq_{\text{Semantical}}$ relation. As semantical Compatibility of a Component Binding may change over time due to changes in state of the bound Components, the evaluation of this relation depends on the state of the bound Components. Thus, our model introduces so called Dependability Checkpoints. They specify points in time during system execution, where the Dependability of Component Bindings need to be reevaluated.

Dependability Checkpoints denote points, where the Dependability is threatened.

Behavior Equivalence Classes are used to reduce points, where semantical Compatibility needs to be reevaluated.

In order to get a reasonable set of Dependability Checkpoints for a Component Binding at runtime, our model introduces the concept of Behavior Equivalence Classes. By using Behavior Equivalence Classes we try to avoid unnecessary evaluations of semantical Compatibility of Component Bindings by recognizing critical changes of a Component's state, which are relevant for the semantical Compatibility of a Component Binding.

Behavior Equivalence Classes specify state spaces, where a Dependable Service behaves equivalently respectively a Dependable Service Reference expects,

that a used Dependable Service behaves equivalently. Thus, they are defined by each Dependable Service as well as by each Dependable Service Reference.

The relations ServiceBehavior respectively ServiceReferenceBehavior return the current Behavior Equivalence Class of a Dependable Service respectively a Dependable Service Reference bound to a specific Dependable Service at any time during runtime by accessing state-preserving methods of the specific Dependable Service and reasoning about their return values.

As long as a Behavior Equivalence Class does not change, equivalent behavior is provided respectively expected.

Behavior Equivalence Classes of Dependable Service References express, that a specific behavior is expected by a Service Provider. This specific behavior needs to be considered during the realization of the $\simeq_{\text{Semantical}}$ relation when implementing our formal system model.

By monitoring changes in Behavior Equivalence Classes, a Dependable System Infrastructure can reevaluate semantical Compatibility of a Service Binding, whenever the Behavior Equivalence Class of a bound Dependable Service or Dependable Service Reference changes. If it detects a semantical incompatibility, it can instantly remove this specific Service Binding from the System Configuration, which prevents method calls at a semantically incompatible Dependable Service and, therefore, increases the system's Dependability.

Changes of a Behavior Equivalence Class trigger a reevaluation of semantical Compatibility.

We provided a reference implementation DAiSI of such a Dependable System Infrastructure. DAiSI updates the System Configuration while it considers semantical Compatibility. Therefore, it implements our formal model introduced before.

DAiSI updates a Dependable System Configuration based on our formal model.

DAiSI uses runtime testing to reason about semantical Compatibility of Service Bindings. A Service User may specify different Compliance Test Cases depending on a currently active Behavior Equivalence Class of this Service Binding to deal with different expected behaviors depending on the Components' states. The Compliance Test Cases of the currently active Behavior Equivalence Class are executed on Service Providers at runtime to reason about the semantical Compatibility of a specific Service Binding.

DAiSI uses testing – a lightweight, well known, mechanism.

Compliance Test Cases are defined by Service Users.

Since tests are executed on Dependable Dynamic Adaptive Components bound to a running system, we need to take care, that testing does not have side-effects on the system's behavior. Thus, we notify all Dependable Dynamic Adaptive Components, which are transitively bound to the Component under test in order to warn them, that they must not rely on the behavior of other Service Providers at the moment. In consequence we also notify them, when testing has finished in order to enable them, to roll back state changes, which mistakenly occurred during test case execution.

Side-effects are addressed by introducing a testing mode.

An implementation framework enables Component vendors to easily implement Dependable Dynamic Adaptive Components which adhere to our formal model. Eclipse templates for code completion complement the formal model as additional tool support for Component vendors.

Our implementation framework enables vendors to implement Dependable Dynamic Adaptive Components quickly.

We implemented our application example using DAiSI. This implementation

serves as a guideline for Component vendors, how they can implement Dependable Dynamic Adaptive Components using our reference implementation of a Dependable System Infrastructure. The implementation of this application example shows, that DAiSI can detect semantical incompatibilities at runtime and remove the specific Service Bindings.

Our approach is application independent.

DAiSI does not know anything about the Dependable Dynamic Adaptive Components except that they adhere to our formal model. Thus, DAiSI can be used to realize and execute Dependable Dynamic Adaptive Components for any Dependable Dynamic Adaptive System.

We applied for patent together with Siemens.

Together with Siemens we applied for a patent on our approach to Dependable Dynamic Adaptive Systems using Behavior Equivalence Classes to (re-)evaluate semantical Compatibility of Component Bindings [NRK+10].

7.2 Outlook

Composition is not part of our model, yet.

In the future we may investigate several issues regarding our approach presented in this thesis. First of all, our formal model does not provide composition, meaning that two Components can be bound together forming a single Component and, therefore, encapsulating those Service Interfaces used for their Component Binding. Today composition is used at development time to encapsulate inner details of Component realizations. We need to investigate, whether encapsulation is required at runtime as well to prevent multiple Service Bindings of different Dependable Service References to the same Dependable Service.

The impact of our approach on a engineering methodology needs to be investigated.

Next to this, our proposed approach will have an impact on the engineering methodology for Dependable Dynamic Adaptive Systems respectively their Components. A comparison between an engineering methodology today and a novel engineering methodology using our approach is depicted in Figure 7.1.

Today system vendors select Components from a Component market and add self-developed Components which they may publish in the Component market as well. They compose the resulting Component selection to a system and verify and validate this system[1]. When they successfully verified and validated this system, they deliver it to different end users, which will use this system in the following.

Activities like verification shift from development time towards runtime.

Different system vendors use this methodology to produce systems based on Components from the same Component market leading to a huge diversity of systems available for end users. This methodology is depicted on top of Figure 7.1.

Using our approach, several parts of this engineering methodology shift from development time towards runtime of the system. First of all, there is no system vendor anymore. Instead there are only Component vendors, which implement Com-

No system vendor anymore.

[1] Of course, this is no one-way waterfall process – loops are not displayed in this Figure to increase readability.

ponents by referring to interfaces jointly defined by a standardization committee in a common Domain Architecture. In addition they may verify their Components against semantical specifications, which may exist in this Domain Architecture[2].

End users integrate the system (semi-)automatically by using a Dependable System Infrastructure.

After publishing these Components in a Component market, end users may select Components from this market and deploy them within any system using a Dependable System Infrastructure. This infrastructure is responsible for composition of the system, which means it needs to update the System Configuration regarding the newly entering Components. During this update of the System Configuration, the infrastructure considers semantical Compatibility by verifying and validating[3] the Component Bindings.

Since Components change their internal states and, therefore, their provided respectively expected behavior, this semantical Compatibility needs to be permanently reevaluated, whenever a change in a Behavior Equivalence Class is recognized by monitoring. If the semantical Compatibility of a Component Binding changes, the Dependable System Infrastructure needs to update the System Configuration immediately.

Users want to know, whether a Component fits into their system, before they buy it.

This engineering methodology is depicted at the bottom of Figure 7.1. We need to evaluate, whether this sketch of an engineering methodology is appropriate for Dependable Dynamic Adaptive Systems, or whether it needs to be further refined. For example, it may be necessary to inform a user about current syntactical and semantical Compatibility of Components available in a Component market to an existing Dependable Dynamic Adaptive System in order to enable him to select fitting Components.

Test case generation needs to be considered to lower the burden of our approach.

For Component developers, we need to consider the integration of test case generation into our framework. This enables them to engineer their test cases in a systematical way by specifying them in languages like TTCN-3 or U2TP. However, there is already a bunch of related work in the area of test case generation from models. Thus, the integration of test cases is not a research challenge.

Regarding test case generation one could also think of generating test case automatically from execution traces collected during executions at development time by Component vendors. This would lower the burden when applying our approach to Dependable Dynamic Adaptive Systems. However, we need to be able to extract the relevant traces to generate test cases to apply this type of test case generation.

[2] This is not depicted in the Figure to increase readability

[3] It depends on the test cases, whether the evaluation of semantical Compatibility is verification, validation, or both. If test cases are derived from a semantical specification of the Service Interface contained in the Domain Architecture, they represent verification. If they are defined by a Service User to express his expectations of the behavior of this Dependable Service, they represent validation. In consequence a combination using both types of test cases means performing validation and verification.

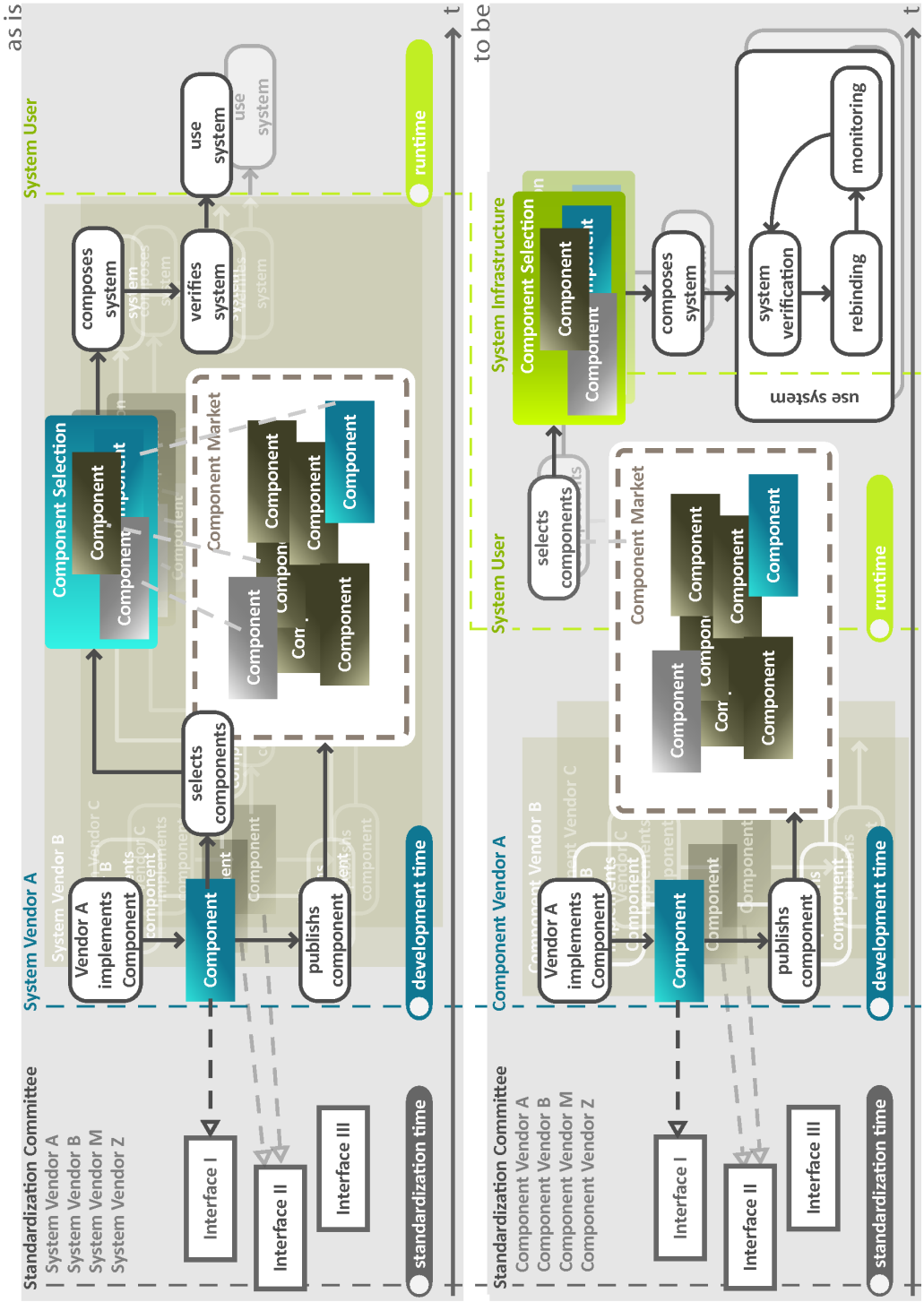


Figure 7.1: An Engineering Methodology for Dependable Dynamic Adaptive Systems Compared to One for Component Based Systems.

Finally there are implementation details, which may be improved in future releases of the reference implementation DAiSI. First of all, the simplifications currently made for the realization of DAiSI compared to our formal model can be removed. These simplifications includes the syntactical Compatibility, which is reduced to identical types of the implemented respectively referred to Service Interface instead of comparing the methods and attributes of the two Service Interfaces.

Syntactical Compatibility in DAiSI is currently based on identical types.

In Addition, a mechanism is needed, which ensures, that the calculation of Behavior Equivalence Classes itself does not change the Behavior Equivalence Classes. Classifying methods of a Service Interface into state-preserving and state-changing methods can help us to ensure this. By annotating [Chi05] these methods with these categories we can change the implementation of DAiSI to ensure, that only state-preserving methods are called during calculation of Behavior Equivalence Classes.

Only state-preserving methods may be called to calculate Behavior Equivalence Classes.

Next to this, DAiSI does not take care about cyclic dependencies of Components, where a Service Interface provided by ComponentA is required by ComponentB and vice versa. In such a situation DAiSI would *not* establish a System Configuration binding ComponentA to ComponentB and vice versa. One could enhance DAiSI to be able to establish such a System Configuration.

Cyclic dependencies are not considered.

However, this bears the risk of livelocks – especially, as the specific Service Provider has not necessarily been known at development time. Thus, one could alternatively include a cycle detection into DAiSI and inform the end user, that a Component has not been integrated into the system due to a cyclic dependency, which would be a huge threat for the system's Dependability.

Finally, DAiSI currently evaluates semantical Compatibility by executing Compliance Test Cases. This could be complemented by further verification and validation techniques like formal proofs. However, we need to focus on specific properties like deadlock-recognition, which can be evaluated at runtime using these techniques.

Test case execution can be complemented by techniques like formal proofs.

In this case, we need to provide a guideline, which properties may be evaluated using which verification and validation technique. However, combining several techniques will result in a further increase of the system's Dependability.

7.3 Additional Material

You can find additional material on the thesis' homepage http://www.dirkniebuhr.de/. It includes the following.

- A PDF version of this thesis.
- Our reference implementation DAiSI.
- The implementation of our application example.

This page will also contain known errata of this thesis in the future.

Appendices

Glossary

Activatable Dependable Component Configuration

If each Dependable Service required by a specific Dependable Component Configuration is available within a Dependable Dynamic Adaptive System, we call this Dependable Component Configuration *activatable*, as it could be activated by a Dependable System Infrastructure. A Component activated in a specific Dependable Component Configuration provides all Dependable Services of this configuration by using its Dependable Service References.

Behavior Equivalence Class

A *Behavior Equivalence Class* defines a state space of equivalent behavior of a Dependable Service. If *Behavior Equivalence Classes* are defined by a Dependable Service itself, they express equivalent provided behavior. These *Behavior Equivalence Classes* may have been derived from the control flow of the implementation respectively from its specification.

If *Behavior Equivalence Classes* in contrary are defined by a Dependable Service Reference, they define state spaces, where equivalent behavior is *assumed* by the Requesting Service Partner. The Requesting Service Partner declaring the Dependable Service Reference does not necessarily know the Unknown Service Partner providing the Dependable Service. Thus, these *Behavior Equivalence Classes* do not have to correspond to the state spaces defined as *Behavior Equivalence Classes* by a specific Dependable Service – they may differ, instead.

C-Unit

cf. Casualty Unit.

Casualty Unit

The *Casualty Unit* replaces the casualty card (cf. Figure 3.2) in our application example from the emergency assistance domain. It stores the Triage Class as well as the treatment history of a casualty. If it is connected to a so called

Peripheral Unit, it is able to calculate the Triage Class automatically from vital data information of a casualty.

Combined Behavior Equivalence Class

A *Combined Behavior Equivalence Class* describes an Behavior Equivalence Class for a Dependable Service Binding at runtime. It contains the User's Behavior Equivalence Class (Current Behavior Equivalence Class of the Service User) as well as the Provider's Behavior Equivalence Class (Current Behavior Equivalence Class of the Service Provider). *Combined Behavior Equivalence Classes* are not specified in advance but are combined from the Current Behavior Equivalence Classes of the associated Service Partners at runtime.

A change in a *Combined Behavior Equivalence Class* of a Dependable Service Binding means, that now a different behavior is assumed by the Requesting Service Partner or a different behavior is provided by the Unknown Service Partner. Thus, it has to be checked, whether this Dependable Service Binding is still valid by (re-)executing all Compliance Test Cases associated with the User's Behavior Equivalence Class of the *Combined Behavior Equivalence Class*.

Compatibility

The term *Compatibility* is used within this thesis to describe, that a Service Provider P and a Service User U can be bound together. Compatibility splits up into syntactical Compatibility and semantical Compatibility.

A Service S provided by Service Provider P is syntactically compatible with a Service Reference declared by a Service User U, if each method or attribute from R is provided by S.

Service S and Service Reference R are semantical compatible, if the behavior provided by S implies the behavior expected by R.

Since this implication cannot be proven in general, our approach is, describing the expected behavior by Compliance Test Cases, which are executed at runtime. As the result of the test execution may vary depending on the state of P and U, the semantical Compatibility may change during runtime of a Dependable Dynamic Adaptive System.

Compatibility Component

A *Compatibility Component* monitors the semantical Compatibility of Component Bindings of a System Configuration. It can detect incompatibilities between Components by executing Compliance Test Cases defined by the Requesting Service Partner of the Service Bindings.

A Compatibility Component can, therefore, be used by a Dependable Configuration Component to establish only Dependable System Configurations (System Configurations containing only Dependable Component Bindings).

Compliance Test Case

A *Compliance Test Case* is a test case defined by a Dependable Service Reference. It can be executed to evaluate, whether a Dependable Service behaves as expected by a Service User. Therefore, the *Compliance Test Case* will access the specific Dependable Service and reason whether its behavior is compatible with the Service User's expectations.

A Compliance Test Case is associated with a set of Behavior Equivalence Classes defined by the Dependable Service Reference. This means, that it needs to be executed, whenever the Combined Behavior Equivalence Class of a Dependable Service Binding associated with the Dependable Service Reference changes and the User's Behavior Equivalence Class of the Combined Behavior Equivalence Class is among the set of associated Behavior Equivalence Classes.

A passing *Compliance Test Case* states, that the Dependable Service Binding between this specific Dependable Service Reference and Dependable Service is semantically compatible in this specific Combined Behavior Equivalence Class. A failing *Compliance Test Case* states, that the Dependable Service Binding must not be established respectively needs to be removed since it is semantically incompatible in the specific Combined Behavior Equivalence Class.

Component

Several attempts were made to define a *Component* in the past [BDH+98]. We will try to focus on the commonalities of these definitions in the following. A Component is the fundamental building block [Gro04a] of Component-based systems. Thus, Components represent the *parts* of a System Configuration in Component-based systems. According to [Szy02] a Component is a *unit of independent deployment* and *subject to third-party composition.*

Component Binding

A *Component Binding* describes the connection between two specific Dynamic Adaptive Components R and U. A Component Binding exists between these two Components, if U provides at least one Service in its Current Configuration, which is bound to a Service Reference of the Current Configuration of R by a Service Binding. A *Component Binding* contains all Service Bindings which exist between these two Components.

Component Configuration

A Dynamic Adaptive Component describes the mapping between provided Services and required Services (declared as Service References) as *Component Configurations*. The meaning of a *Component Configuration* is, that all offered Services provided by this Component Configuration are offered to other Dynamic Adaptive Components, iff each Service Reference is bound to a Service provided by a Component Configuration of another Dynamic Adaptive Component from the System Configuration.

Configuration Component

The *Configuration Component* is a specific System Infrastructure Component, which is responsible for finding and establishing a System Configuration. To achieve this, it introspects the dependencies declared by Dynamic Adaptive Components in terms of Service References respectively Services.

Current Behavior Equivalence Class

The *Current Behavior Equivalence Class* describes the Behavior Equivalence Class of a Dependable Service at runtime from the Service User's view or the Service Provider's view.

The Service Provider's view of the *Current Behavior Equivalence Class* is provided by the Dependable Service based on its internal state. The Service User's view of the *Current Behavior Equivalence Class* is provided by the Dependable Service Reference based on its internal state as well as based on the Observable State of the Dependable Service.

Current Configuration

For each Dynamic Adaptive Component within a Dynamic Adaptive System's System Configuration, exactly one of its Component Configurations is the so called *Current Configuration*. This means, that for each Service Reference R declared by the *Current Configuration* a Service Binding exists, which binds R to a Service provided by a *Current Configuration* of another Dynamic Adaptive Component from the System Configuration.

Dependability

[ALR04] defines *Dependability* based on service failures. They define a service failure as an "event that occurs, when the delivered service deviates from correct service, either because the system does not comply with the specification, or because the specification did not adequately describe its function". Consequently they define Dependability as the "ability to avoid

service failures that are more frequent or more severe than acceptable". This is the definition of Dependability to which we refer within this thesis.

Dependability Checkpoint

We call a point in time during system runtime a *Dependability Checkpoint* if there are changes to the structure or behavior of the Dependable Dynamic Adaptive Systems which threaten its Dependability.

Dependable Component Binding

A *Dependable Component Binding* is a specific Component Binding between two Dependable Dynamic Adaptive Components, where all contained Service Bindings are semantically compatible Dependable Service Bindings meaning that all Dependable Services provided by Unknown Service Partners bound to the Dependable Service References of this Requesting Service Partner behave as expected by this Requesting Service Partner.

Dependable Component Configuration

A *Dependable Component Configuration* extends a Component Configuration in a way, that it provides Dependable Services and declares Dependable Service References instead of Services respectively Service References. This enables us to reason about the semantical Compatibility of a specific Component Binding, since each Dependable Service Reference may contain Compliance Test Cases which need to pass when they are bound to compatible Unknown Service Partners.

Dependable Configuration Component

The *Dependable Configuration Component* extends the Configuration Component in a way, that it only establishes Dependable System Configurations. To achieve this, it includes the runtime-testing mechanisms as introduced in this thesis to establish only Dependable Component Bindings between semantically compatible Dependable Dynamic Adaptive Components.

Dependable Dynamic Adaptive Component

Dependable Dynamic Adaptive Components are specific Dynamic Adaptive Components, which contain only Dependable Component Configurations and, therefore, may be bound in a dependable way (cf. Dependable Component Binding) by a Dependable Configuration Component.

Dependable Dynamic Adaptive System

A *Dependable Dynamic Adaptive System* shares all the characteristics of a Dynamic Adaptive System. In addition it needs to detect and avoid possi-

ble resulting semantically incompatible System Configurations during runtime and, therefore, establish only Dependable System Configurations.

This is done by using a Dependable System Infrastructure, which monitors that no semantically incompatible System Configurations are established at runtime.

Dependable Integration

By *Dependable Integration* we mean an integration of Dynamic Adaptive Components into a Dynamic Adaptive System while taking actions to ensure, that only semantically compatible Service Partners are bound.

In our approach this is achieved by runtime testing, enabling each Dynamic Adaptive Component acting as Requesting Service Partner to specify the desired behavior of Unknown Service Partners by test cases, which need to pass prior to integration as well as after integration during system runtime.

Dependable Service

A *Dependable Service* is a specific Service characterized by the feature that it has a set of Behavior Equivalence Classes. If a *Dependable Service* is offered by the Current Configuration of a Dependable Dynamic Adaptive Component at runtime, it is associated with exactly one of these Behavior Equivalence Classes. This so called Current Behavior Equivalence Class is determined by the internal state of the Dependable Service.

Whenever the Observable State of a *Dependable Service* changes, it notifies the Dependable System Infrastructure, which will check, whether the Combined Behavior Equivalence Classes of associated Dependable Service Bindings have changed. If this is the case, it will perform another check – using the runtime testing mechanisms from this thesis – whether the semantical Compatibility of Dependable Service Bindings has changed and, therefore, the Dependable Component Bindings from the Dependable System Configuration need to be updated due to a detected Semantical Incompatibility respectively Semantical Compatibility.

Dependable Service Binding

A *Dependable Service Binding* is a Service Binding between Dependable Service U and Dependable Service Reference R, where U provides an implementation of the Service Interface I. A *Dependable Service Binding* is associated with a semantical Compatibility, declaring, whether U behaves as expected by R. Using our approach that means, that a *Dependable Service Binding* between U and R is semantical compatible, if the Compliance Test Cases specified by R pass when executed on U.

Dependable Service Bindings may only be established by a Dependable Configuration Component if they are semantically compatible. If they are semantically incompatible, they still have to be considered by the Dependable System Infrastructure, since they may become semantically compatible if the state of one of the involved Components changes.

Dependable Service Reference

A *Dependable Service Reference* extends a Service Reference in a way, that it defines a set of Behavior Equivalence Classes regarding the Dependable Service provided by a Unknown Service Partner of a specific Dependable Service Binding.

These Behavior Equivalence Classes define state spaces, where the vendor of the Dependable Dynamic Adaptive Component containing this *Dependable Service Reference* assumes, that a Dependable Service provided by a third-party Component behaves equivalently. Exactly one of these Behavior Equivalence Classes – the so called Current Behavior Equivalence Class – is associated with each Dependable Service Binding at runtime. This means that a Dependable Service Reference referring to a Service Interface I needs to be able to return the Current Behavior Equivalence Class for any given Dependable Service, which implements I.

Next to the definitions of Behavior Equivalence Classes, a *Dependable Service Reference* defines Compliance Test Cases, which are associated with the Behavior Equivalence Classes and may be executed by the Dependable System Infrastructure if the Current Behavior Equivalence Class of the Dependable Service Reference is among the associated Behavior Equivalence Classes. The Dependable System Infrastructure executes these Compliance Test Cases whenever a Combined Behavior Equivalence Class of a Dependable Service Binding changes in order to decide, whether the semantical Compatibility of this Dependable Service Binding has changed.

Dependable System Configuration

A *Dependable System Configuration* is a System Configuration, which only contains Dependable Component Bindings between Dependable Dynamic Adaptive Components.

Dependable System Infrastructure

A *Dependable System Infrastructure* extends the System Infrastructure in a way, that it uses a Dependable Configuration Component which establishes only Dependable System Configurations.

This is achieved by checking the semantical Compatibility of Component Bindings within a System Configuration by using a so called Compatibility Component. Therefore, semantical incompatible System Configurations can be detected in advance.

Domain Architecture

By *Domain Architecture* we mean the specific Service Interfaces and datatypes, which are used by Dynamic Adaptive Components in Dynamic Adaptive Systems of a specific domain. These Service Interfaces and datatypes are expected to be *standardized in advance*, enabling a vendor to develop Dynamic Adaptive Components for a specific domain.

Next to syntactical information regarding Service Interfaces and datatypes, the *Domain Architecture* is expected to contain semantical information as well. This may include Service Interface protocols or a formal specification of the behavior of correct implementations of a Service Interface's methods. However, we do assume in this thesis, that Dynamic Adaptive Components may be incorrect regarding this semantical specification from the *Domain Architecture*.

Dynamic Adaptive Component

A *Dynamic Adaptive Component* is a Component, characterized by its continuous adaptation to a System Context or a User Context. You can think of a speech recognition Component, which adapts its speech recognition algorithm regarding the current physical stress of the speaking user (*adaptation to the User Context*) or a printer Component, which can provide a copy-function after adaptation, whenever a scanner Component is present in the system (*adaptation to the System Context*).

In order to support this adaptation, *Dynamic Adaptive Components* may specify several Component Configurations describing, which Services they offer depending on Services, which are offered by other *Dynamic Adaptive Components* in the Dynamic Adaptive System. This enables a System Infrastructure to update their Current Configurations, if new *Dynamic Adaptive Components* enter the Dynamic Adaptive System, which provide Services required by them.

Dynamic Adaptive Components act as Service Partners in Component Bindings meaning that they offer Services to *Dynamic Adaptive Components* or they use Services, which are offered by other *Dynamic Adaptive Components*. Component Bindings describe, how *Dynamic Adaptive Components* are connected among each other within a Dynamic Adaptive System.

Dynamic Adaptive Components are expected to publish their adaptation triggers to System Infrastructure Components, enabling automatic adaptation by a System Infrastructure.

Dynamic Adaptive System

A *Dynamic Adaptive System* is a system built from Dynamic Adaptive Components. How the system is composed from Dynamic Adaptive Components – namely which Dynamic Adaptive Components are contained in the system and how they are bound – is described in its System Configuration.

A *Dynamic Adaptive System* is characterized by the openness towards entering Dynamic Adaptive Components. This results in a continuous adaptation of the System Configuration during runtime due to changes in the System Context (e.g. newly entering respectively leaving Dynamic Adaptive Components) or due to changes in the User Context (e.g. a new user or changing usage preferences).

Thus, the System Configuration in a *Dynamic Adaptive System* needs to be established and updated at runtime automatically. This continuous adaptation of the System Configuration needs to be supported by a System Infrastructure.

IC-Unit

cf. Incident Command Unit.

Incident Command Unit

The *Incident Command Unit* is a Component of our application example from the emergency assistance domain. It may be a large display mounted to the inside of an ambulance vehicle. It enables medics, which coordinate a rescue operation, to assign medics to casualties efficiently.

To do this, the Incident Command Unit provides an graphical overview map, where all medics and casualties are displayed. It can display current activities of the medics as well as vital data and Triage Classes of the casualties.

M-Unit

cf. Medic Unit.

Medic Unit

The *Medic Unit* is the interaction device of a medic in our application example from the emergency assistance domain. A medic can assign Triage Classes to casualties, view their vital data or their treatment history. In addition, commands sent by the Incident Command Unit are displayed on the Medic Unit.

Observable State

If we talk about *Observable State* of a Dependable Service in this thesis, we mean its public visible state, observable by the value of its public attributes and return values of its state-preserving methods.

P-Unit

cf. Peripheral Unit.

Peripheral Unit

The *Peripheral Unit* is a Component of our application example from the emergency assistance domain. It features vital data sensors like a pulse rate sensor or a blood pressure sensor. If a medic attaches these sensors to a casualty, the *Peripheral Unit* enters the Dependable Dynamic Adaptive System. As a consequence the vital data of this casualty can be monitored by any medic.

Requesting Service Partner

A Dynamic Adaptive Component acts as a *Requesting Service Partner* when it declares Service References which may be bound to Services provided by other Dynamic Adaptive Components in their Current Configurations.

A Dependable Dynamic Adaptive Component acting as a *Requesting Service Partner* in addition needs to be able to decide, whether the specific Dependable Service to which its Dependable Service Reference may be bound by a Dependable Service Binding fulfills its expectations regarding the behavior.

Semantical Compatibility

We call a Dependable Service S implementing a Service Interface I and a Dependable Service Reference R referring to I *semantical compatible*, if the provided behavior of S matches the expectations of R.

Semantical Incompatibility

We call a Dependable Service S implementing a Service Interface I and a Dependable Service Reference R referring to I *semantical incompatible*, if the provided behavior of S does not match the expectations of R.

Service

A *Service* in this thesis is the implementation of a Service Interface provided by a specific Dynamic Adaptive Component in at least one of its Component Configurations. A Dynamic Adaptive Component provides *Services* as an Unknown Service Partner towards Requesting Service Partners.

In contrast to common *Service* definitions from service-oriented architecture, we do *not* assume, that a *Service* needs to be stateless.

Service Binding

A *Service Binding* describes the connection between two specific Dynamic Adaptive Components which declare a Service Reference R (cf. Requesting Service Partner) respectively provide a Service U (cf. Unknown Service Partner) that refers to respectively implements the same Service Interface I. The *Service Binding* describes the connection between R and U.

Service Interface

Each Domain Architecture contains a set of *Service Interfaces*. These *Service Interfaces* describe, how Dynamic Adaptive Components within a Dynamic Adaptive System may interact by defining methods and attributes, which may be offered by Dynamic Adaptive Components of the specific domain.

Unknown Service Partners refer to the *Service Interfaces* to describe the type of their provided Services whereas Requesting Service Partners contain Service References referring to *Service Interfaces* to indicate the type of Service, they request.

In practice *Service Interfaces* are typically described in Interface Description Languages like IDL [Gro04b] or WSDL [CCMW01]. These definitions only describe syntactical aspects of a Service Interface.

Service Partner

A *Service Partner* in the sense of this thesis describes the role of a Dynamic Adaptive Component in a Component Binding. A Dynamic Adaptive Component acts as a Requesting Service Partner when it is using Services provided by other Dynamic Adaptive Components whereas it acts as an Unknown Service Partner for other Dynamic Adaptive Components when it provides Services which are used by them.

As a Dynamic Adaptive Component may provide *and* use Services at the same time, it consequently may act as Requesting Service Partner and as Unknown Service Partner at the same time.

Service Provider

cf. Unknown Service Partner

Service Reference

A Dynamic Adaptive Component declares the Services it requires in a specific Component Configuration by *Service References*. A *Service Reference* R refers

to a Service Interface I, which needs to be implemented by a Service S in order to establish a Service Binding between S and R.

From a programmer's view a *Service Reference* simply is a field, which has the type of the Service Interface.

Service User

cf. Requesting Service Partner

System Configuration

By *System Configuration* we mean a set of Dynamic Adaptive Component instances at runtime and the Component Bindings between them. An adaptation of the *System Configuration*, therefore, may result from entering or leaving Dynamic Adaptive Components (change in the set of Dynamic Adaptive Component instances) as well as from changes of Component Bindings between Dynamic Adaptive Component instances of the *System Configuration*.

System Context

The *Context* of a Dynamic Adaptive System is described by all Dynamic Adaptive Components present in the system and their internal state. The *System Context* changes at runtime when new Dynamic Adaptive Components enter, existing Dynamic Adaptive Components leave the system or existing Dynamic Adaptive Components adapt themselves.

System Infrastructure

A *System Infrastructure* in this thesis contains infrastructure Components, which support us to build a Dynamic Adaptive System. This includes infrastructure Components responsible for instantiation or execution of Dynamic Adaptive Components.

The main focus of this thesis lies on a specific infrastructure Component responsible for establishing or updating the Dependable System Configuration within a Dependable Dynamic Adaptive System: the Dependable Configuration Component.

Triage Class

Triage Classes are used during rescue operations with a large number of casualties to classify casualties according to their treatment priority.

In Germany and further countries, casualties are classified in five Triage Classes: *T I* to *T IV* as well as *EX*.

- Triage Class *T I* characterizes casualties, which are severely wounded and need to be treated immediately.
- A casualty classified as *T II* needs to be evacuated to a hospital for further treatment.
- Casualties classified as *T III* are only slightly injured and are only treated rudimentary on-site.
- Triage Class *T IV* characterizes casualties, which have no chance to survive and are only treated in terms of terminal care.
- Moreover there is one additional Triage Class *EX* denoting casualties which have died.

Unknown Service Partner

A Dynamic Adaptive Component acts as an *Unknown Service Partner* for other Dynamic Adaptive Components (Requesting Service Partners) when it is bound to them since its Current Configuration offers Services, which they requested.

It is unknown in terms of the implementation of the associated Service Interfaces, as Requesting Service Partners need to ensure, that the behavior of the implementation provided by the *Unknown Service Partner* is semantically compatible with the behavior they expected when they declared the Service Reference.

User Context

The *User Context* contains information about the specific user(s) of a Dynamic Adaptive System. This context changes at runtime when a new user starts interacting with a Dynamic Adaptive System or when a user changes his interaction style, for example, due to a new task.

Index

A

Formal Specification of the Application Example

Within this appendix, we depict the relations from our formal model for the application example. We left out the behavioral parts for the provided Dependable Services of the C-Units and M-Units and for the Dependable Service References of the M-Units intentionally to get a more compact appendix.

A.1 Type Specification

A.1.1 Service Interface mUnitServiceIf

$$\text{Attributes}(\text{mUnitServiceIf}) = \emptyset \tag{A.1}$$

$$\text{Methods}(\text{mUnitServiceIf}) = \{\text{getLongitude}_M, \text{getLatitude}_M, \text{getActivity}_M, \text{getAssignedCasualty}_C\} \tag{A.2}$$

$$\begin{aligned} \text{MethodName}(\text{getLongitude}_M) &= \text{"getLongitude"} \\ \text{ReturnType}(\text{getLongitude}_M) &= \text{int} \\ \text{Parameters}(\text{getLongitude}_M) &= \emptyset \end{aligned} \tag{A.3}$$

$$\begin{aligned} \text{MethodName}(\text{getLatitude}_M) &= \text{"getLatitude"} \\ \text{ReturnType}(\text{getLatitude}_M) &= \text{int} \\ \text{Parameters}(\text{getLatitude}_M) &= \emptyset \end{aligned} \tag{A.4}$$

$$\begin{aligned} \text{MethodName}(\text{getActivity}_M) &= \text{"getActivity"} \\ \text{ReturnType}(\text{getActivity}_M) &= \text{Activity} \\ \text{Parameters}(\text{getActivity}_M) &= \emptyset \end{aligned} \tag{A.5}$$

$$\begin{aligned} \text{MethodName}(\text{getAssignedCasualty}_M) &= \text{"getAssignedCasualty"} \\ \text{ReturnType}(\text{getAssignedCasualty}_M) &= \text{CUnitServiceIf} \\ \text{Parameters}(\text{getAssignedCasualty}_M) &= \emptyset \end{aligned} \tag{A.6}$$

A.1.2 Service Interface cUnitServiceIf

$$\text{Attributes}(\text{cUnitServiceIf}) = \emptyset \tag{A.7}$$

$$\begin{aligned} \text{Methods}(\text{cUnitServiceIf}) = \{&\text{getLongitude}_C, \text{getLatitude}_C, \text{getTriageClass}_C, \\ &\text{setTriageClass}_C, \text{getAssignedPeripheralUnit}_C\} \end{aligned} \tag{A.8}$$

$$\begin{aligned} \text{MethodName}(\text{getLongitude}_C) &= \text{"getLongitude"} \\ \text{ReturnType}(\text{getLongitude}_C) &= \text{int} \\ \text{Parameters}(\text{getLongitude}_C) &= \emptyset \end{aligned} \tag{A.9}$$

$$\begin{aligned} \text{MethodName}(\text{getLatitude}_C) &= \text{"getLatitude"} \\ \text{ReturnType}(\text{getLatitude}_C) &= \text{int} \\ \text{Parameters}(\text{getLatitude}_C) &= \emptyset \end{aligned} \tag{A.10}$$

$$\begin{aligned} \text{MethodName}(\text{setTriageClass}_C) &= \text{"setTriageClass"} \\ \text{ReturnType}(\text{setTriageClass}_C) &= \text{void} \\ \text{Parameters}(\text{setTriageClass}_C) &= \{\text{triageClass}_C\} \end{aligned} \tag{A.11}$$

$$\begin{aligned} \text{MethodName}(\text{getTriageClass}_C) &= \text{"getTriageClass"} \\ \text{ReturnType}(\text{getTriageClass}_C) &= \text{TriageClass} \\ \text{Parameters}(\text{getTriageClass}_C) &= \emptyset \end{aligned} \tag{A.12}$$

$$\begin{aligned} \text{MethodName}(\text{getAssignedPeripheralUnit}_C) &= \text{"getAssignedPeripheralUnit"} \\ \text{ReturnType}(\text{getAssignedPeripheralUnit}_C) &= \text{PUnitServiceIf} \\ \text{Parameters}(\text{getAssignedPeripheralUnit}_C) &= \emptyset \end{aligned} \tag{A.13}$$

$$\begin{aligned} \text{AttributeName}(\text{triageClass}_C) &= \text{"triageClass"} \\ \text{AttributeType}(\text{triageClass}_C) &= \text{TriageClass} \end{aligned} \tag{A.14}$$

A.1.3 Service Interface pUnitServiceIf

$$\text{Attributes}(\text{pUnitServiceIf}) = \emptyset \tag{A.15}$$

$$\begin{aligned} \text{Methods}(\text{pUnitServiceIf}) = \{&\text{getSystolicBloodPressure}_P, \\ &\text{getDiastolicBloodPressure}_P, \text{getPulseRate}_P\} \end{aligned} \tag{A.16}$$

$$\begin{aligned} \text{MethodName}(\text{getSystolicBloodPressure}_P) &= \text{"getSystolicBloodPressure"} \\ \text{ReturnType}(\text{getSystolicBloodPressure}_P) &= \text{int} \\ \text{Parameters}(\text{getSystolicBloodPressure}_P) &= \emptyset \end{aligned} \tag{A.17}$$

$$\begin{aligned} \text{MethodName}(\text{getDiastolicBloodPressure}_P) &= \text{"getDiastolicBloodPressure"} \\ \text{ReturnType}(\text{getDiastolicBloodPressure}_P) &= \text{int} \\ \text{Parameters}(\text{getDiastolicBloodPressure}_P) &= \emptyset \end{aligned} \tag{A.18}$$

$$\begin{aligned} \text{MethodName}(\text{getPulseRate}_P) &= \text{"getPulseRate"} \\ \text{ReturnType}(\text{getPulseRate}_P) &= \text{int} \\ \text{Parameters}(\text{getPulseRate}_P) &= \emptyset \end{aligned} \tag{A.19}$$

A.2 Instances at Dependability Checkpoint t_0

At t_0, we face the situation that no Dependable Dynamic Adaptive Component is present in our Dependable Dynamic Adaptive System s_ae yet.

$$\text{ApplicationComponents}_{s_{ae}}^{t_0} = \emptyset \tag{A.20}$$

A.3 Instances at Dependability Checkpoint $t_0 + 1$

At $t_0 + 1$, we face the situation that a German M-Unit is present in our Dependable Dynamic Adaptive System s_ae.

$$\text{ApplicationComponents}_{s_{ae}}^{t_0+1} = \{\text{mUnit}_{\text{German}}\} \tag{A.21}$$

A.3.1 German M-Unit

$$\text{Contains}_{s_{ae}}^{t_0+1}(\text{mUnit}_{\text{German}}) = \{\text{mConfiguration1}_{\text{German}}, \text{mConfiguration2}_{\text{German}}\} \tag{A.22}$$

$$\text{mConfiguration2}_{\text{German}} \geq_{s_{ae}}^{t_0+1} \text{mConfiguration1}_{\text{German}} \tag{A.23}$$

$$\begin{aligned} &\text{Provides}_{s_{ae}}^{t_0+1}(\text{mConfiguration1}_{\text{German}}) = \{\text{mUnitService}_{\text{German}}\} \\ &\text{Declares}_{s_{ae}}^{t_0+1}(\text{mConfiguration1}_{\text{German}}) = \emptyset \\ &\text{Provides}_{s_{ae}}^{t_0+1}(\text{mConfiguration2}_{\text{German}}) = \{\text{mUnitService}_{\text{German}}\} \\ &\text{Declares}_{s_{ae}}^{t_0+1}(\text{mConfiguration2}_{\text{German}}) = \{\text{cUnitReference}_{\text{German}}\} \end{aligned} \tag{A.24}$$

$$\text{Current}_{s_{ae}}^{t_0+1}(\text{mUnit}_{\text{German}}) = \text{mConfiguration1}_{\text{German}} \tag{A.25}$$

$$\text{Uses}_{s_{ae}}^{t_0+1}(\text{cUnitReference}_{\text{German}}) = \emptyset \tag{A.26}$$

A.4 Instances at Dependability Checkpoint $t_0 + 2$

At $t_0 + 2$, we face the situation that next to the German M-Unit a German C-Unit is present in our Dependable Dynamic Adaptive System s_ae.

A.5 Instances at Dependability Checkpoint $t_0 + 3$

At t_0+3, we face the situation that a German P-Unit is deployed in our Dependable Dynamic Adaptive System s_ae.

$$\text{ApplicationComponents}_{s_{ae}}^{t_0+3} = \{\text{mUnit}_{\text{German}}, \text{cUnit}_{\text{German}}, \text{pUnit}_{\text{German}}\} \tag{A.27}$$

A.5.1 German M-Unit

$$\mathsf{Contains}_{s_{ae}}^{t_0+3}(\mathsf{mUnit}_{\mathsf{German}}) = \{\mathsf{mConfiguration1}_{\mathsf{German}}, \mathsf{mConfiguration2}_{\mathsf{German}}\} \tag{A.28}$$

$$\mathsf{mConfiguration2}_{\mathsf{German}} \geq_{s_{ae}}^{t_0+3} \mathsf{mConfiguration1}_{\mathsf{German}} \tag{A.29}$$

$$\begin{aligned} \mathsf{Provides}_{s_{ae}}^{t_0+3}(\mathsf{mConfiguration1}_{\mathsf{German}}) &= \{\mathsf{mUnitService}_{\mathsf{German}}\} \\ \mathsf{Declares}_{s_{ae}}^{t_0+3}(\mathsf{mConfiguration1}_{\mathsf{German}}) &= \emptyset \\ \mathsf{Provides}_{s_{ae}}^{t_0+3}(\mathsf{mConfiguration2}_{\mathsf{German}}) &= \{\mathsf{mUnitService}_{\mathsf{German}}\} \\ \mathsf{Declares}_{s_{ae}}^{t_0+3}(\mathsf{mConfiguration2}_{\mathsf{German}}) &= \{\mathsf{cUnitReference}_{\mathsf{German}}\} \end{aligned} \tag{A.30}$$

$$\mathsf{Current}_{s_{ae}}^{t_0+3}(\mathsf{mUnit}_{\mathsf{German}}) = \mathsf{mConfiguration2}_{\mathsf{German}} \tag{A.31}$$

$$\mathsf{Uses}_{s_{ae}}^{t_0+3}(\mathsf{cUnitReference}_{\mathsf{German}}) = \mathsf{cUnitService}_{\mathsf{German}} \tag{A.32}$$

A.5.2 German C-Unit

$$\mathsf{Contains}_{s_{ae}}^{t_0+3}(\mathsf{cUnit}_{\mathsf{German}}) = \{\mathsf{cConfiguration1}_{\mathsf{German}}, \mathsf{cConfiguration2}_{\mathsf{German}}\} \tag{A.33}$$

$$\mathsf{cConfiguration2}_{\mathsf{German}} \geq_{s_{ae}}^{t_0+3} \mathsf{cConfiguration1}_{\mathsf{German}} \tag{A.34}$$

$$\begin{aligned} \mathsf{Provides}_{s_{ae}}^{t_0+3}(\mathsf{cConfiguration1}_{\mathsf{German}}) &= \{\mathsf{cUnitService}_{\mathsf{German}}\} \\ \mathsf{Declares}_{s_{ae}}^{t_0+3}(\mathsf{cConfiguration1}_{\mathsf{German}}) &= \emptyset \\ \mathsf{Provides}_{s_{ae}}^{t_0+3}(\mathsf{cConfiguration2}_{\mathsf{German}}) &= \{\mathsf{cUnitService}_{\mathsf{German}}\} \\ \mathsf{Declares}_{s_{ae}}^{t_0+3}(\mathsf{cConfiguration2}_{\mathsf{German}}) &= \{\mathsf{pUnitReference}_{\mathsf{German}}\} \end{aligned} \tag{A.35}$$

$$\mathsf{Current}_{s_{ae}}^{t_0+3}(\mathsf{cUnit}_{\mathsf{German}}) = \mathsf{cConfiguration2}_{\mathsf{German}} \tag{A.36}$$

$$\mathsf{Uses}_{s_{ae}}^{t_0+3}(\mathsf{pUnitReference}_{\mathsf{German}}) = \mathsf{pUnitService}_{\mathsf{German}} \tag{A.37}$$

A.5.3 German P-Unit

$$\mathsf{Contains}_{s_{ae}}^{t_0+3}(\mathsf{pUnit}_{\mathsf{German}}) = \{\mathsf{pConfiguration1}_{\mathsf{German}}\} \tag{A.38}$$

$$\begin{aligned}\mathsf{Provides}_{s_{ae}}^{t_0+3}(\mathsf{pConfiguration1}_{\mathsf{German}}) &= \{\mathsf{pUnitService}_{\mathsf{German}}\}\\ \mathsf{Declares}_{s_{ae}}^{t_0+3}(\mathsf{pConfiguration1}_{\mathsf{German}}) &= \emptyset\end{aligned} \tag{A.39}$$

$$\mathsf{Current}_{s_{ae}}^{t_0+3}(\mathsf{pUnit}_{\mathsf{German}}) = \mathsf{pConfiguration1}_{\mathsf{German}} \tag{A.40}$$

A.5.4 Semantical Compatibility

$$\begin{aligned}\mathsf{ServiceBehavior}_{s_{ae}}^{t_0+3}(\mathsf{pUnitService}_{\mathsf{German}}) =\\ \mathsf{pUnitServiceUsualOperation}_{\mathsf{German}}\end{aligned} \tag{A.41}$$

$$\begin{aligned}\mathsf{ServiceReferenceBehavior}_{s_{ae}}^{t_0+3}(\mathsf{pUnitReference}_{\mathsf{German}},\\ \mathsf{pUnitService}_{\mathsf{German}}) = \mathsf{pUnitReferenceUsualOperation}_{\mathsf{German}}\end{aligned} \tag{A.42}$$

$$\mathsf{pUnitReference}_{\mathsf{German}} \simeq_{\mathsf{Semantical}_{s_{ae}}^{t_0+3}} \mathsf{pUnitService}_{\mathsf{German}} \tag{A.43}$$

A.6 Instances at Dependability Checkpoint $t_0 + 4$

At $t_0 + 4$, we face the situation that a Dutch M-Unit joins our Dependable Dynamic Adaptive System s_ae.

$$\begin{aligned}\mathsf{ApplicationComponents}_{s_{ae}}^{t_0+4} = \{&\mathsf{mUnit}_{\mathsf{German}}, \mathsf{cUnit}_{\mathsf{German}}, \mathsf{pUnit}_{\mathsf{German}},\\ &\mathsf{mUnit}_{\mathsf{Dutch}}\}\end{aligned} \tag{A.44}$$

A.6.1 German M-Unit

$$\begin{aligned}\mathsf{Contains}_{s_{ae}}^{t_0+4}(\mathsf{mUnit}_{\mathsf{German}}) = \{&\mathsf{mConfiguration1}_{\mathsf{German}},\\ &\mathsf{mConfiguration2}_{\mathsf{German}}\}\end{aligned} \tag{A.45}$$

$$\mathsf{mConfiguration2}_{\mathsf{German}} \geq_{s_{ae}}^{t_0+4} \mathsf{mConfiguration1}_{\mathsf{German}} \tag{A.46}$$

$$\begin{aligned}
&\mathsf{Provides}_{s_{ae}}^{t_0+4}(\mathsf{mConfiguration1}_{\mathsf{German}}) = \{\mathsf{mUnitService}_{\mathsf{German}}\} \\
&\mathsf{Declares}_{s_{ae}}^{t_0+4}(\mathsf{mConfiguration1}_{\mathsf{German}}) = \emptyset \\
&\mathsf{Provides}_{s_{ae}}^{t_0+4}(\mathsf{mConfiguration2}_{\mathsf{German}}) = \{\mathsf{mUnitService}_{\mathsf{German}}\} \\
&\mathsf{Declares}_{s_{ae}}^{t_0+4}(\mathsf{mConfiguration2}_{\mathsf{German}}) = \{\mathsf{cUnitReference}_{\mathsf{German}}\}
\end{aligned} \tag{A.47}$$

$$\mathsf{Current}_{s_{ae}}^{t_0+4}(\mathsf{mUnit}_{\mathsf{German}}) = \mathsf{mConfiguration2}_{\mathsf{German}} \tag{A.48}$$

$$\mathsf{Uses}_{s_{ae}}^{t_0+4}(\mathsf{cUnitReference}_{\mathsf{German}}) = \mathsf{cUnitService}_{\mathsf{German}} \tag{A.49}$$

A.6.2 German C-Unit

$$\begin{aligned}
\mathsf{Contains}_{s_{ae}}^{t_0+4}(\mathsf{cUnit}_{\mathsf{German}}) = \{&\mathsf{cConfiguration1}_{\mathsf{German}}, \\
&\mathsf{cConfiguration2}_{\mathsf{German}}\}
\end{aligned} \tag{A.50}$$

$$\mathsf{cConfiguration2}_{\mathsf{German}} \geq_{s_{ae}}^{t_0+4} \mathsf{cConfiguration1}_{\mathsf{German}} \tag{A.51}$$

$$\begin{aligned}
&\mathsf{Provides}_{s_{ae}}^{t_0+4}(\mathsf{cConfiguration1}_{\mathsf{German}}) = \{\mathsf{cUnitService}_{\mathsf{German}}\} \\
&\mathsf{Declares}_{s_{ae}}^{t_0+4}(\mathsf{cConfiguration1}_{\mathsf{German}}) = \emptyset \\
&\mathsf{Provides}_{s_{ae}}^{t_0+4}(\mathsf{cConfiguration2}_{\mathsf{German}}) = \{\mathsf{cUnitService}_{\mathsf{German}}\} \\
&\mathsf{Declares}_{s_{ae}}^{t_0+4}(\mathsf{cConfiguration2}_{\mathsf{German}}) = \{\mathsf{pUnitReference}_{\mathsf{German}}\}
\end{aligned} \tag{A.52}$$

$$\mathsf{Current}_{s_{ae}}^{t_0+4}(\mathsf{cUnit}_{\mathsf{German}}) = \mathsf{cConfiguration2}_{\mathsf{German}} \tag{A.53}$$

$$\mathsf{Uses}_{s_{ae}}^{t_0+4}(\mathsf{pUnitReference}_{\mathsf{German}}) = \mathsf{pUnitService}_{\mathsf{German}} \tag{A.54}$$

A.6.3 German P-Unit

$$\mathsf{Contains}_{s_{ae}}^{t_0+4}(\mathsf{pUnit}_{\mathsf{German}}) = \{\mathsf{pConfiguration1}_{\mathsf{German}}\} \tag{A.55}$$

$$\begin{aligned}
&\mathsf{Provides}_{s_{ae}}^{t_0+4}(\mathsf{pConfiguration1}_{\mathsf{German}}) = \{\mathsf{pUnitService}_{\mathsf{German}}\} \\
&\mathsf{Declares}_{s_{ae}}^{t_0+4}(\mathsf{pConfiguration1}_{\mathsf{German}}) = \emptyset
\end{aligned} \tag{A.56}$$

$$\mathsf{Current}_{s_{ae}}^{t_0+4}(\mathsf{pUnit}_{\mathsf{German}}) = \mathsf{pConfiguration1}_{\mathsf{German}} \tag{A.57}$$

A.6.4 Dutch M-Unit

$$\text{Contains}_{s_{ae}}^{t_0+4}(\text{mUnit}_{\text{Dutch}}) = \{\text{mConfiguration1}_{\text{Dutch}}, \text{mConfiguration2}_{\text{Dutch}}\} \tag{A.58}$$

$$\text{mConfiguration2}_{\text{Dutch}} \geq_{s_{ae}}^{t_0+4} \text{mConfiguration1}_{\text{Dutch}} \tag{A.59}$$

$$\begin{aligned} &\text{Provides}_{s_{ae}}^{t_0+4}(\text{mConfiguration1}_{\text{Dutch}}) = \{\text{mUnitService}_{\text{Dutch}}\} \\ &\text{Declares}_{s_{ae}}^{t_0+4}(\text{mConfiguration1}_{\text{Dutch}}) = \emptyset \\ &\text{Provides}_{s_{ae}}^{t_0+4}(\text{mConfiguration2}_{\text{Dutch}}) = \{\text{mUnitService}_{\text{Dutch}}\} \\ &\text{Declares}_{s_{ae}}^{t_0+4}(\text{mConfiguration2}_{\text{Dutch}}) = \{\text{cUnitReference}_{\text{Dutch}}\} \end{aligned} \tag{A.60}$$

$$\text{Current}_{s_{ae}}^{t_0+4}(\text{mUnit}_{\text{Dutch}}) = \text{mConfiguration1}_{\text{Dutch}} \tag{A.61}$$

$$\text{Uses}_{s_{ae}}^{t_0+4}(\text{cUnitReference}_{\text{Dutch}}) = \emptyset \tag{A.62}$$

A.6.5 Semantical Compatibility

$$\begin{aligned} &\text{ServiceBehavior}_{s_{ae}}^{t_0+4}(\text{pUnitService}_{\text{German}}) = \\ &\text{pUnitServiceUsualOperation}_{\text{German}} \end{aligned} \tag{A.63}$$

$$\begin{aligned} &\text{ServiceReferenceBehavior}_{s_{ae}}^{t_0+4}(\text{pUnitReference}_{\text{German}}, \\ &\text{pUnitService}_{\text{German}}) = \text{pUnitReferenceUsualOperation}_{\text{German}} \end{aligned} \tag{A.64}$$

$$\text{pUnitReference}_{\text{German}} \simeq_{\text{Semantical}_{s_{ae}}^{t_0+4}} \text{pUnitService}_{\text{German}} \tag{A.65}$$

A.7 Instances at Dependability Checkpoint $t_0 + 5$

At $t_0 + 5$, we face the situation that a Dutch C-Unit is introduced in our Dependable Dynamic Adaptive System s_ae.

$$\begin{aligned} \text{ApplicationComponents}_{s_{ae}}^{t_0+5} = \{&\text{mUnit}_{\text{German}}, \text{cUnit}_{\text{German}}, \text{pUnit}_{\text{German}}, \\ &\text{mUnit}_{\text{Dutch}}, \text{cUnit}_{\text{Dutch}}\} \end{aligned} \tag{A.66}$$

A.7.1 German M-Unit

$$\mathsf{Contains}_{s_{ae}}^{t_0+5}(\mathsf{mUnit}_{\mathsf{German}}) = \{\mathsf{mConfiguration1}_{\mathsf{German}}, \mathsf{mConfiguration2}_{\mathsf{German}}\} \tag{A.67}$$

$$\mathsf{mConfiguration2}_{\mathsf{German}} \geq_{s_{ae}}^{t_0+5} \mathsf{mConfiguration1}_{\mathsf{German}} \tag{A.68}$$

$$\begin{aligned} &\mathsf{Provides}_{s_{ae}}^{t_0+5}(\mathsf{mConfiguration1}_{\mathsf{German}}) = \{\mathsf{mUnitService}_{\mathsf{German}}\} \\ &\mathsf{Declares}_{s_{ae}}^{t_0+5}(\mathsf{mConfiguration1}_{\mathsf{German}}) = \emptyset \\ &\mathsf{Provides}_{s_{ae}}^{t_0+5}(\mathsf{mConfiguration2}_{\mathsf{German}}) = \{\mathsf{mUnitService}_{\mathsf{German}}\} \\ &\mathsf{Declares}_{s_{ae}}^{t_0+5}(\mathsf{mConfiguration2}_{\mathsf{German}}) = \{\mathsf{cUnitReference}_{\mathsf{German}}\} \end{aligned} \tag{A.69}$$

$$\mathsf{Current}_{s_{ae}}^{t_0+5}(\mathsf{mUnit}_{\mathsf{German}}) = \mathsf{mConfiguration2}_{\mathsf{German}} \tag{A.70}$$

$$\mathsf{Uses}_{s_{ae}}^{t_0+5}(\mathsf{cUnitReference}_{\mathsf{German}}) = \mathsf{cUnitService}_{\mathsf{German}} \tag{A.71}$$

A.7.2 German C-Unit

$$\mathsf{Contains}_{s_{ae}}^{t_0+5}(\mathsf{cUnit}_{\mathsf{German}}) = \{\mathsf{cConfiguration1}_{\mathsf{German}}, \mathsf{cConfiguration2}_{\mathsf{German}}\} \tag{A.72}$$

$$\mathsf{cConfiguration2}_{\mathsf{German}} \geq_{s_{ae}}^{t_0+5} \mathsf{cConfiguration1}_{\mathsf{German}} \tag{A.73}$$

$$\begin{aligned} &\mathsf{Provides}_{s_{ae}}^{t_0+5}(\mathsf{cConfiguration1}_{\mathsf{German}}) = \{\mathsf{cUnitService}_{\mathsf{German}}\} \\ &\mathsf{Declares}_{s_{ae}}^{t_0+5}(\mathsf{cConfiguration1}_{\mathsf{German}}) = \emptyset \\ &\mathsf{Provides}_{s_{ae}}^{t_0+5}(\mathsf{cConfiguration2}_{\mathsf{German}}) = \{\mathsf{cUnitService}_{\mathsf{German}}\} \\ &\mathsf{Declares}_{s_{ae}}^{t_0+5}(\mathsf{cConfiguration2}_{\mathsf{German}}) = \{\mathsf{pUnitReference}_{\mathsf{German}}\} \end{aligned} \tag{A.74}$$

$$\mathsf{Current}_{s_{ae}}^{t_0+5}(\mathsf{cUnit}_{\mathsf{German}}) = \mathsf{cConfiguration2}_{\mathsf{German}} \tag{A.75}$$

$$\mathsf{Uses}_{s_{ae}}^{t_0+5}(\mathsf{pUnitReference}_{\mathsf{German}}) = \mathsf{pUnitService}_{\mathsf{German}} \tag{A.76}$$

A.7.3 German P-Unit

$$\mathsf{Contains}_{s_{ae}}^{t_0+5}(\mathsf{pUnit}_{\mathsf{German}}) = \{\mathsf{pConfiguration1}_{\mathsf{German}}\} \tag{A.77}$$

$$\begin{aligned}\mathsf{Provides}_{s_{ae}}^{t_0+5}(\mathsf{pConfiguration1}_{\mathsf{German}}) &= \{\mathsf{pUnitService}_{\mathsf{German}}\}\\ \mathsf{Declares}_{s_{ae}}^{t_0+5}(\mathsf{pConfiguration1}_{\mathsf{German}}) &= \emptyset\end{aligned} \tag{A.78}$$

$$\mathsf{Current}_{s_{ae}}^{t_0+5}(\mathsf{pUnit}_{\mathsf{German}}) = \mathsf{pConfiguration1}_{\mathsf{German}} \tag{A.79}$$

A.7.4 Dutch M-Unit

$$\mathsf{Contains}_{s_{ae}}^{t_0+5}(\mathsf{mUnit}_{\mathsf{Dutch}}) = \{\mathsf{mConfiguration1}_{\mathsf{Dutch}}, \mathsf{mConfiguration2}_{\mathsf{Dutch}}\} \tag{A.80}$$

$$\mathsf{mConfiguration2}_{\mathsf{Dutch}} \geq_{s_{ae}}^{t_0+5} \mathsf{mConfiguration1}_{\mathsf{Dutch}} \tag{A.81}$$

$$\begin{aligned}\mathsf{Provides}_{s_{ae}}^{t_0+5}(\mathsf{mConfiguration1}_{\mathsf{Dutch}}) &= \{\mathsf{mUnitService}_{\mathsf{Dutch}}\}\\ \mathsf{Declares}_{s_{ae}}^{t_0+5}(\mathsf{mConfiguration1}_{\mathsf{Dutch}}) &= \emptyset\\ \mathsf{Provides}_{s_{ae}}^{t_0+5}(\mathsf{mConfiguration2}_{\mathsf{Dutch}}) &= \{\mathsf{mUnitService}_{\mathsf{Dutch}}\}\\ \mathsf{Declares}_{s_{ae}}^{t_0+5}(\mathsf{mConfiguration2}_{\mathsf{Dutch}}) &= \{\mathsf{cUnitReference}_{\mathsf{Dutch}}\}\end{aligned} \tag{A.82}$$

$$\mathsf{Current}_{s_{ae}}^{t_0+5}(\mathsf{mUnit}_{\mathsf{Dutch}}) = \mathsf{mConfiguration2}_{\mathsf{Dutch}} \tag{A.83}$$

$$\mathsf{Uses}_{s_{ae}}^{t_0+5}(\mathsf{cUnitReference}_{\mathsf{Dutch}}) = \mathsf{cUnitService}_{\mathsf{Dutch}} \tag{A.84}$$

A.7.5 Dutch C-Unit

$$\mathsf{Contains}_{s_{ae}}^{t_0+5}(\mathsf{cUnit}_{\mathsf{Dutch}}) = \{\mathsf{cConfiguration1}_{\mathsf{Dutch}}, \mathsf{cConfiguration2}_{\mathsf{Dutch}}\} \tag{A.85}$$

$$\mathsf{cConfiguration2}_{\mathsf{Dutch}} \geq_{s_{ae}}^{t_0+5} \mathsf{cConfiguration1}_{\mathsf{Dutch}} \tag{A.86}$$

$$\begin{aligned}\mathsf{Provides}_{s_{ae}}^{t_0+5}(\mathsf{cConfiguration1}_{\mathsf{Dutch}}) &= \{\mathsf{cUnitService}_{\mathsf{Dutch}}\}\\ \mathsf{Declares}_{s_{ae}}^{t_0+5}(\mathsf{cConfiguration1}_{\mathsf{Dutch}}) &= \emptyset\\ \mathsf{Provides}_{s_{ae}}^{t_0+5}(\mathsf{cConfiguration2}_{\mathsf{Dutch}}) &= \{\mathsf{cUnitService}_{\mathsf{Dutch}}\}\\ \mathsf{Declares}_{s_{ae}}^{t_0+5}(\mathsf{cConfiguration2}_{\mathsf{Dutch}}) &= \{\mathsf{pUnitReference}_{\mathsf{Dutch}}\}\end{aligned} \tag{A.87}$$

$$\mathsf{Current}_{s_{ae}}^{t_0+5}(\mathsf{cUnit}_{\mathsf{Dutch}}) = \mathsf{cConfiguration1}_{\mathsf{Dutch}} \tag{A.88}$$

$$\mathsf{Uses}_{s_{ae}}^{t_0+5}(\mathsf{pUnitReference}_{\mathsf{Dutch}}) = \emptyset \tag{A.89}$$

A.7.6 Semantical Compatibility

$$\text{ServiceBehavior}_{s_{ae}}^{t_0+5}(\text{pUnitService}_{\text{German}}) = \text{pUnitServiceUsualOperation}_{\text{German}} \tag{A.90}$$

$$\text{ServiceReferenceBehavior}_{s_{ae}}^{t_0+5}(\text{pUnitReference}_{\text{German}}, \text{pUnitService}_{\text{German}}) = \text{pUnitReferenceUsualOperation}_{\text{German}} \tag{A.91}$$

$$\text{ServiceReferenceBehavior}_{s_{ae}}^{t_0+5}(\text{pUnitReference}_{\text{Dutch}}, \text{pUnitService}_{\text{German}}) = \text{pUnitReferenceUsualOperation}_{\text{Dutch}} \tag{A.92}$$

$$\text{pUnitReference}_{\text{German}} \simeq_{\text{Semantical}}^{t_0+5} \text{pUnitService}_{\text{German}} \tag{A.93}$$

$$\text{pUnitReference}_{\text{Dutch}} \simeq_{\text{Semantical}}^{t_0+5} \text{pUnitService}_{\text{German}} \tag{A.94}$$

A.8 Instances at Dependability Checkpoint $t_0 + 6$

At $t_0 + 6$, we face the situation that a Dutch P-Unit is attached to the Dutch C-Unit in our Dependable Dynamic Adaptive System s_ae.

$$\text{ApplicationComponents}_{s_{ae}}^{t_0+6} = \{\text{mUnit}_{\text{German}}, \text{cUnit}_{\text{German}}, \text{pUnit}_{\text{German}}, \text{mUnit}_{\text{Dutch}}, \text{cUnit}_{\text{Dutch}}, \text{pUnit}_{\text{Dutch}}\} \tag{A.95}$$

A.8.1 German M-Unit

$$\text{Contains}_{s_{ae}}^{t_0+6}(\text{mUnit}_{\text{German}}) = \{\text{mConfiguration1}_{\text{German}}, \text{mConfiguration2}_{\text{German}}\} \tag{A.96}$$

$$\text{mConfiguration2}_{\text{German}} \geq_{s_{ae}}^{t_0+6} \text{mConfiguration1}_{\text{German}} \tag{A.97}$$

$$\begin{aligned} \text{Provides}_{s_{ae}}^{t_0+6}(\text{mConfiguration1}_{\text{German}}) &= \{\text{mUnitService}_{\text{German}}\} \\ \text{Declares}_{s_{ae}}^{t_0+6}(\text{mConfiguration1}_{\text{German}}) &= \emptyset \\ \text{Provides}_{s_{ae}}^{t_0+6}(\text{mConfiguration2}_{\text{German}}) &= \{\text{mUnitService}_{\text{German}}\} \\ \text{Declares}_{s_{ae}}^{t_0+6}(\text{mConfiguration2}_{\text{German}}) &= \{\text{cUnitReference}_{\text{German}}\} \end{aligned} \tag{A.98}$$

$$\text{Current}_{s_{ae}}^{t_0+6}(\text{mUnit}_{\text{German}}) = \text{mConfiguration2}_{\text{German}} \tag{A.99}$$

$$\text{Uses}_{s_{ae}}^{t_0+6}(\text{cUnitReference}_{\text{German}}) = \text{cUnitService}_{\text{German}} \tag{A.100}$$

A.8.2 German C-Unit

$$\mathsf{Contains}_{s_{ae}}^{t_0+6}(\mathsf{cUnit}_{\mathsf{German}}) = \{\mathsf{cConfiguration1}_{\mathsf{German}}, \mathsf{cConfiguration2}_{\mathsf{German}}\} \tag{A.101}$$

$$\mathsf{cConfiguration2}_{\mathsf{German}} \geq_{s_{ae}}^{t_0+6} \mathsf{cConfiguration1}_{\mathsf{German}} \tag{A.102}$$

$$\begin{aligned} \mathsf{Provides}_{s_{ae}}^{t_0+6}(\mathsf{cConfiguration1}_{\mathsf{German}}) &= \{\mathsf{cUnitService}_{\mathsf{German}}\} \\ \mathsf{Declares}_{s_{ae}}^{t_0+6}(\mathsf{cConfiguration1}_{\mathsf{German}}) &= \emptyset \\ \mathsf{Provides}_{s_{ae}}^{t_0+6}(\mathsf{cConfiguration2}_{\mathsf{German}}) &= \{\mathsf{cUnitService}_{\mathsf{German}}\} \\ \mathsf{Declares}_{s_{ae}}^{t_0+6}(\mathsf{cConfiguration2}_{\mathsf{German}}) &= \{\mathsf{pUnitReference}_{\mathsf{German}}\} \end{aligned} \tag{A.103}$$

$$\mathsf{Current}_{s_{ae}}^{t_0+6}(\mathsf{cUnit}_{\mathsf{German}}) = \mathsf{cConfiguration2}_{\mathsf{German}} \tag{A.104}$$

$$\mathsf{Uses}_{s_{ae}}^{t_0+6}(\mathsf{pUnitReference}_{\mathsf{German}}) = \mathsf{pUnitService}_{\mathsf{German}} \tag{A.105}$$

A.8.3 German P-Unit

$$\mathsf{Contains}_{s_{ae}}^{t_0+6}(\mathsf{pUnit}_{\mathsf{German}}) = \{\mathsf{pConfiguration1}_{\mathsf{German}}\} \tag{A.106}$$

$$\begin{aligned} \mathsf{Provides}_{s_{ae}}^{t_0+6}(\mathsf{pConfiguration1}_{\mathsf{German}}) &= \{\mathsf{pUnitService}_{\mathsf{German}}\} \\ \mathsf{Declares}_{s_{ae}}^{t_0+6}(\mathsf{pConfiguration1}_{\mathsf{German}}) &= \emptyset \end{aligned} \tag{A.107}$$

$$\mathsf{Current}_{s_{ae}}^{t_0+6}(\mathsf{pUnit}_{\mathsf{German}}) = \mathsf{pConfiguration1}_{\mathsf{German}} \tag{A.108}$$

A.8.4 Dutch M-Unit

$$\mathsf{Contains}_{s_{ae}}^{t_0+6}(\mathsf{mUnit}_{\mathsf{Dutch}}) = \{\mathsf{mConfiguration1}_{\mathsf{Dutch}}, \mathsf{mConfiguration2}_{\mathsf{Dutch}}\} \tag{A.109}$$

$$\mathsf{mConfiguration2}_{\mathsf{Dutch}} \geq_{s_{ae}}^{t_0+6} \mathsf{mConfiguration1}_{\mathsf{Dutch}} \tag{A.110}$$

$$\begin{aligned} \mathsf{Provides}_{s_{ae}}^{t_0+6}(\mathsf{mConfiguration1}_{\mathsf{Dutch}}) &= \{\mathsf{mUnitService}_{\mathsf{Dutch}}\} \\ \mathsf{Declares}_{s_{ae}}^{t_0+6}(\mathsf{mConfiguration1}_{\mathsf{Dutch}}) &= \emptyset \\ \mathsf{Provides}_{s_{ae}}^{t_0+6}(\mathsf{mConfiguration2}_{\mathsf{Dutch}}) &= \{\mathsf{mUnitService}_{\mathsf{Dutch}}\} \\ \mathsf{Declares}_{s_{ae}}^{t_0+6}(\mathsf{mConfiguration2}_{\mathsf{Dutch}}) &= \{\mathsf{cUnitReference}_{\mathsf{Dutch}}\} \end{aligned} \tag{A.111}$$

$$\mathsf{Current}_{s_{ae}}^{t_0+6}(\mathsf{mUnit}_{\mathsf{Dutch}}) = \mathsf{mConfiguration2}_{\mathsf{Dutch}} \tag{A.112}$$

$$\mathsf{Uses}_{s_{ae}}^{t_0+6}(\mathsf{cUnitReference}_{\mathsf{Dutch}}) = \mathsf{cUnitService}_{\mathsf{Dutch}} \tag{A.113}$$

A.8.5 Dutch C-Unit

$$\mathsf{Contains}_{s_{ae}}^{t_0+6}(\mathsf{cUnit}_{\mathsf{Dutch}}) = \{\mathsf{cConfiguration1}_{\mathsf{Dutch}}, \mathsf{cConfiguration2}_{\mathsf{Dutch}}\} \tag{A.114}$$

$$\mathsf{cConfiguration2}_{\mathsf{Dutch}} \geq_{s_{ae}}^{t_0+6} \mathsf{cConfiguration1}_{\mathsf{Dutch}} \tag{A.115}$$

$$\begin{aligned} &\mathsf{Provides}_{s_{ae}}^{t_0+6}(\mathsf{cConfiguration1}_{\mathsf{Dutch}}) = \{\mathsf{cUnitService}_{\mathsf{Dutch}}\} \\ &\mathsf{Declares}_{s_{ae}}^{t_0+6}(\mathsf{cConfiguration1}_{\mathsf{Dutch}}) = \emptyset \\ &\mathsf{Provides}_{s_{ae}}^{t_0+6}(\mathsf{cConfiguration2}_{\mathsf{Dutch}}) = \{\mathsf{cUnitService}_{\mathsf{Dutch}}\} \\ &\mathsf{Declares}_{s_{ae}}^{t_0+6}(\mathsf{cConfiguration2}_{\mathsf{Dutch}}) = \{\mathsf{pUnitReference}_{\mathsf{Dutch}}\} \end{aligned} \tag{A.116}$$

$$\mathsf{Current}_{s_{ae}}^{t_0+6}(\mathsf{cUnit}_{\mathsf{Dutch}}) = \mathsf{cConfiguration2}_{\mathsf{Dutch}} \tag{A.117}$$

$$\mathsf{Uses}_{s_{ae}}^{t_0+6}(\mathsf{pUnitReference}_{\mathsf{Dutch}}) = \mathsf{pUnitService}_{\mathsf{Dutch}} \tag{A.118}$$

A.8.6 Dutch P-Unit

$$\mathsf{Contains}_{s_{ae}}^{t_0+6}(\mathsf{pUnit}_{\mathsf{Dutch}}) = \{\mathsf{pConfiguration1}_{\mathsf{Dutch}}\} \tag{A.119}$$

$$\begin{aligned} &\mathsf{Provides}_{s_{ae}}^{t_0+6}(\mathsf{pConfiguration1}_{\mathsf{Dutch}}) = \{\mathsf{pUnitService}_{\mathsf{Dutch}}\} \\ &\mathsf{Declares}_{s_{ae}}^{t_0+6}(\mathsf{pConfiguration1}_{\mathsf{Dutch}}) = \emptyset \end{aligned} \tag{A.120}$$

$$\mathsf{Current}_{s_{ae}}^{t_0+6}(\mathsf{pUnit}_{\mathsf{Dutch}}) = \mathsf{pConfiguration1}_{\mathsf{Dutch}} \tag{A.121}$$

A.8.7 Semantical Compatibility

$$\mathsf{ServiceBehavior}_{s_{ae}}^{t_0+6}(\mathsf{pUnitService}_{\mathsf{German}}) = \mathsf{pUnitServiceUsualOperation}_{\mathsf{German}} \tag{A.122}$$

$$\mathsf{ServiceBehavior}_{s_{ae}}^{t_0+6}(\mathsf{pUnitService}_{\mathsf{Dutch}}) = \mathsf{pUnitServiceUsualOperation}_{\mathsf{Dutch}} \tag{A.123}$$

$$\mathsf{ServiceReferenceBehavior}_{s_{ae}}^{t_0+6}(\mathsf{pUnitReference}_{\mathsf{German}}, \mathsf{pUnitService}_{\mathsf{German}}) = \mathsf{pUnitReferenceUsualOperation}_{\mathsf{German}} \tag{A.124}$$

$$\mathsf{ServiceReferenceBehavior}_{s_{ae}}^{t_0+6}(\mathsf{pUnitReference}_{\mathsf{German}}, \mathsf{pUnitService}_{\mathsf{Dutch}}) = \mathsf{pUnitReferenceUsualOperation}_{\mathsf{German}} \tag{A.125}$$

$$\mathsf{ServiceReferenceBehavior}_{s_{ae}}^{t_0+6}(\mathsf{pUnitReference}_{\mathsf{Dutch}}, \mathsf{pUnitService}_{\mathsf{German}}) = \mathsf{pUnitReferenceUsualOperation}_{\mathsf{Dutch}} \tag{A.126}$$

$$\mathsf{ServiceReferenceBehavior}_{s_{ae}}^{t_0+6}(\mathsf{pUnitReference}_{\mathsf{Dutch}}, \mathsf{pUnitService}_{\mathsf{Dutch}}) = \mathsf{pUnitReferenceUsualOperation}_{\mathsf{Dutch}} \tag{A.127}$$

$$\mathsf{pUnitReference}_{\mathsf{German}} \simeq_{\mathsf{Semantical}_{s_{ae}}^{t_0+6}} \mathsf{pUnitService}_{\mathsf{German}} \tag{A.128}$$

$$\mathsf{pUnitReference}_{\mathsf{German}} \simeq_{\mathsf{Semantical}_{s_{ae}}^{t_0+6}} \mathsf{pUnitService}_{\mathsf{Dutch}} \tag{A.129}$$

$$\mathsf{pUnitReference}_{\mathsf{Dutch}} \simeq_{\mathsf{Semantical}_{s_{ae}}^{t_0+6}} \mathsf{pUnitService}_{\mathsf{German}} \tag{A.130}$$

$$\mathsf{pUnitReference}_{\mathsf{Dutch}} \simeq_{\mathsf{Semantical}_{s_{ae}}^{t_0+6}} \mathsf{pUnitService}_{\mathsf{Dutch}} \tag{A.131}$$

A.9 Instances at Dependability Checkpoint t_n

At t_n, we face the situation that a second Dutch C-Unit is started in our Dependable Dynamic Adaptive System s_ae to represent a third casualty.

$$\begin{aligned}\mathsf{ApplicationComponents}_{s_{ae}}^{t_n} = \{&\mathsf{mUnit}_{\mathsf{German}}, \mathsf{cUnit}_{\mathsf{German}}, \mathsf{pUnit}_{\mathsf{German}},\\ &\mathsf{mUnit}_{\mathsf{Dutch}}, \mathsf{cUnit}_{\mathsf{Dutch}}, \mathsf{pUnit}_{\mathsf{Dutch}},\\ &\mathsf{cUnit2}_{\mathsf{Dutch}}\}\end{aligned} \tag{A.132}$$

A.9.1 German M-Unit

$$\begin{aligned}\mathsf{Contains}_{s_{ae}}^{t_n}(\mathsf{mUnit}_{\mathsf{German}}) = \{&\mathsf{mConfiguration1}_{\mathsf{German}},\\ &\mathsf{mConfiguration2}_{\mathsf{German}}\}\end{aligned} \tag{A.133}$$

$$\mathsf{mConfiguration2}_{\mathsf{German}} \geq_{s_{ae}}^{t_n} \mathsf{mConfiguration1}_{\mathsf{German}} \tag{A.134}$$

$$\begin{aligned}&\mathsf{Provides}^{t_n}_{s_{ae}}(\mathsf{mConfiguration1}_{\mathsf{German}}) = \{\mathsf{mUnitService}_{\mathsf{German}}\}\\&\mathsf{Declares}^{t_n}_{s_{ae}}(\mathsf{mConfiguration1}_{\mathsf{German}}) = \emptyset\\&\mathsf{Provides}^{t_n}_{s_{ae}}(\mathsf{mConfiguration2}_{\mathsf{German}}) = \{\mathsf{mUnitService}_{\mathsf{German}}\}\\&\mathsf{Declares}^{t_n}_{s_{ae}}(\mathsf{mConfiguration2}_{\mathsf{German}}) = \{\mathsf{cUnitReference}_{\mathsf{German}}\}\end{aligned} \tag{A.135}$$

$$\mathsf{Current}^{t_n}_{s_{ae}}(\mathsf{mUnit}_{\mathsf{German}}) = \mathsf{mConfiguration2}_{\mathsf{German}} \tag{A.136}$$

$$\mathsf{Uses}^{t_n}_{s_{ae}}(\mathsf{cUnitReference}_{\mathsf{German}}) = \mathsf{cUnitService}_{\mathsf{German}} \tag{A.137}$$

A.9.2 German C-Unit

$$\mathsf{Contains}^{t_n}_{s_{ae}}(\mathsf{cUnit}_{\mathsf{German}}) = \{\mathsf{cConfiguration1}_{\mathsf{German}}, \mathsf{cConfiguration2}_{\mathsf{German}}\} \tag{A.138}$$

$$\mathsf{cConfiguration2}_{\mathsf{German}} \geq^{t_n}_{s_{ae}} \mathsf{cConfiguration1}_{\mathsf{German}} \tag{A.139}$$

$$\begin{aligned}&\mathsf{Provides}^{t_n}_{s_{ae}}(\mathsf{cConfiguration1}_{\mathsf{German}}) = \{\mathsf{cUnitService}_{\mathsf{German}}\}\\&\mathsf{Declares}^{t_n}_{s_{ae}}(\mathsf{cConfiguration1}_{\mathsf{German}}) = \emptyset\\&\mathsf{Provides}^{t_n}_{s_{ae}}(\mathsf{cConfiguration2}_{\mathsf{German}}) = \{\mathsf{cUnitService}_{\mathsf{German}}\}\\&\mathsf{Declares}^{t_n}_{s_{ae}}(\mathsf{cConfiguration2}_{\mathsf{German}}) = \{\mathsf{pUnitReference}_{\mathsf{German}}\}\end{aligned} \tag{A.140}$$

$$\mathsf{Current}^{t_n}_{s_{ae}}(\mathsf{cUnit}_{\mathsf{German}}) = \mathsf{cConfiguration2}_{\mathsf{German}} \tag{A.141}$$

$$\mathsf{Uses}^{t_n}_{s_{ae}}(\mathsf{pUnitReference}_{\mathsf{German}}) = \mathsf{pUnitService}_{\mathsf{German}} \tag{A.142}$$

A.9.3 German P-Unit

$$\mathsf{Contains}^{t_n}_{s_{ae}}(\mathsf{pUnit}_{\mathsf{German}}) = \{\mathsf{pConfiguration1}_{\mathsf{German}}\} \tag{A.143}$$

$$\begin{aligned}&\mathsf{Provides}^{t_n}_{s_{ae}}(\mathsf{pConfiguration1}_{\mathsf{German}}) = \{\mathsf{pUnitService}_{\mathsf{German}}\}\\&\mathsf{Declares}^{t_n}_{s_{ae}}(\mathsf{pConfiguration1}_{\mathsf{German}}) = \emptyset\end{aligned} \tag{A.144}$$

$$\mathsf{Current}^{t_n}_{s_{ae}}(\mathsf{pUnit}_{\mathsf{German}}) = \mathsf{pConfiguration1}_{\mathsf{German}} \tag{A.145}$$

A.9.4 Dutch M-Unit

$$\mathsf{Contains}^{t_n}_{s_{ae}}(\mathsf{mUnit}_{\mathsf{Dutch}}) = \{\mathsf{mConfiguration1}_{\mathsf{Dutch}}, \mathsf{mConfiguration2}_{\mathsf{Dutch}}\} \tag{A.146}$$

$$\mathsf{mConfiguration2}_{\mathsf{Dutch}} \geq^{t_n}_{s_{ae}} \mathsf{mConfiguration1}_{\mathsf{Dutch}} \tag{A.147}$$

$$\begin{aligned} \mathsf{Provides}^{t_n}_{s_{ae}}(\mathsf{mConfiguration1}_{\mathsf{Dutch}}) &= \{\mathsf{mUnitService}_{\mathsf{Dutch}}\} \\ \mathsf{Declares}^{t_n}_{s_{ae}}(\mathsf{mConfiguration1}_{\mathsf{Dutch}}) &= \emptyset \\ \mathsf{Provides}^{t_n}_{s_{ae}}(\mathsf{mConfiguration2}_{\mathsf{Dutch}}) &= \{\mathsf{mUnitService}_{\mathsf{Dutch}}\} \\ \mathsf{Declares}^{t_n}_{s_{ae}}(\mathsf{mConfiguration2}_{\mathsf{Dutch}}) &= \{\mathsf{cUnitReference}_{\mathsf{Dutch}}\} \end{aligned} \tag{A.148}$$

$$\mathsf{Current}^{t_n}_{s_{ae}}(\mathsf{mUnit}_{\mathsf{Dutch}}) = \mathsf{mConfiguration2}_{\mathsf{Dutch}} \tag{A.149}$$

$$\mathsf{Uses}^{t_n}_{s_{ae}}(\mathsf{cUnitReference}_{\mathsf{Dutch}}) = \mathsf{cUnitService}_{\mathsf{Dutch}}2 \tag{A.150}$$

A.9.5 Dutch C-Unit

$$\mathsf{Contains}^{t_n}_{s_{ae}}(\mathsf{cUnit}_{\mathsf{Dutch}}) = \{\mathsf{cConfiguration1}_{\mathsf{Dutch}}, \mathsf{cConfiguration2}_{\mathsf{Dutch}}\} \tag{A.151}$$

$$\mathsf{cConfiguration2}_{\mathsf{Dutch}} \geq^{t_n}_{s_{ae}} \mathsf{cConfiguration1}_{\mathsf{Dutch}} \tag{A.152}$$

$$\begin{aligned} \mathsf{Provides}^{t_n}_{s_{ae}}(\mathsf{cConfiguration1}_{\mathsf{Dutch}}) &= \{\mathsf{cUnitService}_{\mathsf{Dutch}}\} \\ \mathsf{Declares}^{t_n}_{s_{ae}}(\mathsf{cConfiguration1}_{\mathsf{Dutch}}) &= \emptyset \\ \mathsf{Provides}^{t_n}_{s_{ae}}(\mathsf{cConfiguration2}_{\mathsf{Dutch}}) &= \{\mathsf{cUnitService}_{\mathsf{Dutch}}\} \\ \mathsf{Declares}^{t_n}_{s_{ae}}(\mathsf{cConfiguration2}_{\mathsf{Dutch}}) &= \{\mathsf{pUnitReference}_{\mathsf{Dutch}}\} \end{aligned} \tag{A.153}$$

$$\mathsf{Current}^{t_n}_{s_{ae}}(\mathsf{cUnit}_{\mathsf{Dutch}}) = \mathsf{cConfiguration2}_{\mathsf{Dutch}} \tag{A.154}$$

$$\mathsf{Uses}^{t_n}_{s_{ae}}(\mathsf{pUnitReference}_{\mathsf{Dutch}}) = \mathsf{pUnitService}_{\mathsf{Dutch}} \tag{A.155}$$

A.9.6 Second Dutch C-Unit

$$\mathsf{Contains}^{t_n}_{s_{ae}}(\mathsf{cUnit}_{\mathsf{Dutch}}2) = \{\mathsf{cConfiguration1}_{\mathsf{Dutch}}2, \mathsf{cConfiguration2}_{\mathsf{Dutch}}2\} \tag{A.156}$$

$$\mathsf{cConfiguration2}_{\mathsf{Dutch}}2 \geq^{t_n}_{s_{ae}} \mathsf{cConfiguration1}_{\mathsf{Dutch}}2 \tag{A.157}$$

$$\begin{aligned}\mathsf{Provides}^{t_n}_{s_{ae}}(\mathsf{cConfiguration1}_{\mathsf{Dutch}}2) &= \{\mathsf{cUnitService}_{\mathsf{Dutch}}2\}\\ \mathsf{Declares}^{t_n}_{s_{ae}}(\mathsf{cConfiguration1}_{\mathsf{Dutch}}2) &= \emptyset\\ \mathsf{Provides}^{t_n}_{s_{ae}}(\mathsf{cConfiguration2}_{\mathsf{Dutch}}2) &= \{\mathsf{cUnitService}_{\mathsf{Dutch}}2\}\\ \mathsf{Declares}^{t_n}_{s_{ae}}(\mathsf{cConfiguration2}_{\mathsf{Dutch}}2) &= \{\mathsf{pUnitReference}_{\mathsf{Dutch}}2\}\end{aligned} \tag{A.158}$$

$$\mathsf{Current}^{t_n}_{s_{ae}}(\mathsf{cUnit}_{\mathsf{Dutch}}2) = \mathsf{cConfiguration1}_{\mathsf{Dutch}}2 \tag{A.159}$$

$$\mathsf{Uses}^{t_n}_{s_{ae}}(\mathsf{pUnitReference}_{\mathsf{Dutch}}2) = \emptyset \tag{A.160}$$

A.9.7 Dutch P-Unit

$$\mathsf{Contains}^{t_n}_{s_{ae}}(\mathsf{pUnit}_{\mathsf{Dutch}}) = \{\mathsf{pConfiguration1}_{\mathsf{Dutch}}\} \tag{A.161}$$

$$\begin{aligned}\mathsf{Provides}^{t_n}_{s_{ae}}(\mathsf{pConfiguration1}_{\mathsf{Dutch}}) &= \{\mathsf{pUnitService}_{\mathsf{Dutch}}\}\\ \mathsf{Declares}^{t_n}_{s_{ae}}(\mathsf{pConfiguration1}_{\mathsf{Dutch}}) &= \emptyset\end{aligned} \tag{A.162}$$

$$\mathsf{Current}^{t_n}_{s_{ae}}(\mathsf{pUnit}_{\mathsf{Dutch}}) = \mathsf{pConfiguration1}_{\mathsf{Dutch}} \tag{A.163}$$

A.9.8 Semantical Compatibility

$$\mathsf{ServiceBehavior}^{t_n}_{s_{ae}}(\mathsf{pUnitService}_{\mathsf{German}}) = \mathsf{pUnitServiceUsualOperation}_{\mathsf{German}} \tag{A.164}$$

$$\mathsf{ServiceBehavior}^{t_n}_{s_{ae}}(\mathsf{pUnitService}_{\mathsf{Dutch}}) = \mathsf{pUnitServiceUsualOperation}_{\mathsf{Dutch}} \tag{A.165}$$

$$\text{ServiceReferenceBehavior}_{s_{ae}}^{t_n}(\text{pUnitReference}_{\text{German}}, \text{pUnitService}_{\text{German}}) = \text{pUnitReferenceUsualOperation}_{\text{German}} \tag{A.166}$$

$$\text{ServiceReferenceBehavior}_{s_{ae}}^{t_n}(\text{pUnitReference}_{\text{German}}, \text{pUnitService}_{\text{Dutch}}) = \text{pUnitReferenceUsualOperation}_{\text{German}} \tag{A.167}$$

$$\text{ServiceReferenceBehavior}_{s_{ae}}^{t_n}(\text{pUnitReference}_{\text{Dutch}}, \text{pUnitService}_{\text{German}}) = \text{pUnitReferenceUsualOperation}_{\text{Dutch}} \tag{A.168}$$

$$\text{ServiceReferenceBehavior}_{s_{ae}}^{t_n}(\text{pUnitReference}_{\text{Dutch}}, \text{pUnitService}_{\text{Dutch}}) = \text{pUnitReferenceUsualOperation}_{\text{Dutch}} \tag{A.169}$$

$$\text{ServiceReferenceBehavior}_{s_{ae}}^{t_n}(\text{pUnitReference}_{\text{Dutch}}2, \text{pUnitService}_{\text{German}}) = \text{pUnitReferenceUsualOperation}_{\text{Dutch}} \tag{A.170}$$

$$\text{ServiceReferenceBehavior}_{s_{ae}}^{t_n}(\text{pUnitReference}_{\text{Dutch}}2, \text{pUnitService}_{\text{Dutch}}) = \text{pUnitReferenceUsualOperation}_{\text{Dutch}} \tag{A.171}$$

$$\text{pUnitReference}_{\text{German}} \simeq_{\text{Semantical}_{s_{ae}}^{t_n}} \text{pUnitService}_{\text{German}} \tag{A.172}$$

$$\text{pUnitReference}_{\text{German}} \simeq_{\text{Semantical}_{s_{ae}}^{t_n}} \text{pUnitService}_{\text{Dutch}} \tag{A.173}$$

$$\text{pUnitReference}_{\text{Dutch}} \simeq_{\text{Semantical}_{s_{ae}}^{t_n}} \text{pUnitService}_{\text{German}} \tag{A.174}$$

$$\text{pUnitReference}_{\text{Dutch}} \simeq_{\text{Semantical}_{s_{ae}}^{t_n}} \text{pUnitService}_{\text{Dutch}} \tag{A.175}$$

$$\text{pUnitReference}_{\text{Dutch}}2 \simeq_{\text{Semantical}_{s_{ae}}^{t_n}} \text{pUnitService}_{\text{German}} \tag{A.176}$$

$$\text{pUnitReference}_{\text{Dutch}}2 \simeq_{\text{Semantical}_{s_{ae}}^{t_n}} \text{pUnitService}_{\text{Dutch}} \tag{A.177}$$

A.10 Instances at Dependability Checkpoint $t_n + 1$

At t_n+1, we face the situation that a German P-Unit is attached to the Dutch C-Unit introduced at t_n in our Dependable Dynamic Adaptive System s_ae.

$$\begin{aligned}\text{ApplicationComponents}_{s_{ae}}^{t_n+1} = \{&\text{mUnit}_{\text{German}}, \text{cUnit}_{\text{German}}, \text{pUnit}_{\text{German}},\\ &\text{mUnit}_{\text{Dutch}}, \text{cUnit}_{\text{Dutch}}, \text{pUnit}_{\text{Dutch}},\\ &\text{cUnit2}_{\text{Dutch}}, \text{pUnit2}_{\text{German}}\}\end{aligned} \tag{A.178}$$

A.10.1 German M-Unit

$$\begin{aligned}\text{Contains}_{s_{ae}}^{t_n+1}(\text{mUnit}_{\text{German}}) = \{&\text{mConfiguration1}_{\text{German}},\\ &\text{mConfiguration2}_{\text{German}}\}\end{aligned} \tag{A.179}$$

$$\text{mConfiguration2}_{\text{German}} \geq_{s_{ae}}^{t_n+1} \text{mConfiguration1}_{\text{German}} \tag{A.180}$$

$$\begin{aligned}&\text{Provides}_{s_{ae}}^{t_n+1}(\text{mConfiguration1}_{\text{German}}) = \{\text{mUnitService}_{\text{German}}\}\\ &\text{Declares}_{s_{ae}}^{t_n+1}(\text{mConfiguration1}_{\text{German}}) = \emptyset\\ &\text{Provides}_{s_{ae}}^{t_n+1}(\text{mConfiguration2}_{\text{German}}) = \{\text{mUnitService}_{\text{German}}\}\\ &\text{Declares}_{s_{ae}}^{t_n+1}(\text{mConfiguration2}_{\text{German}}) = \{\text{cUnitReference}_{\text{German}}\}\end{aligned} \tag{A.181}$$

$$\text{Current}_{s_{ae}}^{t_n+1}(\text{mUnit}_{\text{German}}) = \text{mConfiguration2}_{\text{German}} \tag{A.182}$$

$$\text{Uses}_{s_{ae}}^{t_n+1}(\text{cUnitReference}_{\text{German}}) = \text{cUnitService}_{\text{German}} \tag{A.183}$$

A.10.2 German C-Unit

$$\begin{aligned}\text{Contains}_{s_{ae}}^{t_n+1}(\text{cUnit}_{\text{German}}) = \{&\text{cConfiguration1}_{\text{German}},\\ &\text{cConfiguration2}_{\text{German}}\}\end{aligned} \tag{A.184}$$

$$\text{cConfiguration2}_{\text{German}} \geq_{s_{ae}}^{t_n+1} \text{cConfiguration1}_{\text{German}} \tag{A.185}$$

$$\begin{aligned}&\text{Provides}_{s_{ae}}^{t_n+1}(\text{cConfiguration1}_{\text{German}}) = \{\text{cUnitService}_{\text{German}}\}\\ &\text{Declares}_{s_{ae}}^{t_n+1}(\text{cConfiguration1}_{\text{German}}) = \emptyset\\ &\text{Provides}_{s_{ae}}^{t_n+1}(\text{cConfiguration2}_{\text{German}}) = \{\text{cUnitService}_{\text{German}}\}\\ &\text{Declares}_{s_{ae}}^{t_n+1}(\text{cConfiguration2}_{\text{German}}) = \{\text{pUnitReference}_{\text{German}}\}\end{aligned} \tag{A.186}$$

$$\text{Current}_{s_{ae}}^{t_n+1}(\text{cUnit}_{\text{German}}) = \text{cConfiguration2}_{\text{German}} \tag{A.187}$$

$$\text{Uses}_{s_{ae}}^{t_n+1}(\text{pUnitReference}_{\text{German}}) = \text{pUnitService}_{\text{German}} \tag{A.188}$$

A.10.3 German P-Unit

$$\mathsf{Contains}_{s_{ae}}^{t_n+1}(\mathsf{pUnit}_{\mathsf{German}}) = \{\mathsf{pConfiguration1}_{\mathsf{German}}\} \tag{A.189}$$

$$\begin{aligned}\mathsf{Provides}_{s_{ae}}^{t_n+1}(\mathsf{pConfiguration1}_{\mathsf{German}}) &= \{\mathsf{pUnitService}_{\mathsf{German}}\}\\ \mathsf{Declares}_{s_{ae}}^{t_n+1}(\mathsf{pConfiguration1}_{\mathsf{German}}) &= \emptyset\end{aligned} \tag{A.190}$$

$$\mathsf{Current}_{s_{ae}}^{t_n+1}(\mathsf{pUnit}_{\mathsf{German}}) = \mathsf{pConfiguration1}_{\mathsf{German}} \tag{A.191}$$

A.10.4 Second German P-Unit

$$\mathsf{Contains}_{s_{ae}}^{t_n+1}(\mathsf{pUnit}_{\mathsf{German}}2) = \{\mathsf{pConfiguration1}_{\mathsf{German}}2\} \tag{A.192}$$

$$\begin{aligned}\mathsf{Provides}_{s_{ae}}^{t_n+1}(\mathsf{pConfiguration1}_{\mathsf{German}}2) &= \{\mathsf{pUnitService}_{\mathsf{German}}2\}\\ \mathsf{Declares}_{s_{ae}}^{t_n+1}(\mathsf{pConfiguration1}_{\mathsf{German}}2) &= \emptyset\end{aligned} \tag{A.193}$$

$$\mathsf{Current}_{s_{ae}}^{t_n+1}(\mathsf{pUnit}_{\mathsf{German}}2) = \mathsf{pConfiguration1}_{\mathsf{German}}2 \tag{A.194}$$

A.10.5 Dutch M-Unit

$$\begin{aligned}\mathsf{Contains}_{s_{ae}}^{t_n+1}(\mathsf{mUnit}_{\mathsf{Dutch}}) = \{&\mathsf{mConfiguration1}_{\mathsf{Dutch}},\\ &\mathsf{mConfiguration2}_{\mathsf{Dutch}}\}\end{aligned} \tag{A.195}$$

$$\mathsf{mConfiguration2}_{\mathsf{Dutch}} \geq_{s_{ae}}^{t_n+1} \mathsf{mConfiguration1}_{\mathsf{Dutch}} \tag{A.196}$$

$$\begin{aligned}\mathsf{Provides}_{s_{ae}}^{t_n+1}(\mathsf{mConfiguration1}_{\mathsf{Dutch}}) &= \{\mathsf{mUnitService}_{\mathsf{Dutch}}\}\\ \mathsf{Declares}_{s_{ae}}^{t_n+1}(\mathsf{mConfiguration1}_{\mathsf{Dutch}}) &= \emptyset\\ \mathsf{Provides}_{s_{ae}}^{t_n+1}(\mathsf{mConfiguration2}_{\mathsf{Dutch}}) &= \{\mathsf{mUnitService}_{\mathsf{Dutch}}\}\\ \mathsf{Declares}_{s_{ae}}^{t_n+1}(\mathsf{mConfiguration2}_{\mathsf{Dutch}}) &= \{\mathsf{cUnitReference}_{\mathsf{Dutch}}\}\end{aligned} \tag{A.197}$$

$$\mathsf{Current}_{s_{ae}}^{t_n+1}(\mathsf{mUnit}_{\mathsf{Dutch}}) = \mathsf{mConfiguration2}_{\mathsf{Dutch}} \tag{A.198}$$

$$\mathsf{Uses}_{s_{ae}}^{t_n+1}(\mathsf{cUnitReference}_{\mathsf{Dutch}}) = \mathsf{cUnitService}_{\mathsf{Dutch}}2 \tag{A.199}$$

A.10.6 Dutch C-Unit

$$\mathsf{Contains}_{s_{ae}}^{t_n+1}(\mathsf{cUnit}_{\mathsf{Dutch}}) = \{\mathsf{cConfiguration1}_{\mathsf{Dutch}}, \mathsf{cConfiguration2}_{\mathsf{Dutch}}\} \tag{A.200}$$

$$\mathsf{cConfiguration2}_{\mathsf{Dutch}} \geq_{s_{ae}}^{t_n+1} \mathsf{cConfiguration1}_{\mathsf{Dutch}} \tag{A.201}$$

$$\begin{aligned}
&\mathsf{Provides}_{s_{ae}}^{t_n+1}(\mathsf{cConfiguration1}_{\mathsf{Dutch}}) = \{\mathsf{cUnitService}_{\mathsf{Dutch}}\} \\
&\mathsf{Declares}_{s_{ae}}^{t_n+1}(\mathsf{cConfiguration1}_{\mathsf{Dutch}}) = \emptyset \\
&\mathsf{Provides}_{s_{ae}}^{t_n+1}(\mathsf{cConfiguration2}_{\mathsf{Dutch}}) = \{\mathsf{cUnitService}_{\mathsf{Dutch}}\} \\
&\mathsf{Declares}_{s_{ae}}^{t_n+1}(\mathsf{cConfiguration2}_{\mathsf{Dutch}}) = \{\mathsf{pUnitReference}_{\mathsf{Dutch}}\}
\end{aligned} \tag{A.202}$$

$$\mathsf{Current}_{s_{ae}}^{t_n+1}(\mathsf{cUnit}_{\mathsf{Dutch}}) = \mathsf{cConfiguration2}_{\mathsf{Dutch}} \tag{A.203}$$

$$\mathsf{Uses}_{s_{ae}}^{t_n+1}(\mathsf{pUnitReference}_{\mathsf{Dutch}}) = \mathsf{pUnitService}_{\mathsf{Dutch}} \tag{A.204}$$

A.10.7 Second Dutch C-Unit

$$\begin{aligned}
\mathsf{Contains}_{s_{ae}}^{t_n+1}(\mathsf{cUnit}_{\mathsf{Dutch}}2) = \{&\mathsf{cConfiguration1}_{\mathsf{Dutch}}2, \\
&\mathsf{cConfiguration2}_{\mathsf{Dutch}}2\}
\end{aligned} \tag{A.205}$$

$$\mathsf{cConfiguration2}_{\mathsf{Dutch}}2 \geq_{s_{ae}}^{t_n+1} \mathsf{cConfiguration1}_{\mathsf{Dutch}}2 \tag{A.206}$$

$$\begin{aligned}
&\mathsf{Provides}_{s_{ae}}^{t_n+1}(\mathsf{cConfiguration1}_{\mathsf{Dutch}}2) = \{\mathsf{cUnitService}_{\mathsf{Dutch}}2\} \\
&\mathsf{Declares}_{s_{ae}}^{t_n+1}(\mathsf{cConfiguration1}_{\mathsf{Dutch}}2) = \emptyset \\
&\mathsf{Provides}_{s_{ae}}^{t_n+1}(\mathsf{cConfiguration2}_{\mathsf{Dutch}}2) = \{\mathsf{cUnitService}_{\mathsf{Dutch}}2\} \\
&\mathsf{Declares}_{s_{ae}}^{t_n+1}(\mathsf{cConfiguration2}_{\mathsf{Dutch}}2) = \{\mathsf{pUnitReference}_{\mathsf{Dutch}}2\}
\end{aligned} \tag{A.207}$$

$$\mathsf{Current}_{s_{ae}}^{t_n+1}(\mathsf{cUnit}_{\mathsf{Dutch}}2) = \mathsf{cConfiguration2}_{\mathsf{Dutch}}2 \tag{A.208}$$

$$\mathsf{Uses}_{s_{ae}}^{t_n+1}(\mathsf{pUnitReference}_{\mathsf{Dutch}}2) = \mathsf{pUnitService}_{\mathsf{German}}2 \tag{A.209}$$

A.10.8 Dutch P-Unit

$$\text{Contains}_{s_{ae}}^{t_n+1}(\text{pUnit}_{\text{Dutch}}) = \{\text{pConfiguration1}_{\text{Dutch}}\} \tag{A.210}$$

$$\begin{aligned}\text{Provides}_{s_{ae}}^{t_n+1}(\text{pConfiguration1}_{\text{Dutch}}) &= \{\text{pUnitService}_{\text{Dutch}}\}\\ \text{Declares}_{s_{ae}}^{t_n+1}(\text{pConfiguration1}_{\text{Dutch}}) &= \emptyset\end{aligned} \tag{A.211}$$

$$\text{Current}_{s_{ae}}^{t_n+1}(\text{pUnit}_{\text{Dutch}}) = \text{pConfiguration1}_{\text{Dutch}} \tag{A.212}$$

A.10.9 Semantical Compatibility

$$\text{ServiceBehavior}_{s_{ae}}^{t_n+1}(\text{pUnitService}_{\text{German}}) = \text{pUnitServiceUsualOperation}_{\text{German}} \tag{A.213}$$

$$\text{ServiceBehavior}_{s_{ae}}^{t_n+1}(\text{pUnitService}_{\text{Dutch}}) = \text{pUnitServiceUsualOperation}_{\text{Dutch}} \tag{A.214}$$

$$\text{ServiceBehavior}_{s_{ae}}^{t_n+1}(\text{pUnitService}_{\text{German}}2) = \text{pUnitServiceUsualOperation}_{\text{German}} \tag{A.215}$$

$$\text{ServiceReferenceBehavior}_{s_{ae}}^{t_n+1}(\text{pUnitReference}_{\text{German}}, \text{pUnitService}_{\text{German}}) = \text{pUnitReferenceUsualOperation}_{\text{German}} \tag{A.216}$$

$$\text{ServiceReferenceBehavior}_{s_{ae}}^{t_n+1}(\text{pUnitReference}_{\text{German}}, \text{pUnitService}_{\text{Dutch}}) = \text{pUnitReferenceUsualOperation}_{\text{German}} \tag{A.217}$$

$$\text{ServiceReferenceBehavior}_{s_{ae}}^{t_n+1}(\text{pUnitReference}_{\text{German}}, \text{pUnitService}_{\text{German}}2) = \text{pUnitReferenceUsualOperation}_{\text{German}} \tag{A.218}$$

$$\text{ServiceReferenceBehavior}_{s_{ae}}^{t_n+1}(\text{pUnitReference}_{\text{Dutch}}, \text{pUnitService}_{\text{German}}) = \text{pUnitReferenceUsualOperation}_{\text{Dutch}} \tag{A.219}$$

$$\text{ServiceReferenceBehavior}_{s_{ae}}^{t_n+1}(\text{pUnitReference}_{\text{Dutch}}, \text{pUnitService}_{\text{Dutch}}) = \text{pUnitReferenceUsualOperation}_{\text{Dutch}} \tag{A.220}$$

$$\text{ServiceReferenceBehavior}_{s_{ae}}^{t_n+1}(\text{pUnitReference}_{\text{Dutch}}, \text{pUnitService}_{\text{German}}2) = \text{pUnitReferenceUsualOperation}_{\text{Dutch}} \tag{A.221}$$

$$\text{ServiceReferenceBehavior}_{s_{ae}}^{t_n+1}(\text{pUnitReference}_{\text{Dutch}}2, \text{pUnitService}_{\text{German}}) = \text{pUnitReferenceUsualOperation}_{\text{Dutch}} \tag{A.222}$$

$$\text{ServiceReferenceBehavior}_{s_{ae}}^{t_n+1}(\text{pUnitReference}_{\text{Dutch}}2, \text{pUnitService}_{\text{Dutch}}) = \text{pUnitReferenceUsualOperation}_{\text{Dutch}} \tag{A.223}$$

$$\text{ServiceReferenceBehavior}_{s_{ae}}^{t_n+1}(\text{pUnitReference}_{\text{Dutch}}2, \text{pUnitService}_{\text{German}}2) = \text{pUnitReferenceUsualOperation}_{\text{Dutch}} \tag{A.224}$$

$$\text{pUnitReference}_{\text{German}} \simeq_{\text{Semantical}_{s_{ae}}^{t_n+1}} \text{pUnitService}_{\text{German}} \tag{A.225}$$

$$\text{pUnitReference}_{\text{German}} \simeq_{\text{Semantical}_{s_{ae}}^{t_n+1}} \text{pUnitService}_{\text{Dutch}} \tag{A.226}$$

$$\text{pUnitReference}_{\text{German}} \simeq_{\text{Semantical}_{s_{ae}}^{t_n+1}} \text{pUnitService}_{\text{German}}2 \tag{A.227}$$

$$\text{pUnitReference}_{\text{Dutch}} \simeq_{\text{Semantical}_{s_{ae}}^{t_n+1}} \text{pUnitService}_{\text{German}} \tag{A.228}$$

$$\text{pUnitReference}_{\text{Dutch}} \simeq_{\text{Semantical}_{s_{ae}}^{t_n+1}} \text{pUnitService}_{\text{Dutch}} \tag{A.229}$$

$$\text{pUnitReference}_{\text{Dutch}} \simeq_{\text{Semantical}_{s_{ae}}^{t_n+1}} \text{pUnitService}_{\text{German}}2 \tag{A.230}$$

$$\text{pUnitReference}_{\text{Dutch}}2 \simeq_{\text{Semantical}_{s_{ae}}^{t_n+1}} \text{pUnitService}_{\text{German}} \tag{A.231}$$

$$\text{pUnitReference}_{\text{Dutch}}2 \simeq_{\text{Semantical}_{s_{ae}}^{t_n+1}} \text{pUnitService}_{\text{Dutch}} \tag{A.232}$$

$$\text{pUnitReference}_{\text{Dutch}}2 \simeq_{\text{Semantical}_{s_{ae}}^{t_n+1}} \text{pUnitService}_{\text{German}}2 \tag{A.233}$$

A.11 Instances at Dependability Checkpoint $t_n + 2$

At $t_n + 2$, we face the situation that the fingerclip of the casualty at the German P-Unit, which has been introduced at t_n in our Dependable Dynamic Adaptive System $s_a e$ slips off. Thus, the compatibility between this P-Unit and the Dutch C-Units changes and the Component Binding between Dutch C-Unit and German P-Unit is removed.

$$\begin{aligned}\mathsf{ApplicationComponents}_{s_{ae}}^{t_n+2} = \{&\mathsf{mUnit}_{\mathsf{German}}, \mathsf{cUnit}_{\mathsf{German}}, \mathsf{pUnit}_{\mathsf{German}},\\ &\mathsf{mUnit}_{\mathsf{Dutch}}, \mathsf{cUnit}_{\mathsf{Dutch}}, \mathsf{pUnit}_{\mathsf{Dutch}},\\ &\mathsf{cUnit2}_{\mathsf{Dutch}}, \mathsf{pUnit2}_{\mathsf{German}}\}\end{aligned} \tag{A.234}$$

A.11.1 German M-Unit

$$\begin{aligned}\mathsf{Contains}_{s_{ae}}^{t_n+2}(\mathsf{mUnit}_{\mathsf{German}}) = \{&\mathsf{mConfiguration1}_{\mathsf{German}},\\ &\mathsf{mConfiguration2}_{\mathsf{German}}\}\end{aligned} \tag{A.235}$$

$$\mathsf{mConfiguration2}_{\mathsf{German}} \geq_{s_{ae}}^{t_n+2} \mathsf{mConfiguration1}_{\mathsf{German}} \tag{A.236}$$

$$\begin{aligned}\mathsf{Provides}_{s_{ae}}^{t_n+2}(\mathsf{mConfiguration1}_{\mathsf{German}}) &= \{\mathsf{mUnitService}_{\mathsf{German}}\}\\ \mathsf{Declares}_{s_{ae}}^{t_n+2}(\mathsf{mConfiguration1}_{\mathsf{German}}) &= \emptyset\\ \mathsf{Provides}_{s_{ae}}^{t_n+2}(\mathsf{mConfiguration2}_{\mathsf{German}}) &= \{\mathsf{mUnitService}_{\mathsf{German}}\}\\ \mathsf{Declares}_{s_{ae}}^{t_n+2}(\mathsf{mConfiguration2}_{\mathsf{German}}) &= \{\mathsf{cUnitReference}_{\mathsf{German}}\}\end{aligned} \tag{A.237}$$

$$\mathsf{Current}_{s_{ae}}^{t_n+2}(\mathsf{mUnit}_{\mathsf{German}}) = \mathsf{mConfiguration2}_{\mathsf{German}} \tag{A.238}$$

$$\mathsf{Uses}_{s_{ae}}^{t_n+2}(\mathsf{cUnitReference}_{\mathsf{German}}) = \mathsf{cUnitService}_{\mathsf{German}} \tag{A.239}$$

A.11.2 German C-Unit

$$\begin{aligned}\mathsf{Contains}_{s_{ae}}^{t_n+2}(\mathsf{cUnit}_{\mathsf{German}}) = \{&\mathsf{cConfiguration1}_{\mathsf{German}},\\ &\mathsf{cConfiguration2}_{\mathsf{German}}\}\end{aligned} \tag{A.240}$$

$$\mathsf{cConfiguration2}_{\mathsf{German}} \geq_{s_{ae}}^{t_n+2} \mathsf{cConfiguration1}_{\mathsf{German}} \tag{A.241}$$

$$\begin{aligned}&\mathsf{Provides}_{s_{ae}}^{t_n+2}(\mathsf{cConfiguration1}_{\mathsf{German}}) = \{\mathsf{cUnitService}_{\mathsf{German}}\}\\&\mathsf{Declares}_{s_{ae}}^{t_n+2}(\mathsf{cConfiguration1}_{\mathsf{German}}) = \emptyset\\&\mathsf{Provides}_{s_{ae}}^{t_n+2}(\mathsf{cConfiguration2}_{\mathsf{German}}) = \{\mathsf{cUnitService}_{\mathsf{German}}\}\\&\mathsf{Declares}_{s_{ae}}^{t_n+2}(\mathsf{cConfiguration2}_{\mathsf{German}}) = \{\mathsf{pUnitReference}_{\mathsf{German}}\}\end{aligned} \tag{A.242}$$

$$\mathsf{Current}_{s_{ae}}^{t_n+2}(\mathsf{cUnit}_{\mathsf{German}}) = \mathsf{cConfiguration2}_{\mathsf{German}} \tag{A.243}$$

$$\mathsf{Uses}_{s_{ae}}^{t_n+2}(\mathsf{pUnitReference}_{\mathsf{German}}) = \mathsf{pUnitService}_{\mathsf{German}} \tag{A.244}$$

A.11.3 German P-Unit

$$\mathsf{Contains}_{s_{ae}}^{t_n+2}(\mathsf{pUnit}_{\mathsf{German}}) = \{\mathsf{pConfiguration1}_{\mathsf{German}}\} \tag{A.245}$$

$$\begin{aligned}&\mathsf{Provides}_{s_{ae}}^{t_n+2}(\mathsf{pConfiguration1}_{\mathsf{German}}) = \{\mathsf{pUnitService}_{\mathsf{German}}\}\\&\mathsf{Declares}_{s_{ae}}^{t_n+2}(\mathsf{pConfiguration1}_{\mathsf{German}}) = \emptyset\end{aligned} \tag{A.246}$$

$$\mathsf{Current}_{s_{ae}}^{t_n+2}(\mathsf{pUnit}_{\mathsf{German}}) = \mathsf{pConfiguration1}_{\mathsf{German}} \tag{A.247}$$

A.11.4 Second German P-Unit

$$\mathsf{Contains}_{s_{ae}}^{t_n+2}(\mathsf{pUnit}_{\mathsf{German}}2) = \{\mathsf{pConfiguration1}_{\mathsf{German}}2\} \tag{A.248}$$

$$\begin{aligned}&\mathsf{Provides}_{s_{ae}}^{t_n+2}(\mathsf{pConfiguration1}_{\mathsf{German}}2) = \{\mathsf{pUnitService}_{\mathsf{German}}2\}\\&\mathsf{Declares}_{s_{ae}}^{t_n+2}(\mathsf{pConfiguration1}_{\mathsf{German}}2) = \emptyset\end{aligned} \tag{A.249}$$

$$\mathsf{Current}_{s_{ae}}^{t_n+2}(\mathsf{pUnit}_{\mathsf{German}}2) = \mathsf{pConfiguration1}_{\mathsf{German}}2 \tag{A.250}$$

A.11.5 Dutch M-Unit

$$\begin{aligned}\mathsf{Contains}_{s_{ae}}^{t_n+2}(\mathsf{mUnit}_{\mathsf{Dutch}}) = \{&\mathsf{mConfiguration1}_{\mathsf{Dutch}},\\&\mathsf{mConfiguration2}_{\mathsf{Dutch}}\}\end{aligned} \tag{A.251}$$

$$\mathsf{mConfiguration2}_{\mathsf{Dutch}} \geq_{s_{ae}}^{t_n+2} \mathsf{mConfiguration1}_{\mathsf{Dutch}} \tag{A.252}$$

$$\begin{aligned}
&\mathsf{Provides}_{s_{ae}}^{t_n+2}(\mathsf{mConfiguration1}_{\mathsf{Dutch}}) = \{\mathsf{mUnitService}_{\mathsf{Dutch}}\} \\
&\mathsf{Declares}_{s_{ae}}^{t_n+2}(\mathsf{mConfiguration1}_{\mathsf{Dutch}}) = \emptyset \\
&\mathsf{Provides}_{s_{ae}}^{t_n+2}(\mathsf{mConfiguration2}_{\mathsf{Dutch}}) = \{\mathsf{mUnitService}_{\mathsf{Dutch}}\} \\
&\mathsf{Declares}_{s_{ae}}^{t_n+2}(\mathsf{mConfiguration2}_{\mathsf{Dutch}}) = \{\mathsf{cUnitReference}_{\mathsf{Dutch}}\}
\end{aligned} \tag{A.253}$$

$$\mathsf{Current}_{s_{ae}}^{t_n+2}(\mathsf{mUnit}_{\mathsf{Dutch}}) = \mathsf{mConfiguration2}_{\mathsf{Dutch}} \tag{A.254}$$

$$\mathsf{Uses}_{s_{ae}}^{t_n+2}(\mathsf{cUnitReference}_{\mathsf{Dutch}}) = \mathsf{cUnitService}_{\mathsf{Dutch}}2 \tag{A.255}$$

A.11.6 Dutch C-Unit

$$\mathsf{Contains}_{s_{ae}}^{t_n+2}(\mathsf{cUnit}_{\mathsf{Dutch}}) = \{\mathsf{cConfiguration1}_{\mathsf{Dutch}}, \mathsf{cConfiguration2}_{\mathsf{Dutch}}\} \tag{A.256}$$

$$\mathsf{cConfiguration2}_{\mathsf{Dutch}} \geq_{s_{ae}}^{t_n+2} \mathsf{cConfiguration1}_{\mathsf{Dutch}} \tag{A.257}$$

$$\begin{aligned}
&\mathsf{Provides}_{s_{ae}}^{t_n+2}(\mathsf{cConfiguration1}_{\mathsf{Dutch}}) = \{\mathsf{cUnitService}_{\mathsf{Dutch}}\} \\
&\mathsf{Declares}_{s_{ae}}^{t_n+2}(\mathsf{cConfiguration1}_{\mathsf{Dutch}}) = \emptyset \\
&\mathsf{Provides}_{s_{ae}}^{t_n+2}(\mathsf{cConfiguration2}_{\mathsf{Dutch}}) = \{\mathsf{cUnitService}_{\mathsf{Dutch}}\} \\
&\mathsf{Declares}_{s_{ae}}^{t_n+2}(\mathsf{cConfiguration2}_{\mathsf{Dutch}}) = \{\mathsf{pUnitReference}_{\mathsf{Dutch}}\}
\end{aligned} \tag{A.258}$$

$$\mathsf{Current}_{s_{ae}}^{t_n+2}(\mathsf{cUnit}_{\mathsf{Dutch}}) = \mathsf{cConfiguration2}_{\mathsf{Dutch}} \tag{A.259}$$

$$\mathsf{Uses}_{s_{ae}}^{t_n+2}(\mathsf{pUnitReference}_{\mathsf{Dutch}}) = \mathsf{pUnitService}_{\mathsf{Dutch}} \tag{A.260}$$

A.11.7 Second Dutch C-Unit

$$\begin{aligned}
\mathsf{Contains}_{s_{ae}}^{t_n+2}(\mathsf{cUnit}_{\mathsf{Dutch}}2) = \{&\mathsf{cConfiguration1}_{\mathsf{Dutch}}2, \\
&\mathsf{cConfiguration2}_{\mathsf{Dutch}}2\}
\end{aligned} \tag{A.261}$$

$$\mathsf{cConfiguration2}_{\mathsf{Dutch}}2 \geq_{s_{ae}}^{t_n+2} \mathsf{cConfiguration1}_{\mathsf{Dutch}}2 \tag{A.262}$$

$$\begin{aligned}
&\mathsf{Provides}_{s_{ae}}^{t_n+2}(\mathsf{cConfiguration1}_{\mathsf{Dutch}}2) = \{\mathsf{cUnitService}_{\mathsf{Dutch}}2\} \\
&\mathsf{Declares}_{s_{ae}}^{t_n+2}(\mathsf{cConfiguration1}_{\mathsf{Dutch}}2) = \emptyset \\
&\mathsf{Provides}_{s_{ae}}^{t_n+2}(\mathsf{cConfiguration2}_{\mathsf{Dutch}}2) = \{\mathsf{cUnitService}_{\mathsf{Dutch}}2\} \\
&\mathsf{Declares}_{s_{ae}}^{t_n+2}(\mathsf{cConfiguration2}_{\mathsf{Dutch}}2) = \{\mathsf{pUnitReference}_{\mathsf{Dutch}}2\}
\end{aligned} \tag{A.263}$$

$$\mathsf{Current}_{s_{ae}}^{t_n+2}(\mathsf{cUnit}_{\mathsf{Dutch}}2) = \mathsf{cConfiguration1}_{\mathsf{Dutch}}2 \tag{A.264}$$

$$\mathsf{Uses}_{s_{ae}}^{t_n+2}(\mathsf{pUnitReference}_{\mathsf{Dutch}}2) = \emptyset \tag{A.265}$$

A.11.8 Dutch P-Unit

$$\text{Contains}_{s_{ae}}^{t_n+2}(\text{pUnit}_{\text{Dutch}}) = \{\text{pConfiguration1}_{\text{Dutch}}\} \tag{A.266}$$

$$\begin{aligned}\text{Provides}_{s_{ae}}^{t_n+2}(\text{pConfiguration1}_{\text{Dutch}}) &= \{\text{pUnitService}_{\text{Dutch}}\}\\ \text{Declares}_{s_{ae}}^{t_n+2}(\text{pConfiguration1}_{\text{Dutch}}) &= \emptyset\end{aligned} \tag{A.267}$$

$$\text{Current}_{s_{ae}}^{t_n+2}(\text{pUnit}_{\text{Dutch}}) = \text{pConfiguration1}_{\text{Dutch}} \tag{A.268}$$

A.11.9 Semantical Compatibility

$$\begin{aligned}&\text{ServiceBehavior}_{s_{ae}}^{t_n+2}(\text{pUnitService}_{\text{German}}) =\\ &\text{pUnitServiceUsualOperation}_{\text{German}}\end{aligned} \tag{A.269}$$

$$\begin{aligned}&\text{ServiceBehavior}_{s_{ae}}^{t_n+2}(\text{pUnitService}_{\text{Dutch}}) =\\ &\text{pUnitServiceUsualOperation}_{\text{Dutch}}\end{aligned} \tag{A.270}$$

$$\begin{aligned}&\text{ServiceBehavior}_{s_{ae}}^{t_n+2}(\text{pUnitService}_{\text{German}}2) =\\ &\text{pUnitServiceUsualOperation}_{\text{German}}\end{aligned} \tag{A.271}$$

$$\begin{aligned}&\text{ServiceReferenceBehavior}_{s_{ae}}^{t_n+2}(\text{pUnitReference}_{\text{German}},\\ &\text{pUnitService}_{\text{German}}) = \text{pUnitReferenceUsualOperation}_{\text{German}}\end{aligned} \tag{A.272}$$

$$\begin{aligned}&\text{ServiceReferenceBehavior}_{s_{ae}}^{t_n+2}(\text{pUnitReference}_{\text{German}}, \text{pUnitService}_{\text{Dutch}}) =\\ &\text{pUnitReferenceUsualOperation}_{\text{German}}\end{aligned} \tag{A.273}$$

$$\begin{aligned}&\text{ServiceReferenceBehavior}_{s_{ae}}^{t_n+2}(\text{pUnitReference}_{\text{German}},\\ &\text{pUnitService}_{\text{German}}2) = \text{pUnitReferenceInconsistentMeasurement}_{\text{German}}\end{aligned} \tag{A.274}$$

$$\begin{aligned}&\text{ServiceReferenceBehavior}_{s_{ae}}^{t_n+2}(\text{pUnitReference}_{\text{Dutch}}, \text{pUnitService}_{\text{German}}) =\\ &\text{pUnitReferenceUsualOperation}_{\text{Dutch}}\end{aligned} \tag{A.275}$$

$$\mathsf{ServiceReferenceBehavior}_{s_{ae}}^{t_n+2}(\mathsf{pUnitReference}_{\mathsf{Dutch}}, \mathsf{pUnitService}_{\mathsf{Dutch}}) = \mathsf{pUnitReferenceUsualOperation}_{\mathsf{Dutch}} \tag{A.276}$$

$$\mathsf{ServiceReferenceBehavior}_{s_{ae}}^{t_n+2}(\mathsf{pUnitReference}_{\mathsf{Dutch}}, \mathsf{pUnitService}_{\mathsf{German}}2) = \mathsf{pUnitReferenceDeadOperation}_{\mathsf{Dutch}} \tag{A.277}$$

$$\mathsf{ServiceReferenceBehavior}_{s_{ae}}^{t_n+2}(\mathsf{pUnitReference}_{\mathsf{Dutch}}2, \mathsf{pUnitService}_{\mathsf{German}}) = \mathsf{pUnitReferenceUsualOperation}_{\mathsf{Dutch}} \tag{A.278}$$

$$\mathsf{ServiceReferenceBehavior}_{s_{ae}}^{t_n+2}(\mathsf{pUnitReference}_{\mathsf{Dutch}}2, \mathsf{pUnitService}_{\mathsf{Dutch}}) = \mathsf{pUnitReferenceUsualOperation}_{\mathsf{Dutch}} \tag{A.279}$$

$$\mathsf{ServiceReferenceBehavior}_{s_{ae}}^{t_n+2}(\mathsf{pUnitReference}_{\mathsf{Dutch}}2, \mathsf{pUnitService}_{\mathsf{German}}2) = \mathsf{pUnitReferenceDeadOperation}_{\mathsf{Dutch}} \tag{A.280}$$

$$\mathsf{pUnitReference}_{\mathsf{German}} \simeq_{\mathsf{Semantical}_{s_{ae}}^{t_n+2}} \mathsf{pUnitService}_{\mathsf{German}} \tag{A.281}$$

$$\mathsf{pUnitReference}_{\mathsf{German}} \simeq_{\mathsf{Semantical}_{s_{ae}}^{t_n+2}} \mathsf{pUnitService}_{\mathsf{Dutch}} \tag{A.282}$$

$$\mathsf{pUnitReference}_{\mathsf{German}} \simeq_{\mathsf{Semantical}_{s_{ae}}^{t_n+2}} \mathsf{pUnitService}_{\mathsf{German}}2 \tag{A.283}$$

$$\mathsf{pUnitReference}_{\mathsf{Dutch}} \simeq_{\mathsf{Semantical}_{s_{ae}}^{t_n+2}} \mathsf{pUnitService}_{\mathsf{German}} \tag{A.284}$$

$$\mathsf{pUnitReference}_{\mathsf{Dutch}} \simeq_{\mathsf{Semantical}_{s_{ae}}^{t_n+2}} \mathsf{pUnitService}_{\mathsf{Dutch}} \tag{A.285}$$

$$\mathsf{pUnitReference}_{\mathsf{Dutch}} \not\simeq_{\mathsf{Semantical}_{s_{ae}}^{t_n+2}} \mathsf{pUnitService}_{\mathsf{German}}2 \tag{A.286}$$

$$\mathsf{pUnitReference}_{\mathsf{Dutch}}2 \simeq_{\mathsf{Semantical}_{s_{ae}}^{t_n+2}} \mathsf{pUnitService}_{\mathsf{German}} \tag{A.287}$$

$$\mathsf{pUnitReference}_{\mathsf{Dutch}}2 \simeq_{\mathsf{Semantical}_{s_{ae}}^{t_n+2}} \mathsf{pUnitService}_{\mathsf{Dutch}} \tag{A.288}$$

$$\mathsf{pUnitReference}_{\mathsf{Dutch}}2 \not\simeq_{\mathsf{Semantical}_{s_{ae}}^{t_n+2}} \mathsf{pUnitService}_{\mathsf{German}}2 \tag{A.289}$$

A.12 Mapping Instances To Types

$$\mathsf{Implements}(\mathsf{mUnitService}_{\mathsf{German}}) = \mathsf{mUnitServiceIf} \tag{A.290}$$

$$\mathsf{Implements}(\mathsf{mUnitService}_{\mathsf{Dutch}}) = \mathsf{mUnitServiceIf} \tag{A.291}$$

$$\mathsf{RefersTo}(\mathsf{cUnitReference}_{\mathsf{German}}) = \mathsf{cUnitServiceIf} \tag{A.292}$$

$$\mathsf{RefersTo}(\mathsf{cUnitReference}_{\mathsf{Dutch}}) = \mathsf{cUnitServiceIf} \tag{A.293}$$

$$\mathsf{Implements}(\mathsf{cUnitService}_{\mathsf{German}}) = \mathsf{cUnitServiceIf} \tag{A.294}$$

$$\mathsf{Implements}(\mathsf{cUnitService}_{\mathsf{Dutch}}) = \mathsf{cUnitServiceIf} \tag{A.295}$$

$$\mathsf{Implements}(\mathsf{cUnitService}_{\mathsf{Dutch}}2) = \mathsf{cUnitServiceIf} \tag{A.296}$$

$$\mathsf{RefersTo}(\mathsf{pUnitReference}_{\mathsf{German}}) = \mathsf{pUnitServiceIf} \tag{A.297}$$

$$\mathsf{RefersTo}(\mathsf{pUnitReference}_{\mathsf{Dutch}}) = \mathsf{pUnitServiceIf} \tag{A.298}$$

$$\mathsf{RefersTo}(\mathsf{pUnitReference}_{\mathsf{Dutch}}2) = \mathsf{pUnitServiceIf} \tag{A.299}$$

$$\mathsf{Implements}(\mathsf{pUnitService}_{\mathsf{German}}) = \mathsf{pUnitServiceIf} \tag{A.300}$$

$$\mathsf{Implements}(\mathsf{pUnitService}_{\mathsf{German}}2) = \mathsf{pUnitServiceIf} \tag{A.301}$$

$$\mathsf{Implements}(\mathsf{pUnitService}_{\mathsf{Dutch}}) = \mathsf{pUnitServiceIf} \tag{A.302}$$

A.13 Syntactical Compatibility

$$\begin{aligned} \mathsf{cUnitReference}_{\mathsf{German}} &\simeq_{\mathsf{Syntactical}} \mathsf{cUnitService}_{\mathsf{German}} \\ \mathsf{cUnitReference}_{\mathsf{German}} &\simeq_{\mathsf{Syntactical}} \mathsf{cUnitService}_{\mathsf{Dutch}} \\ \mathsf{cUnitReference}_{\mathsf{German}} &\simeq_{\mathsf{Syntactical}} \mathsf{cUnitService}_{\mathsf{Dutch}}2 \end{aligned} \tag{A.303}$$

$$\begin{aligned} \mathsf{pUnitReference}_{\mathsf{German}} &\simeq_{\mathsf{Syntactical}} \mathsf{pUnitService}_{\mathsf{German}} \\ \mathsf{pUnitReference}_{\mathsf{German}} &\simeq_{\mathsf{Syntactical}} \mathsf{pUnitService}_{\mathsf{Dutch}} \\ \mathsf{pUnitReference}_{\mathsf{German}} &\simeq_{\mathsf{Syntactical}} \mathsf{pUnitService}_{\mathsf{German}}2 \end{aligned} \tag{A.304}$$

$$\begin{aligned} \mathsf{cUnitReference}_{\mathsf{Dutch}} &\simeq_{\mathsf{Syntactical}} \mathsf{cUnitService}_{\mathsf{German}} \\ \mathsf{cUnitReference}_{\mathsf{Dutch}} &\simeq_{\mathsf{Syntactical}} \mathsf{cUnitService}_{\mathsf{Dutch}} \\ \mathsf{cUnitReference}_{\mathsf{Dutch}} &\simeq_{\mathsf{Syntactical}} \mathsf{cUnitService}_{\mathsf{Dutch}}2 \end{aligned} \tag{A.305}$$

$$\begin{aligned} \mathsf{pUnitReference}_{\mathsf{Dutch}} &\simeq_{\mathsf{Syntactical}} \mathsf{pUnitService}_{\mathsf{German}} \\ \mathsf{pUnitReference}_{\mathsf{Dutch}} &\simeq_{\mathsf{Syntactical}} \mathsf{pUnitService}_{\mathsf{Dutch}} \\ \mathsf{pUnitReference}_{\mathsf{Dutch}} &\simeq_{\mathsf{Syntactical}} \mathsf{pUnitService}_{\mathsf{German}}2 \end{aligned} \tag{A.306}$$

$$\begin{aligned} \mathsf{pUnitReference}_{\mathsf{Dutch}}2 &\simeq_{\mathsf{Syntactical}} \mathsf{pUnitService}_{\mathsf{German}} \\ \mathsf{pUnitReference}_{\mathsf{Dutch}}2 &\simeq_{\mathsf{Syntactical}} \mathsf{pUnitService}_{\mathsf{Dutch}} \\ \mathsf{pUnitReference}_{\mathsf{Dutch}}2 &\simeq_{\mathsf{Syntactical}} \mathsf{pUnitService}_{\mathsf{German}}2 \end{aligned} \tag{A.307}$$

B

Implementation of our Application Example

In this chapter of the Appendix, we will have a look at some of the Dependable Dynamic Adaptive Components from our application example to illustrate, how component vendors can implement them using our framework. Then we will have a look, how node specifications look like, enabling users to start up these Dependable Dynamic Adaptive Components.

B.1 Dependable Dynamic Adaptive Components from our example

In our application example we identified several Dependable Dynamic Adaptive Components like IC-Units, M-Units, C-Units, or P-Units establishing our overall Dependable Dynamic Adaptive System supporting medics during the triage classification process in major incidents. Each component may be implemented by a different vendor and there might be different Dependable Dynamic Adaptive Components implementing syntactically compatible Service Interfaces.

First of all, the Domain Architecture describing Service Interfaces for Dependable Dynamic Adaptive Components of emergency assistance systems needs to be implemented. Using our reference implementation DAiSI Service Interfaces need to be specified in IDL, since DAiSI is based on CORBA.

Let us here have a look at Service Interface `PUnitServiceIf`, which is pro-

vided by P-Units and required by C-Units in our application example. This Service Interface has already been defined in Figure 3.10 by inheriting from Service Interfaces `PulseRateSensorIf` and `BloodPressureSensorIf`. This is depicted in Figure B.1 as a reminder.

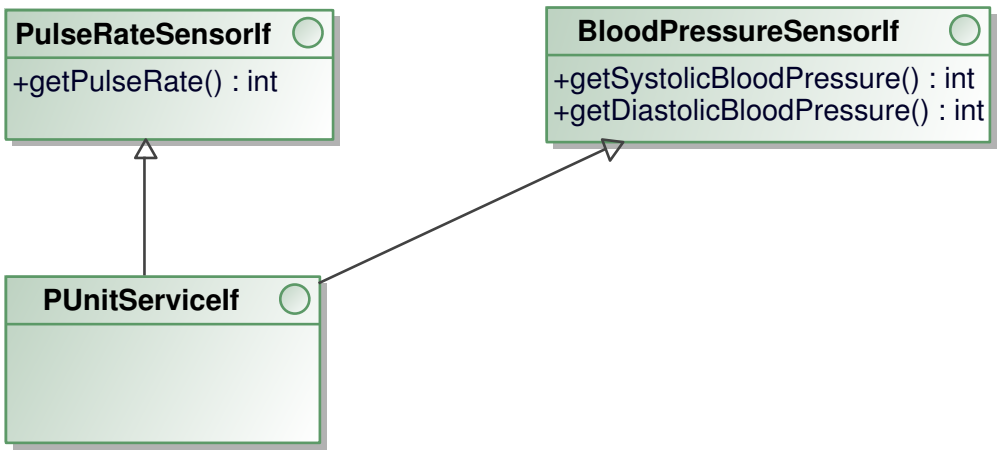


Figure B.1: Specification of PUnitServiceIf from our Domain Architecture.

Definition of a Service Interface in IDL.

The IDL file depicted in Listing B.1 specifies these three Service Interfaces.

```
1  #ifndef _PUNITSERVICE_IF
2    #define _PUNITSERVICE_IF
3    #ifndef _INSTANCECOMPONENT_IDL
4      #include <de/tuc/ifi/sse/daisi/componentModel/
           instanceComponent/instanceComponent.idl>
5    #endif
6    module de{ module tuc{ module ifi{ module sse{ module daisi{
         module applicationExample{ module domainArchitecture{
7      interface PulseRateSensorIf: ::de::tuc::ifi::sse::daisi::
           componentModel::instanceComponent::
           InstanceComponentServiceIf {
8       long getPulseRate();
9      };
10     interface BloodPressureSensorIf: ::de::tuc::ifi::sse::daisi
           ::componentModel::instanceComponent::
           InstanceComponentServiceIf{
11      long getSystolicBloodPressure();
12      long getDiastolicBloodPressure();
13     };
14     interface PUnitServiceIf: BloodPressureSensorIf,
           PulseRateSensorIf {
15     };
16 }; }; }; }; }; }; };
```

```
17 #endif
```

Listing B.1: Specification of Service Interface PUnitServiceIf.

The preamble of this IDL file in lines 1 – 5 is necessary for two reasons: on the one hand, it resolves cyclic dependencies during parsing by an IDL compiler in lines 1 and 2. On the other hand, it imports the IDL specification of Service Interfaces in general (lines 3 – 5), which is defined by our reference implementation DAiSI.

This standard preamble is necessary for each IDL specification of a Service Interface when using DAiSI. You only need to adapt the defined name (*_PUNITSERVICE_IF*) in lines 1 and 2 to the name of the Service Interface to be specified in this IDL file.

In line 6, the package of the following Service Interfaces is defined as hierarchical modules which will be translated into the Java package *de.tuc.ifi.sse.daisi.applicationExample.domainArchitecture* during compilation of the IDL file.

Lines 7 – 9 define the Service Interface `PulseRateSensorIf` containing a single method `getPulseRate()` returning the pulse rate of a casualty. As specified in line 7, this Service Interface extends *InstanceComponentServiceIf* from package *de.tuc.ifi.sse.daisi.componentModel.instanceComponent*. This marks it as a Service Interface for our reference implementation DAiSI. The same is specified for Service Interface `BloodPressureSensorIf` in line 10.

For Service Interface `PUnitServiceIf` InstanceComponentServiceIf does not need to be extended, as the two previously marked Service Interfaces `PulseRateSensorIf` and `BloodPressureSensorIf` are extended.

Within this section we will focus on three different Dependable Dynamic Adaptive Components of our application example to depict the benefits of our approach and to provide a guideline for developers, how to develop Dependable Dynamic Adaptive Components using our approach.

1. A German P-Unit measuring the pulse rate using a fingerclip and measuring the blood pressure using a wrist cuff.

2. A German C-Unit assuming, that pulse rate and blood pressure are measured independent of each other.

3. A Dutch C-Unit assuming, that pulse rate and are measured by the same physical sensor.

In the following we will describe the implementation of these three Dependable Dynamic Adaptive Components starting with the German P-Unit. The Listings in the following are not complete but focus on the interesting parts of these implementations. If you are interested in the complete Listings: this thesis is accompanied by a CD containing the complete sources.

B.1.1 German P-Unit

The German P-Unit in our application example has been structured as depicted in Figure B.2. It only has a single Dependable Component Configuration. In this configuration it provides Service Interface `PUnitServiceIf` from our Domain Architecture. Thus, this Dependable Dynamic Adaptive Component serves as an example, how a component vendor can implement a component, providing a Dependable Service.

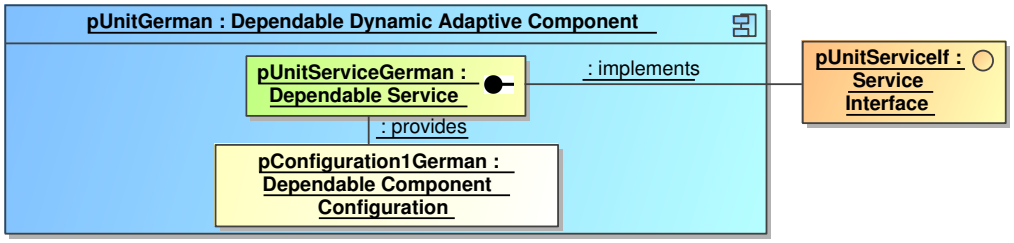


Figure B.2: Structure of the German P-Unit from our Application Example.

Let us first of all have a look at the declaration part of the German P-Unit, which is depicted in Listing B.2. The German P-Unit extends `AbstractInstanceComponentImpl` of our framework as described in section 6.2. Next to this, it implements two interfaces: Service Interface `PUnitServiceIf`[1] and the `Observer` interface.

The Observer interface is implemented, as this German P-Unit has an internal object for its physical sensor (c.f. line 3 of Listing B.2), which is notifies the P-Unit by calling its `update(Observable o, Object arg)` method, whenever the sensor measures changed sensor values. By this, the German P-Unit can notify the Dependable Configuration Component in case of an internal state change. Thus, changing Behavior Equivalence Classes and associated BehavioralChange triggers are considered for reconfiguration by the Dependable Configuration Component.

Creating a Service. Within this listing fragment, the Dependable Component Configuration is created and structured as depicted in Figure B.2: First an attribute for this Dependable Component Configuration is declared in line 2. This attribute is initialized with an empty Dependable Component Configuration in Line 7 by calling the `createConfiguration()` method, which is implemented in its superclass `AbstractInstanceComponentImpl` of our framework. Then a remote object is created in line 9 by calling the `createService` method at the superclass.

[1] The suffix *Operations* results from the CORBA Tie approach used in our reference implementation DAiSI. The Tie approach avoids multiple inheritance by applying the Delegation pattern.

This method takes two parameters: first a class specifying the Service Interface[2] and second the object implementing the Service Interface[3].

The created remote object representing a Dependable Service is added to the Dependable Component Configuration in line 11 by calling `addProvides` at the Dependable Component Configuration and passing the Dependable Service as a parameter. In line 18 we add the Dependable Component Configuration to the component by calling `addContains`. In consequence we have created a structure as it had been depicted in Figure B.2.

```
1 public class PUnitImpl extends AbstractInstanceComponentImpl
     implements PUnitServiceIfOperations, Observer{
2   private InstanceComponentConfigurationIf
       pConfiguration1German;
3   private PhysicalSensorImpl physicalSensor;
4
5   public PUnitImpl() {
6     super();
7     pConfiguration1German = createConfiguration();
8     // Create and add Provided Services here
9     PUnitServiceIf pUnitServiceGerman = (PUnitServiceIf)
         createService(
10       PUnitServiceIf.class, this);
11    pConfiguration1German.addProvides(pUnitServiceGerman);
12
13    /** The Configuration is added to the Component.
14     * Note, that the order of adding the configurations
          decides, which
15     * Configuration is treated as "better" by DAiSI.
16     * Configurations added earlier are treated as better.
17     */
18    addContains(pConfiguration1German);
19  }
```

Listing B.2: Declaration Part of the P-Unit Implementation by the German Vendor.

In the following, we have a look at methods, which are called by DAiSI during the component's lifecycle. The `initialize()` method is called by the Node Component before it registers this Dependable Dynamic Adaptive Component at DAiSI's Dependable Configuration Component. Thus, here initialization operations may be performed analogously to an object's constructor. In our example here, we create an object representing the physical sensor measuring the vital data of a casualty in line 22.

`initialize()` is called before registration.

[2]This is required, as a Dependable Dynamic Adaptive Component could implement multiple Service Interfaces and, therefore, we need to know, for which implemented Service Interface a Dependable Service needs to be created.

[3]This enables component vendors to implement Service Interfaces using separate objects like inner classes.

runCurrent() is called by the Dependable Configuration Component to activate a Dependable Dynamic Adaptive Component.

Method `runCurrent()` is called by DAiSI's Dependable Configuration Component when the Dependable Dynamic Adaptive Component has been bound to other components of the Dependable Dynamic Adaptive System and should start. Thus, the Dependable Configuration Component has set a Dependable Component Configuration of this component as Current Configuration and it has set all Dependable Service References of this configuration to syntactically and semantically compatible Dependable Services provided by other components.

Usually we need to check, which is the Current Configuration, when `runCurrent()` is called in order to behave accordingly. Since the German P-Unit only has a single Dependable Component Configuration, we leave out this part here and directly call `setVisible(true)` on the physical sensor. This causes, that a simulated vital data sensor pops up as depicted in Figure B.3, providing sliders for the sensor values[4].

Method `stopCurrent()` is called by DAiSI's Dependable Configuration Component, if the Dependable Dynamic Adaptive Component is unbound from other components and should not be executed anymore. Thus, we need to ensure, that we do not access Dependable Service References anymore, when this method is called. Since the German P-Unit does not declare any Dependable Service References, we only use it here to hide the GUI of the physical sensor by calling `setVisible(false)`.

```
20  @Override
21  public void initialize() {
22    physicalSensor = new PhysicalSensorImpl(this);
23  }
24
25  @Override
26  public void runCurrent() {
27    physicalSensor.setVisible(true);
28  }
29
30  @Override
31  public void stopCurrent() {
32    physicalSensor.setVisible(false);
33
34  }
```

Listing B.3: DAiSI Specific Part of the P-Unit Implementation by the German Vendor.

Implementation of a Service Interface.

In the following we show the implementation of the specific methods of the provided Service Interface `PUnitServiceIf`. These are depicted in Listing B.4 and are realized simply by delegating the call to the physical sensor. Note, that

[4]There also is a *real* implementation of this physical sensor. However, we use a simulated version here, in order to enable each reader of this thesis to execute our application example – contained on the enclosed DVD – without owning such an expensive sensor.

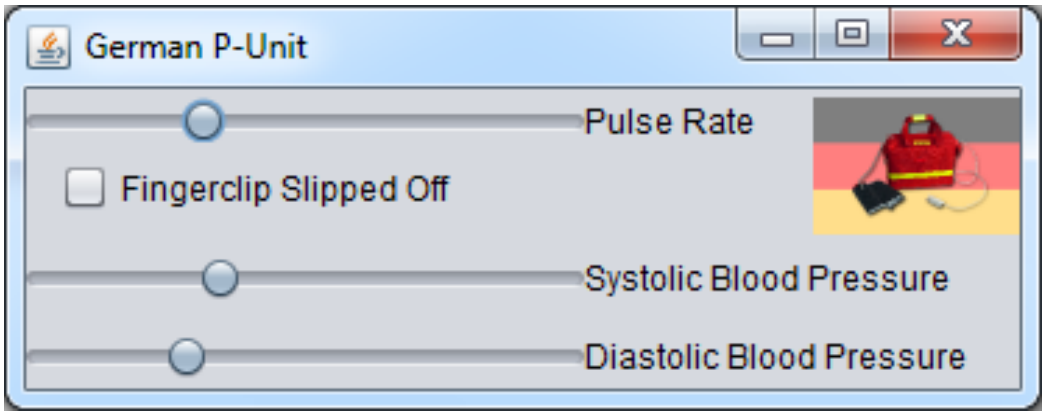


Figure B.3: GUI of the German P-Unit from our Application Example Featuring a Simulated Sensor.

usually we need to query the Current Configuration here as well, as we may realize methods of a Service Interface differently according to the Dependable Component Configuration. However, we leave it out here again, as we only have a single configuration.

```
35  public int getDiastolicBloodPressure() {
36    return physicalSensor.getDiastolicBloodPressure();
37  }
38
39  @Override
40  public int getSystolicBloodPressure() {
41    return physicalSensor.getSystolicBloodPressure();
42  }
43
44  @Override
45  public int getPulseRate() {
46    return physicalSensor.getPulseRate();
47  }
```

Listing B.4: Service Implementation Part of the P-Unit Implementation by the German Vendor.

Next to the service-specific methods, the German P-Unit implements the Observer interface and, therefore, has a method `update(Observable o, Object arg)`, which is called by the physical sensor, whenever the measured sensor values change. This means, that the internal state of this component has changed and, therefore, the Dependable Configuration Component needs to be notified. The realization of the `update` method is depicted in Listing B.5.

In line 51 we call method `stateChanged()` realized by the superclass. This method causes a notification of DAiSI's Dependable Configuration Component,

A call of stateChanged causes a reevaluation of semantical Compatibility of all bindings involving this Component.

which checks, whether a Behavior Equivalence Class of its provided Dependable Services respectively a Behavior Equivalence Class of Dependable Service References syntactically compatible with these Dependable Service, or a Behavior Equivalence Class of its Dependable Service References to other Dependable Services has changed. If it recognizes a change, it executes the specified tests of these Dependable Service References to update semantical Compatibility. In addition it updates the System Configuration, if it detects an active BehavioralChange trigger, which means a change in semantical Compatibility.

```
49  public void update(Observable o, Object arg) {
50    System.out.println("Notified");
51    stateChanged();
52  }
```

Listing B.5: Observer Implementation Part of the P-Unit Implementation by the German Vendor.

According to our approach, a Dependable Dynamic Adaptive Component needs to give information about the Behavior Equivalence Class, a Dependable Service is currently in. For our German P-Unit the vendor can think of several different Behavior Equivalence Classes numbered from -3 to 1, where the negative numbers indicate potential malfunctions of the sensor[5].

Behavior Equivalence Classes in our application example.

To improve readability of the source code, the German vendor mapped them to constants `BELOW_ZERO = -3`, `SINGLE_VALUE_ZERO = -2`, `SYSTOLIC_BELOW_DIASTOLIC = -1`, `ALL_ZERO = 0`, and `ABOVE_ZERO = 1`.

- BELOW_ZERO denotes the situation, that one of the sensor values is below zero, which indicates, that there is a malfunction of one of the sensors.

- SINGLE_VALUE_ZERO describes, that one sensor value is zero, while at least one other sensor value is above zero. This may be due to a slipped of fingerclip or wrist cuff or due to a sensor malfunction.

- SYSTOLIC_BELOW_DIASTOLIC characterizes situations, where the systolic blood pressure is below the diastolic blood pressure, which cannot occur in reality and, therefore, indicates, that something is wrong with the physical sensor.

[5]Note that this has been specified by the German vendor. He could also have specified, that the sensor behaves equivalently for each state and, therefore, always return the same Behavior Equivalence Class. However, this would increase the risk, that incompatibilities are missed during runtime, as only changing Behavior Equivalence Classes of Dependable Service References would remain as triggers for test case execution in this case.

- ALL_ZERO denotes a situation where all sensor values are equal to zero. This may be due to a cardiac arrest of the casualty or due to the fact, that the sensor is not connected to a casualty.

- ABOVE_ZERO denotes all other situations, where plausible values for blood pressure and pulse rate are measured.

The method `getServiceBehavior()` returns an integer value representing the currently active Behavior Equivalence Class. The realization of this method by the German P-Unit is depicted in Listing B.6

```
53  public int getServiceBehavior(){
54    if (physicalSensor.getPulseRate()<0||physicalSensor.
         getDiastolicBloodPressure()<0||physicalSensor.
         getSystolicBloodPressure()<0){
55      return BELOW_ZERO; // One Sensor value is below zero.
56    }
57    if ((physicalSensor.getPulseRate()==0&&(physicalSensor.
         getDiastolicBloodPressure()!=0||physicalSensor.
         getSystolicBloodPressure()!=0))||physicalSensor.
         getPulseRate()!=0&&((physicalSensor.
         getDiastolicBloodPressure()==0||physicalSensor.
         getSystolicBloodPressure()==0))){
58      return SINGLE_VALUE_ZERO; // One Sensor value is zero,
           while one of the other ones is above zero.
59    }
60    if (physicalSensor.getSystolicBloodPressure()<
         physicalSensor.getDiastolicBloodPressure()){
61      return SYSTOLIC_BELOW_DIASTOLIC; // The Systolic Blood
           Pressure is lower than the Diastolic Blood Pressure.
62    }
63    if (physicalSensor.getPulseRate()==0||physicalSensor.
         getDiastolicBloodPressure()==0||physicalSensor.
         getSystolicBloodPressure()==0){
64      return ALL_ZERO; // all values are zero -> Casualty dead?
65    }
66    return ABOVE_ZERO; // Values above zero have been measured.
67  }
68
69 }
```

Listing B.6: Provided Service Behavior Calculation of the P-Unit Implementation by the German Vendor.

In the following we will have a look at the C-Unit implementation provided by the German vendor.

B.1.2 German C-Unit

The German C-Unit of our application example has been structured as depicted in Figure B.4. It has two Dependable Component Configurations. In both configurations it provides Service Interface `CUnitServiceIf` from our Domain Architecture.

A single Component, two configurations

The configurations differ in that way, that the C-Unit declares a Dependable Service Reference `pUnitReferenceGerman` in `cConfiguration2German` whereas it declares no Dependable Service Reference in `cConfiguration1German`. Configuration `cConfiguration2German` is the better Dependable Component Configuration and offers automatic Triage Class calculation, whereas the Triage Class of a casualty needs to be set manually in the worse configuration `cConfiguration1German`.

We use this Dependable Dynamic Adaptive Component in the following as an example, how a component vendor can implement a component, requiring a Dependable Service. We will focus on the differences to the German P-Unit and skip identical parts of the implementations, as they already have been explained before.

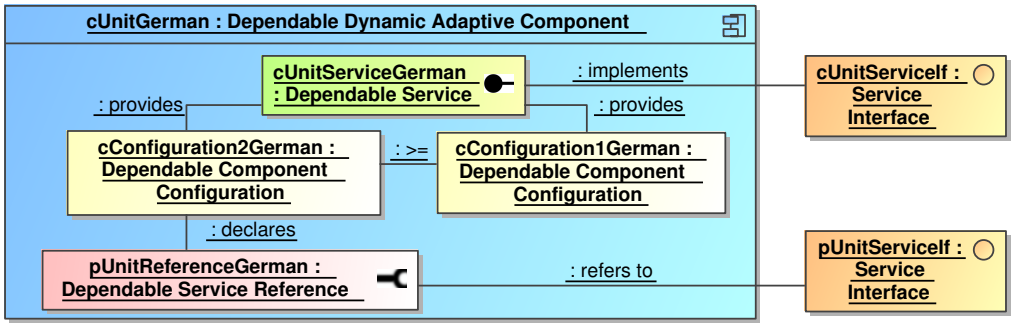


Figure B.4: Structure of the German C-Unit from our Application Example.

Let us first of all have a look at the declaration part of the German C-Unit, which is depicted in Listing B.7. Compared to the declaration part of the German P-Unit we recognize two main differences:

1. The German C-Unit contains *two* Dependable Component Configurations.
2. The German C-Unit contains a Dependable Service Reference.

These differences reflect in the Listing. First of all, two Dependable Component Configurations `cConfiguration1German` and

`cConfiguration2German` are declared in line 1. Then each configuration is structured and added to the component in lines 7 – 32.

Note, that the order of adding configurations to a component decides, which configuration is treated as better in our reference implementation DAiSI. As we add `cConfiguration2German` in line 20 before we add `cConfiguration1German` in line 32, `German` is treated as better, which is reasonable, as it features automatic calculation of Triage Classes.

Order of adding configurations decides about quality, if we do not set the quality of a configuration explicitly.

Regarding Dependable Service Reference `pUnitReferenceGerman`, we need to emphasize two parts in the Listing. At first, a private field `pUnit` is declared in line 2, which has the type of the required Dependable Service. At second, we create a remote object of this field by calling `createReference` and passing the name of this field ("pUnit") in line 13. This remote object representing the Dependable Service Reference is then added to configuration `cConfiguration2German` in line 14.

Creating a Service Reference.

In consequence we have created a structure as it had been depicted in Figure B.4.

```
1  private InstanceComponentConfigurationIf
       cConfiguration1German, cConfiguration2German;
2  private PUnitServiceIf pUnit=null;
3
4  public CUnitImpl() {
5    super();
6
7    cConfiguration2German = createConfiguration();
8    // Create and add Provided Services here
9    CUnitServiceIf cUnitService = (CUnitServiceIf)
       createService(
10     CUnitServiceIf.class, this);
11   cConfiguration2German.addProvides(cUnitService);
12   // Create and add Service References for Required Services
       here
13   InstanceComponentServiceReferenceIf pUnitReferenceGerman =
       createReference("pUnit");
14   cConfiguration2German.addDeclares(pUnitReferenceGerman);
15   /** The Configuration is added to the Component.
16    * Note, that the order of adding the configurations
        decides, which
17    * Configuration is treated as "better" by DAiSI.
18    * Configurations added earlier are treated as better.
19    */
20   addContains(cConfiguration2German);
21
22   cConfiguration1German = createConfiguration();
23   // Create and add Provided Services here
24   cConfiguration1German.addProvides(cUnitService);
```

```
25    // Create and add Service References for Required Services
          here
26
27    /** The Configuration is added to the Component.
28     * Note, that the order of adding the configurations
           decides, which
29     * Configuration is treated as "better" by DAiSI.
30     * Configurations added earlier are treated as better.
31     */
32    addContains(cConfiguration1German);
33  }
```

Listing B.7: Declaration Part of the C-Unit Implementation by the German Vendor.

In the following, we have a look at methods, which are called by DAiSI during the component's lifecycle. The realization of these methods is depicted in Listing B.8. The realization of the `initialize()` method does not differ essentially from the German P-Unit – here a GUI instead of a simulated pulse sensor is created.

Method `runCurrent()` is called by DAiSI's Dependable Configuration Component if the Dependable Dynamic Adaptive Component has been bound to other components of the Dependable Dynamic Adaptive System and should start.

You have to use `_is_equivalent` instead of `equals` for comparisons.

We check, which is the Current Configuration by comparing it to our two Dependable Component Configurations[6] in line 42 and 64. Note, that the comparison is done by calling `_is_equivalent` instead of `equals`.

This is due to the fact, that our reference implementation DAiSI is based on CORBA and, therefore, all remote objects are CORBA objects. As a consequence, two different remote objects can point to the same server object – in this case, `equals` would return false, while `_is_equivalent` will return true, which is our desired behavior in this case. As the configuration has been set remotely by DAiSI's Dependable Configuration Component, the compared remote objects will never be equal, forcing us to use `_is_equivalent` for comparisons.

If `cConfiguration2German` is our Current Configuration, we periodically calculate the Triage Class using a thread. Whenever we recognize, that the calculated Triage Class has changed, we notify DAiSI's Dependable Configuration Component by calling `stateChanged()` in line 50. This method is realized by the superclass `AbstractInstanceComponentImpl`.

The method `stopCurrent()` is called by DAiSI's Dependable Configuration Component, if the Dependable Dynamic Adaptive Component is unbound from other components and should not be executed anymore. Thus, we need to ensure, that we do not access Dependable Service References anymore, when this method is called.

[6] As we only have two configurations, we do not need another if-condition, but can simply put our second behavior in the else part.

The German C-Unit periodically accesses its Dependable Service Reference `pUnitReferenceGerman` using a thread, if `cConfiguration2German` has been the Current Configuration. Thus, we destroy this thread in lines 73 – 76 to avoid further accesses to this Dependable Service Reference if `stopCurrent()` is called.

```
34  @Override
35  public void initialize() {
36    myGui = new GUIImpl();
37  }
38
39  @Override
40  public void runCurrent() {
41    myGui.setVisible(true);
42    if (getCurrent()._is_equivalent(cConfiguration2German)){
43      myGui.setCUnitActive(true);
44      updateThread = new Thread(){
45        public void run(){
46          while(!isInterrupted()){
47            TriageClass currentTriageClass = getTriageClass();
48            if (currentTriageClass!=lastCalculatedTriageClass){
49              lastCalculatedTriageClass = currentTriageClass;
50              stateChanged();
51            }
52            myGui.updateLabelValues(pUnit.getPulseRate(), pUnit.
                  getDiastolicBloodPressure(), pUnit.
                  getSystolicBloodPressure());
53            myGui.updateTriageClass(currentTriageClass);
54            try {
55              Thread.sleep(500);
56            } catch (InterruptedException e) {
57              // TODO Auto-generated catch block
58              e.printStackTrace();
59            }
60          }
61        }
62      };
63      updateThread.start();
64    } else {
65      myGui.setCUnitActive(false);
66      myGui.updateTriageClass(manuallySetTriageClass);
67    }
68
69  }
70
71  @Override
72  public void stopCurrent() {
73    if (updateThread!=null){
74      updateThread.interrupt();
```

```
75      updateThread=null;
76    }
77    myGui.setVisible(false);
78  }
```

Listing B.8: DAiSI Specific Part of the C-Unit Implementation by the German Vendor.

Next to these DAiSI specific methods we also need to implement the methods of Service Interface `CUnitServiceIf`. Remarkable about this implementation depicted in Listing B.9 are two details:

1. The implementation behaves differently depending on the Current Configuration of the C-Unit.

2. The implementation queries pulse rate as well as blood pressure to decide, whether a casualty is dead (`TriageClass.EX`) as described in our application example.

By querying the Current Configuration as depicted in lines 81, 90, and 123 of Listing B.9 the German vendor implements different behaviors depending on the Current Configuration. Thus, the German C-Unit returns a automatically calculated Triage Class if it is in Dependable Component Configuration `cConfiguration2German`, while it returns a manually set one in configuration `cConfiguration1German`.

```
79  @Override
80  public PUnitServiceIf getAssignedPeripheralUnit() {
81    if (getCurrent()._is_equivalent(cConfiguration2German)){
82      return pUnit;
83    } else {
84      return null;
85    }
86  }
87
88  @Override
89  public TriageClass getTriageClass() {
90    if (getCurrent()._is_equivalent(cConfiguration2German)){
91      if (pUnit.getPulseRate()==0 && pUnit.
            getDiastolicBloodPressure() == 0 && pUnit.
            getSystolicBloodPressure() == 0){
92        return TriageClass.EX;
93      }
94      if (pUnit.getPulseRate()==0 && (pUnit.
            getDiastolicBloodPressure() != 0 || pUnit.
            getSystolicBloodPressure() != 0)){
95        return TriageClass.UNKNOWN;
96      }
```

```
97      if (pUnit.getPulseRate()!=0 && (pUnit.
           getDiastolicBloodPressure() == 0 || pUnit.
           getSystolicBloodPressure() == 0)){
98       return TriageClass.UNKNOWN;
99      }
100     if (pUnit.getPulseRate()<0||pUnit.
           getDiastolicBloodPressure() < 0 || pUnit.
           getSystolicBloodPressure() < 0){
101      return TriageClass.UNKNOWN;
102     }
103     if ((T_IV_LOW_BORDER<pUnit.getPulseRate()&&pUnit.
           getPulseRate()<=T_III_LOW_BORDER||(T_III_HIGH_BORDER<
           pUnit.getPulseRate()&&pUnit.getPulseRate()<=
           T_IV_HIGH_BORDER))){
104      return TriageClass.T_IV;
105     }
106     if ((T_III_LOW_BORDER<pUnit.getPulseRate()&&pUnit.
           getPulseRate()<=T_II_LOW_BORDER||(T_II_HIGH_BORDER<
           pUnit.getPulseRate()&&pUnit.getPulseRate()<=
           T_III_HIGH_BORDER))){
107      return TriageClass.T_III;
108     }
109     if ((T_II_LOW_BORDER<pUnit.getPulseRate()&&pUnit.
           getPulseRate()<=T_I_LOW_BORDER||(T_I_HIGH_BORDER<pUnit
           .getPulseRate()&&pUnit.getPulseRate()<=
           T_II_HIGH_BORDER))){
110      return TriageClass.T_II;
111     }
112     if ((T_I_LOW_BORDER<=pUnit.getPulseRate()&&pUnit.
           getPulseRate()<T_I_HIGH_BORDER)){
113      return TriageClass.T_I;
114     }
115     return TriageClass.UNKNOWN;
116    } else {
117     return manuallySetTriageClass;
118    }
119   }
120
121   @Override
122   public boolean setTriageClass(TriageClass newTriageClass) {
123    if (getCurrent()._is_equivalent(cConfiguration1German)){
124     manuallySetTriageClass = newTriageClass;
125     return true;
126    } else {
127     return false;
128    }
129   }
130
131   @Override
```

```
132  public int getLatitude() {
133    return 0;
134  }
135
136  @Override
137  public int getLongitude() {
138    return 0;
139  }
```

Listing B.9: Service Implementation Part of the C-Unit Implementation by the German Vendor.

The remaining part of the implementation is the calculation of Behavior Equivalence Classes regarding the required P-Unit and the test cases, that need to be executed *to decide, whether a given P-Unit is semantically compatible*.

As a Dependable Dynamic Adaptive Component may declare several Dependable Service References, both – the method calculating the Behavior Equivalence Class as well as the method executing test cases – need to contain an identifier specifying the Dependable Service Reference, they refer to. In our reference implementation DAiSI, we indicate this, by adding an underscore followed by the name of the attribute representing this Dependable Service Reference as a suffix to the methods `getServiceReferenceBehavior` and `equivalenceClassTest`.

Behavior Equivalence Classes for Service References.

In our example, the German C-Unit contains an attribute pUnit (cf. line 2 of Listing B.7) representing the Dependable Service Reference. Thus, the two methods calculating the Behavior Equivalence Class and the semantical Compatibility in our example are `getServiceReferenceBehavior_pUnit` and `equivalenceClassTest_pUnit`.

The calculation of the Behavior Equivalence Class in our example is straightforward. The German vendor decides, that the Behavior Equivalence Class of a P-Unit depends on the Triage Class he would calculate by accessing this P-Unit. Thus, the method `getServiceReferenceBehavior_pUnit` returns the Triage Class converted to an integer value.

The German C-Unit has two different Compliance Test Cases for a P-Unit. The first Compliance Test Case in lines 148 – 150 of Listing B.10 simply checks, whether the systolic blood pressure is equal or above the diastolic blood pressure, since otherwise the P-Unit has a malfunction. This Compliance Test Case is executed, whenever the German C-Unit has calculated the Behavior Equivalence Class of a P-Unit to the integer representation of `TriageClass.EX`.

In all other cases, it is checked additionally, whether the sensor values are equal or above zero as depicted in lines 150 – 152 of Listing B.10, since this is a precondition to calculate the Triage Class in a reasonable way. DAiSI's Dependable Configuration Component executes one of these Compliance Test Cases – depending on the Behavior Equivalence Class – if a Dependable Service Binding is established

respectively if a Behavior Equivalence Class combination of a Dependable Service Binding changes.

```
140  public int getServiceReferenceBehavior_pUnit() {
141    // accesses pUnit (encapsulated in the getTriageClass()-
         call) to calculate the current Behavior Equivalence
         Class.
142    TriageClass tc = getTriageClass();
143    return tc.value();
144  }
145
146  public boolean equivalenceClassTest_pUnit(int
       providerEquivalenceClass, int userEquivalenceClass) {
147    // accesses pUnit to check, whether it is semantically
         equivalent
148    if (userEquivalenceClass == TriageClass.UNKNOWN.value()){
149      return (pUnit.getSystolicBloodPressure()>=pUnit.
           getDiastolicBloodPressure());
150    } else {
151      return (pUnit.getPulseRate()>=0&&pUnit.
           getDiastolicBloodPressure()>=0&&pUnit.
           getSystolicBloodPressure()>=0&&(pUnit.
           getSystolicBloodPressure()>=pUnit.
           getDiastolicBloodPressure()));
152    }
153  }
154 }
```

Listing B.10: Provided Service Behavior Calculation of the C-Unit Implementation by the German Vendor.

B.1.3 Dutch C-Unit

The implementation of the C-Unit by the Dutch vendor differs from the German C-Unit in the implementation of its `getTriageClass()` method. It classifies a casualty as dead (`TriageClass.EX`) as soon as the pulse rate is equal to zero, without considering the blood pressure in addition. This is depicted in Lines 4 – 6 of Listing B.11.

The remainder of the implementation of this method is identical to the implementation of the German vendor.

```
1  @Override
2  public TriageClass getTriageClass() {
3    if (getCurrent()._is_equivalent(cConfiguration2Dutch)){
4      if (pUnit.getPulseRate()==0){
5        return TriageClass.EX;
6      }
```

```
7       if (pUnit.getPulseRate()<0||pUnit.
           getDiastolicBloodPressure() < 0 || pUnit.
           getSystolicBloodPressure() < 0){
8        return TriageClass.UNKNOWN;
9       }
10      if ((T_IV_LOW_BORDER<pUnit.getPulseRate()&&pUnit.
           getPulseRate()<=T_III_LOW_BORDER||(T_III_HIGH_BORDER<
           pUnit.getPulseRate()&&pUnit.getPulseRate()<=
           T_IV_HIGH_BORDER))){
11       return TriageClass.T_IV;
12      }
13      if ((T_III_LOW_BORDER<pUnit.getPulseRate()&&pUnit.
           getPulseRate()<=T_II_LOW_BORDER||(T_II_HIGH_BORDER<
           pUnit.getPulseRate()&&pUnit.getPulseRate()<=
           T_III_HIGH_BORDER))){
14       return TriageClass.T_III;
15      }
16      if ((T_II_LOW_BORDER<pUnit.getPulseRate()&&pUnit.
           getPulseRate()<=T_I_LOW_BORDER||(T_I_HIGH_BORDER<pUnit
           .getPulseRate()&&pUnit.getPulseRate()<=
           T_II_HIGH_BORDER))){
17       return TriageClass.T_II;
18      }
19      if ((T_I_LOW_BORDER<=pUnit.getPulseRate()&&pUnit.
           getPulseRate()<T_I_HIGH_BORDER)){
20       return TriageClass.T_I;
21      }
22      return TriageClass.UNKNOWN;
23     } else {
24      return manuallySetTriageClass;
25     }
26   }
```

Listing B.11: Implementation of the getTriageClass() Method by the C-Unit of the Dutch Vendor.

Due to its different implementation of `getTriageClass()`, the Dutch vendor assumes, that a pulse rate of zero implies a blood pressure of zero. This assumption needs to be reflected in the test cases for P-Units specified by the Dutch C-Unit. You can see the specification of these interoperability tests in Listing B.12.

Within this Listing you can see in lines 6 – 8, that this C-Unit expects a P-Unit to return zero for pulse rate as well as for blood pressure, whenever the C-Unit would classify a casualty as dead. Otherwise the test case execution will fail. This causes, that the Dependable Configuration Component automatically removes the Dependable Service Binding between the Dutch C-Unit and the previously bound P-Unit and changes its Dependable Component Configuration towards `configurationManualTriage`.

```
1  public boolean equivalenceClassTest_pUnit(int
       providerEquivalenceClass, int userEquivalenceClass) {
2    // accesses pUnit to check, whether it is semantically
       equivalent
3    if (userEquivalenceClass == TriageClass.UNKNOWN.value()){
4     return (pUnit.getPulseRate()>=0&&pUnit.
         getDiastolicBloodPressure()>=0&&pUnit.
         getSystolicBloodPressure()>=0&&(pUnit.
         getSystolicBloodPressure()>=pUnit.
         getDiastolicBloodPressure()));
5    } else {
6     if (userEquivalenceClass == TriageClass.EX.value()){
7      return (pUnit.getPulseRate()==0 && pUnit.
         getDiastolicBloodPressure() == 0 && pUnit.
         getSystolicBloodPressure() == 0);
8     } else {
9      return (pUnit.getSystolicBloodPressure()>=pUnit.
         getDiastolicBloodPressure());
10    }
11   }
12  }
```

Listing B.12: Implementation of the Interoperability Test for P-Units Included in the C-Unit of the Dutch Vendor.

B.2 Node Models

Our reference implementation DAiSI requires node models, specifying, which Dependable Dynamic Adaptive Components should be started by a specific physical node. This has already been described in section 6.1.1 in detail. Such a node model specifying, that a German C-Unit should be started up, looks as depicted in Listing B.13. In line 6 of this listing we specify the class, that should be instantiated by the physical node.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <daisi:Node xmi:version="2.0" xmlns:xmi="http://www.omg.org/
    XMI" xmlns:daisi="de.tuc.ifi.sse.daisi">
3  <infrastructureComponent name="
     DependableConfigurationComponent"/>
4  <dependableDynamicAdaptiveComponent name="de.tuc.ifi.sse.
     daisi.applicationExample.germanVendor.cUnit.CUnitImpl"
     requestRun="true" />
5 </daisi:Node>
```

Listing B.13: Node Model for a Physical Node that Should Host a German C-Unit.

If we want to specify a node model, that should startup a German P-Unit or a Dutch C-Unit this line needs to be replaced by the corresponding line from Listing B.14 or B.15.

```
1 <dependableDynamicAdaptiveComponent name="de.tuc.ifi.sse.daisi
      .applicationExample.germanVendor.pUnit.PUnitImpl"
      requestRun="false" />
```

Listing B.14: Excerpt From a Node Model for a Physical Node that Should Host a German P-Unit.

```
1 <dependableDynamicAdaptiveComponent name="de.tuc.ifi.sse.daisi
      .applicationExample.dutchVendor.cUnit.CUnitImpl"
      requestRun="true" />
```

Listing B.15: Excerpt From a Node Model for a Physical Node that Should Host a Dutch C-Unit.

In Listing B.14 you can also see, that we specified property *requestRun* to false. This means, that DAiSI's Dependable Configuration Component will not activate this P-Unit unless there is another Dependable Dynamic Adaptive Component requiring a Dependable Service provided by this P-Unit. This makes sense, as the P-Unit on its own is useless until there is another component using its sensor values.

Bibliography

[AE06] E. H. L. Aarts and J. L. Encarnaçao. *True Visions: The Emergence of Ambient Intelligence (Frontiers Collection)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. 7

[AKNR09] André Appel, Holger Klus, Dirk Niebuhr, and Andreas Rausch. Event-Based Realization of Dynamic Adaptive Systems. In *Handbook of Research on Advanced Distributed Event-Based Systems*. Idea Group Inc., 2009. accepted for publication. 127, 130

[All09] The OSGi Alliance. Osgi service platform core specification, release 4, version 4.2. http://www.osgi.org/download/r4v42/r4.core.pdf [Online; accessed 23-March-2010], 2009. 21

[ALR04] Algirdas Avižienis, Jean-Claude Laprie, and Brian Randell. Dependability and its threats: A taxonomy. In *Building the Information Society*, pages 91–120. 2004. 25, 26, 27, 28, 186, 261

[Ang10] Digital Angel. Homepage of digital angel. http://www.digitalangel.com/, 2010. [Online; accessed 23-March-2010]. 6

[BB03] F. Barbier and N. Belloir. Component behavior prediction and monitoring through built-in test. In *Engineering of Computer-Based Systems, 2003. Proceedings. 10th IEEE International Conference and Workshop on the*, pages 17–22, 2003. 32

[BBSN08] Saddek Bensalem, Marius Bozga, Joseph Sifakis, and Thanh-Hung Nguyen. Compositional verification for Component-Based systems and application. In *Proceedings of the 6th International Symposium on Automated Technology for Verification and Analysis*, pages 64–79, Seoul, Korea, 2008. Springer-Verlag. 30

[BCP07] Antonio Brogi, Carlos Canal, and Ernesto Pimentel. Behavioural types for service integration: Achievements and challenges. *Electron. Notes Theor. Comput. Sci.*, 180(2):41–54, 2007. 30

[BDH+98] Manfred Broy, Anton Deimel, Juergen Henn, Kai Koskimies, František Plášil, Gustav Pomberger, Wolfgang Pree, Michael Stal, and Clemens Szyperski. What characterizes a (software) component? *Software - Concepts & Tools*, 19(1):49–56, June 1998. 20, 185

[Ber96] Philip A. Bernstein. Middleware: a model for distributed system services. *Commun. ACM*, 39(2):86–98, 1996. 25

[BGS01] Michael Beigl, Hans-W. Gellersen, and Albrecht Schmidt. Mediacups: experience with design and use of computer-augmented everyday artefacts, 2001. 8

[Bla04] J. Blau. Supermarket's futuristic outlet. *Spectrum, IEEE*, 41(4):21–22, 25, April 2004. 6

[BLS05] Mike Barnett, Leino, and Wolfram Schulte. The spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362/2005, pages 69, 49. Springer, 2005. 32

[BMM09] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 213–222, Amsterdam, The Netherlands, 2009. ACM. 167

[BMMS+06] Jürgen Branke, Moez Mnif, Christian Müller-Schloer, Holger Prothmann, Urban Richter, Fabian Rochner, and Hartmut Schmeck. Organic Computing – Addressing complexity by controlled self-organization. In *Proceedings of the 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2006)*, November 2006. 4

[BMP07] Antonia Bertolino, Henry Muccini, and Andrea Polini. Architectural verification of Black-Box Component-Based systems. In *Rapid Integration of Software Engineering Techniques*, pages 98–113. 2007. 33

[Bro93] Manfred Broy. Functional specification of time-sensitive communicating systems. *ACM Trans. Softw. Eng. Methodol.*, 2(1):1–46, 1993. 3

[BRSV00] Klaus Bergner, Andreas Rausch, Marc Sihling, and Alexander Vilbig. Putting the parts together – concepts, description techniques, and development process for componentware. In *HICSS 33, Proceedings of*

the Thirty-Third Annual Hawaii International Conference on System Sciences. IEEE Computer Society, Jan 2000. 1

[CCMW01] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (wsdl) 1.1. Technical report, W3C, 2001. 23, 193

[CFHL07] Xi Chen, Juejing Feng, Martin Hiller, and Vera Lauer. Application of software watchdog as a dependability software service for automotive safety relevant systems. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 618–624. IEEE Computer Society, 2007. 139

[Chi05] M. M. Islam Chisty. An introduction to java annotations. *Developer.com*, 2005. 166, 179

[Chu36] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345–363, 1936. 30

[Cor08a] Microsoft Corporation. Msdn documentation of the program compatibility assistant in windows vista. http://msdn.microsoft.com/en-us/library/bb756937.aspx, 2008. [Online; accessed 23-March-2010]. 35, 261

[Cor08b] Microsoft Corporation. Windows logo program. http://www.microsoft.com/whdc/winlogo/, 2008. [Online; accessed 23-March-2010]. 34

[Cor08c] Microsoft Corporation. Windows update. http://windowsupdate.microsoft.com/, 2008. [Online; accessed 23-March-2010]. 34

[Cor 6] International Data Corporation. Information industry update. 1995-6. 6, 261

[CPS09] Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Larva – safer monitoring of Real-Time java programs. Hanoi, Vietnam, 2009. 33

[CS05] Christoph Csallner and Yannis Smaragdakis. Check 'n' crash: combining static checking and testing. In *Proceedings of the 27th international conference on Software engineering*, pages 422–431, St. Louis, MO, USA, 2005. ACM. 30

[DJ08] Dino Distefano and Matthew J. Parkinson J. jStar: towards practical verification for java. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 213–226, Nashville, TN, USA, 2008. ACM. 31

[Dow97] Mark Dowson. The Ariane 5 software failure. *SIGSOFT Softw. Eng. Notes*, 22(2):84, 1997. 10

[dRA04] Boris de Ruyter and Emile Aarts. Ambient intelligence: visualizing the future. In *AVI '04: Proceedings of the working conference on Advanced visual interfaces*, pages 203–208, New York, NY, USA, 2004. ACM. 7

[Dum09] Daniel Dumas. Aug. 28, 1988: Ramstein air show disaster kills 70, injures hundreds. http://www.wired.com/thisdayintech/2009/08/0828ramstein-air-disaster/ [Online; accessed 23-March-2010], 2009. 37

[Erl05] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall International, illustrated edition edition, August 2005. 22, 23, 24, 261

[fESE09] Fraunhofer-Institute for Experimental Software Engineering. Bilateral german–hungarian collaboration project on ambient intelligence systems. http://www.belami-project.org/, 2009. [Online; accessed 23-March-2010]. 8

[FGG+06] Peter Feiler, Richard P. Gabriel, John Goodenough, Rick Linger, Tom Longstaff, Rick Kazman, Mark Klein, Linda Northrop, Douglas Schmidt, Kevin Sullivan, and Kurt Wallnau. Ultra-large-scale systems: The software challenge of the future. Technical report, The Carnegie Mellon Software Engineering Institute, 2006. 1, 4

[fIGiVI03] Gesellschaft fuer Informatik (GI) and Informationstechnische Gesellschaft im VDE (ITG). Organic computing, computer- und systemarchitektur im jahr 2010. Position Paper, 2003. 4

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, Berlin, Germany, 2002. ACM. 29, 30, 33

[For06] UPnP Forum. Upnp device architecture 1.0. Technical report, UPnP Forum, 2006. 36

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Softw.*, 12(6):17–26, 1995. 3

[Gar09] Inc. Gartner. Gartner says grey-market sales and destocking drive worldwide mobile phone sales to 309 million units; smartphone sales grew 13 per cent in third quarter of 2009. http://www.gartner.com/it/page.jsp?id=1224645, 2009. [Online; accessed 23-March-2010]. 6

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, illustrated edition edition, November 1994. 166

[GMPS97] Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers. Going beyond the sandbox: an overview of the new security architecture in the java development kit 1.2. In *USITS'97: Proceedings of the USENIX Symposium on Internet Technologies and Systems on USENIX Symposium on Internet Technologies and Systems*, pages 10–10, Berkeley, CA, USA, 1997. USENIX Association. 134

[Gro00] Object Management Group. Trading object service specification. http://www.omg.org/cgi-bin/doc?formal/2000-06-27 [Online; accessed 23-March-2010], 2000. 22

[Gro04a] Hans-Gerhard Gross. *Component-based Software Testing With Uml*. SpringerVerlag, 2004. 20, 185

[Gro04b] Object Management Group. Common object request broker architecture (corba) specification, version 3.1. http://www.corba.org/ [Online; accessed 23-March-2010], 2004. 21, 127, 193

[Gro05] Manchester Triage Group. *Emergency Triage*. BMJ Books, 2005. 38

[Gro07] Object Management Group. UML Testing Profile Webpage. http://www.omg.org/technology/documents/formal/test_profile.htm, 2007. [Online; accessed 23-March-2010]. 17

[GSQ07] Daniel Görlich, Peter Stephan, and Jan Quadflieg. Demonstrating remote operation of industrial devices using mobile phones. In *Mobility '07: Proceedings of the 4th international conference on mobile technology, applications, and systems and the 1st international symposium on Computer human interaction in mobile technology*, pages 474–477, New York, NY, USA, 2007. ACM. 8

[GWTB96] Ian Goldberg, David Wagner, Randi Thomans, and Eric Brewer. A secure environment for untrusted helper applications (confining the wily hacker). In *Proceedings of the Sixth USENIX UNIX Security Symposium*, San Jose, California, USA, July 1996. USENIX, USENIX Association. 134

[Her05] Scott Herrboldt. How to improve driver quality with Winqual / WHQL. In *Windows Hardware Engineering Conference (WinHEC)*. Microsoft Corporation, 2005. 34

[HKNR08] Sebastian Herold, Holger Klus, Dirk Niebuhr, and Andreas Rausch. Engineering of IT ecosystems: design of ultra-large-scale software-intensive systems. In *Proceedings of the 2nd international workshop on Ultra-large-scale software-intensive systems*, pages 49–52, Leipzig, Germany, 2008. ACM. 1

[Huc10] Thomas Huckle. Collection of software bugs, 2010. http://www5.in.tum.de/%7Ehuckle/bugse.html [Online; accessed 23-March-2010]. 10

[Ini09] Metro Group Future Store Initiative. MEA - der Mobile Einkaufsassistent. http://www.future-store.org/fsi-internet/get/documents/FSI/multimedia/pdfs/MEA%20-%20Der%20mobile%20Einkaufsassistent.pdf [Online; accessed 23-March-2010], 2009. 6, 9

[Ins07] The European Telecommunications Standards Institute. Testing & Test Control Notation Webpage. http://www.ttcn-3.org/, 2007. [Online; accessed 23-March-2010]. 17

[JTNK06] Thomas Jaitner, Marcus Trapp, Dirk Niebuhr, and Jan Koch. Indoor-simulation of team training in cycling. In *ISEA 2006*, Jul 2006. 8

[KDF04] Katarzyna Keahey, Karl Doering, and Ian Foster. From sandbox to playground: Dynamic virtual environments in the grid. In *GRID '04: Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*, pages 34–42, Washington, DC, USA, 2004. IEEE Computer Society. 134

[Ken06] Lawrence Kenny. Exploring the business and social impacts of pervasive computing. TA – SWISS, IBM Zurich Research Laboratory, 2006. 5

[KNR07] Holger Klus, Dirk Niebuhr, and Andreas Rausch. A component model for dynamic adaptive systems. In *ESSPE '07: International workshop on Engineering of software services for pervasive environments*, pages 21–28, New York, NY, USA, 2007. ACM. 7

[KNW06] Holger Klus, Dirk Niebuhr, and Oliver Weiß. Integrating sensor nodes into a middleware for ambient intelligence. In *Proceedings of the Workshop Building Software for Sensor Networks*, International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2006), Oct 2006. 127, 128

[KR06] Holger Klus and Andreas Rausch. A general architecture for self-adaptive ami components applied in speech recognition. In *SEAMS - Software Engineering for Adaptive and Self-Managing Systems*, Shanghai, China, May 2006. 2

[LBR06] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006. 32, 49

[Lev95] Nancy G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley Professional, April 1995. 26

[Lig02] Peter Liggesmeyer. *Software-Qualität. Testen, Analysieren und Verifizieren von Software*. Spektrum Akademischer Verlag, August 2002. 4, 28, 29, 31, 261

[LPY97] Kim G Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. 1997. 30

[LTW+06] Zhenmin Li, Lin Tan, Xuanhui Wang, Shan Lu, Yuanyuan Zhou, and Chengxiang Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pages 25–33, San Jose, California, 2006. ACM. 4

[Mat07] Friedemann Mattern. Was bedeuten Pervasive und Ubiquitous Computing? *asut-Bulletin*, (4):33, 2007. 5

[MBM+07] David Martin, Mark Burstein, Drew Mcdermott, Sheila Mcilraith, Massimo Paolucci, Katia Sycara, Deborah L. Mcguinness, Evren Sirin, and Naveen Srinivasan. Bringing semantics to web services with OWL-S. *World Wide Web*, 10(3):243–277, 2007. 23

[Mes09] Deutsche Messe. CeBIT fair – Centrum für Büroautomation, Informationstechnologie und Telekommunikation, 2009. http://www.cebit.de/ [Online; accessed 23-March-2010]. 37

[Mey92] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992. 32, 49

[Mic93] Microsoft. Com: Component object model technologies. http://www.microsoft.com/com/default.mspx [Online; accessed 23-March-2010], 1993. 21

[Mic96] Microsoft. Dcom technical overview. http://msdn.microsoft.com/en-us/library/ms809340.aspx [Online; accessed 23-March-2010], 1996. 21

[Mic07a] Microsoft. Microsoft .net framework. http://www.microsoft.com/net/ [Online; accessed 23-March-2010], 2007. 21

[Mic07b] Sun Microsystems. Java platform, enterprise edition. http://java.sun.com/javaee/ [Online; accessed 23-March-2010], 2007. 21

[Mic09] Microsoft. Managed extensibility framework, 2009. http://www.codeplex.com/MEF [Online; accessed 23-March-2010]. 21

[MKF06] G.C. Murphy, M. Kersten, and L. Findlater. How are java software developers using the elipse IDE? *IEEE Software*, 23(4):76–83, 2006. 167

[MRP+07] Anton Michlmayr, Florian Rosenberg, Christian Platzer, Martin Treiber, and Schahram Dustdar. Towards recovering the broken SOA triangle: a software engineering perspective. In *2nd international workshop on Service oriented software engineering: in conjunction with the 6th ESEC/FSE joint meeting*, pages 22–28, Dubrovnik, Croatia, 2007. ACM. 24, 261

[NBKL06] Jürgen Nehmer, Martin Becker, Arthur Karshmer, and Rosemarie Lamm. Living assistance systems: an ambient intelligence approach. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 43–50, New York, NY, USA, 2006. ACM. 8

[New06] Jan Newmarch. Jan newmarch's guide to jini technologies, 2006. http://jan.newmarch.name/java/jini/tutorial/Jini.html [Online; accessed 23-March-2010]. 22

[Nie09] Dirk Niebuhr. DAiSI - Infrastruktur für Dynamisch Adaptive Systeme – Verlässliche IT Ökosysteme am Beispiel eines Rettungsassistenzsystems, 2009. http://bit.ly/dBs0dS [Online; accessed 23-March-2010]. 37

[Nie10] Dirk Niebuhr. Rettungsassistenzsystem für Katastrophen und Großschadenslagen, 2010. http://www2.in.tu-clausthal.de/%7ERettungsassistenzsystem/video.php [Online; accessed 10-March-2010]. 37

[NKA+07] Dirk Niebuhr, Holger Klus, Michalis Anastasopoulos, Jan Koch, Oliver Weiß, and Andreas Rausch. Daisi - dynamic adaptive system infrastructure. Technical report, Fraunhofer Institut Experimentelles Software Engineering, IESE-Report No. 051.07/E, Jun 2007. 125

[NRK+10] Dirk Niebuhr, Andreas Rausch, Holger Klus, André Appel, Cornel Klein, Jürgen Reichmann, and Reiner Schmid. Method and apparatus for determining a component conforming to the requirements. Patent pending (Patent Nr. WO 2010/040605 A2), 2010. 176

[NSH09] Dirk Niebuhr, Mirco Schindler, and Dirk Herrling. Emergency assistance system – webpage of the cebit exhibit 2009, 2009. http://www2.in.tu-clausthal.de/%7ERettungsassistenzsystem/en [Online; accessed 23-March-2010]. 15, 37

[NSRD08] E. Naroska, K. Scherer, C. Ressel, and T. Dimitrov. Ambient intelligence to support people at home. 2008. 7

[otN09] Royal Philips Electronics of the Netherlands. Ambilight technology. http://www.consumer.philips.com/c/televisions/33092/cat/gb/ [Online; accessed 23-March-2010], 2009. 7

[PBC+06] Neha Padmanabhan, Frada Burstein, Leonid Churilov, Jeff Wassertheil, Bernard Hornblower, and Nyree Parker. A mobile emergency triage decision support system evaluation. In *Proceedings of the 39th Annual Hawaii International Conference on System Sciences - Volume 05*, page 96.2. IEEE Computer Society, 2006. 41

[Pre99] Larry Press. Personal computing: the post-pc era. *Commun. ACM*, 42(10):21–24, 1999. 5

[Rau01] Andreas Rausch. *Componentware: Methodik des evolutionären Architekturentwurfs*. Herbert Utz Verlag, PhD Thesis at Technische Universität München, Nov 2001. 22

[Rau02] Andreas Rausch. "design by contract"+ "componentware"= "design by signed contract". *Journal of Object Technology*, 1(3), Jul 2002. Special issue: TOOLS USA 2002. 22

[Rau05a] Andreas Rausch. Towards a formal foundation for dynamic evolutionary systems. In *Proceedings of the Workshop on Architecture-Centric Evolution (ACE 2005)*, the 19th European Conference on Object-Oriented Programming (ECOOP 2005), Jul 2005. 22

[Rau05b] Andreas Rausch. Towards dynamic evolutionary systems. In *Proceedings of the Workshop on Architecture-Centric Evolution (ACE 2005)*, the 19th European Conference on Object-Oriented Programming (ECOOP 2005), Jul 2005. 15

[Rau07] Andreas Rausch. DisCComp – A Formal Model for Distributed Concurrent Components. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 176(2):5–23, 2007. 22

[RKL+05] Dumitru Roman, Uwe Keller, Holger Lausen, Jos de Bruijn, Rubén Lara, Michael Stollberg, Axel Polleres, Cristina Feier, Cristoph Bussler, and Dieter Fensel. Web service modeling ontology. *Appl. Ontol.*, 1(1):77–106, 2005. 23

[SBL05] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. *ACM Trans. Comput. Syst.*, 23(1):77–110, 2005. 33

[Sec92] International Electrotechnical Commission (IEC) Secretariat. Industrial-process measurement and control – evaluation of system properties for the purpose of system assessment. part 5: Assessment of system dependability, publication 1069-5. Technical report, 1992. 25

[SHDC06] Randall B. Smith, Bernard Horan, John Daniels, and Dave 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*, pages 706–707, New York, NY, USA, 2006. ACM. 40

[SIG05] Bluetooth SIG. Hands-free profile 1.5. Technical report, Bluetooth SIG, 2005. 10, 36

[SIG07] Bluetooth SIG. Specification of the bluetooth system. specification version 2.1. Technical report, Bluetooth SIG, 2007. 10, 36

[Som06] Ian Sommerville. *Software Engineering*. Addison Wesley, 8 edition, June 2006. 28

[SS03] Gitanjali Swamy and Sanjay Sarma. Manufacturing cost simulations for low cost RFID systems. Technical report, Auto-ID Center, Massachusetts Institute of Technology, 2003. 6

[SS06] Fredrik Seehusen and Ketil Stølen. Information flow property preserving transformation of UML interaction diagrams. In *Proceedings of the eleventh ACM symposium on Access control models and technologies*, pages 150–159, Lake Tahoe, California, USA, 2006. ACM. 3

[Szy02] Clemens Szyperski. *Component Software*. Addison Wesley Publishing Company, 2002. 1, 20, 21, 185

[Tec09] Crossbow Technology. Homepage of crossbow technology, 2009. http://www.xbow.com/ [Online; accessed 23-March-2010]. 40

[Tel06] Telephia. European subscriber and device report q1 2006. 2006. 10

[Tos09] Toshiba. Firmware update documentation – firmware 1.5 for the HD DVD players HD-XE1 and HD-E1. http://www.toshibahddvd.co.uk/media/File/firmware_update_1.5.pdf, 2009. [Online; accessed 23-March-2010]. 9

[Tur36] Alan Mathison Turing. On computable numbers, with an application to the Entscheidungsproblem. In *Proceedings of the London Mathematical Society*, pages 230–265, 1936. 30

[US04] Pauliina Ulkuniemi and Veikko Seppanen. Cots component acquisition in an emerging market. *IEEE Softw.*, 21(6):76–82, 2004. 21

[VHT00] Antonio Vallecillo, Juan Hernandez, and Jose M Troya. Component interoperability. *ECOOP '99 READER, NUMBER 1743 IN LNCS*, pages 1—21, 2000. 36, 103

[Vog03] Werner Vogels. Web services are not distributed objects. *IEEE Internet Computing*, 7(6):59–66, 2003. 23

[W3C07] W3C. Simple object access protocol (soap) specification. http://www.w3.org/TR/soap/ [Online; accessed 23-March-2010], 2007. 23, 36

[Wei81] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449, San Diego, California, United States, 1981. IEEE Press. 29

[Wei91] Mark Weiser. The computer for the 21st century. *Scientific American*, 02/1991 1991. 5

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 203–216, New York, NY, USA, 1993. ACM. 134

List of Figures

List of Tables

List of Listings