A Framework for Non-Interference in Component-Based Systems

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Simon Greiner

aus Freising

Tag der mündlichen Prüfung:	29. Januar 2018
Erster Gutachter:	Prof. Dr. Bernhard Beckert
Zweiter Gutachter:	Prof. Dr. Andrei Sabelfeld

A Framework for Non-Interference in Component-Based Systems

Simon Greiner

Ich versichere wahrheitsgemäß, die Dissertation bis auf die dort angegebenen Hilfen selbständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer und eigenen Veröffentlichungen unverändert oder mit Änderungen entnommen wurde.

Karlsruhe, 28. Februar 2018

Simon Greiner

Acknowledgements

I would like to thank my supervisor Prof. Bernhard Beckert for giving me the opportunity to work on many interesting research questions, providing helpful support whenever it was necessary and giving me the freedom to pursue any path that seemed interesting. I would also like to thank Prof. Andrei Sabelfeld for agreeing to act as second reviewer of this thesis. Further, I am thankful to Prof. Ralf Reussner and Prof. Gregor Snelting for their valuable feedback on early presentations of this thesis and for agreeing to be examiners.

I would like to thank Prof. Peter H. Schmitt who originally gave me the idea to pursue a career in research while I was still a student.

Many different people helped me on the way to this thesis by fruitful discussions, on and off-topic. I would like to thank my fellow researchers who I had the chance to work with in the course of the RS^3 and KASTEL projects. In particular, I would like to thank Kaibin Bao, Dr. Pascal Birnstill, Dr. Florian Böhl, Martin Hecker, Dr. Max Kramer, Dr. Erik Krempel, Martin Mohr, and Kateryna Yurchenko. Also, I would like to thank my current and former colleagues at the working group. Dr. Thorsten Bormer, Dr. David Farago, Dr. Daniel Grahl, Sarah Grebing, Mihai Herda, Michael Kirsten, Dr. Vladimir Klebanov, Tianhai Liu, Dr. Florian Merz, Dr. Christoph Scheben, Dr. Mattias Ulbrich, and Alexander Weigl always found the time for discussions on open questions, provided helpful feedback, answered one of the many questions, or just joined into pointless discussions on the dinner table. Special thanks go to everyone who agreed to read preliminary chapters of this thesis and provided very helpful feedback on improving them. In particular I would like to thank Karsten Diekhoff and Jonas Krämer who helped me with implementations, especially during the last months of this thesis.

Last but not least, I would like to thank my family and friends who provided support, and distraction in countless different ways and more and less severe situations. Their trust and help was fundamental throughout the years of work on this thesis.

The work on this thesis was financially supported by the Bundesministerium für Bildung und Forschung as part of the KASTEL projects.

Abstract

Modern IT systems are required to implement complex functionalities and support high scalability. At the same time, security properties like confidentiality and integrity become more and more important for these systems.

An often used approach to realize such systems is component-based system engineering, where different parts of the software are distributed to several components, each realizing parts of the functionality as services and using other components for required functionalities. These components can then be distributed on different machines which allows high flexibility when deploying the overall system. Further, component-based systems engineering supports re-use of existing components in new contexts in possibly very different usage scenarios.

A very popular approach for the specification and analysis of confidentiality and integrity properties for software is using the concept of non-interference, which describes a strict property of the allowed flow of information in a software system. With non-interference specifications, inputs and outputs of a system are labeled as secret (high) or public (low) information. The program is non-interferent, if the public outputs are not influenced by secret inputs. Typically, the separation of information into high or low is performed according to an analysis of potential attackers to the system.

In this thesis, we present a novel and general framework for non-interference in component-based systems. By exploiting restrictions to the programming model of component-based systems the framework allows a very precise specification of intended information flows in a system. The partition of inputs and outputs into high and low values is based on equivalence relations and thus allows the classification of partial information and the existence of service calls to be secret or public. The resulting non-interference property is compositional, a central requirement in the case of component-based systems.

Further, we present as part of the framework a notion of non-interference as a services-local information flow property and show that non-interferent services can be composed in non-interferent components. As a result, it is sufficient to analyze the security of services, i.e. small and often relatively simple programs, in order to gain a security guarantee for entire components and component-based systems. We introduce the idea of dependency clusters as service-local, attacker-independent, and compositional building blocks for information flow specifications and show how these dependency clusters can be used to gain system-wide security guarantees. Dependency clusters are especially useful in the context of evolving components, since they allow to reduce the overhead of software analysis when re-using components in new environments, or adding, removing or changing services.

We show the practical use of our general framework by instantiating it for two concrete tools.

For one, we instantiated the framework to gain an extension of the Palladio Component Model (PCM), a graphical specification language specially designed for component-based systems. Our extension allows an intuitive specification of security properties for components. We map the general definitions of the framework to Palladio entities and show that the compositionality properties of non-interference holds for PCM components. The presented specification approach was used in several case-studies for security specification of systems.

We also present an instantiation of our framework for a deductive software verification tool for JavaEE, a programming framework for distributed components implemented in Java. We present a novel non-interference specification technique on service- and component-level for JavaEE components based on dependency cluster by extending the Java Modeling Language. We also extend the KeY calculus, an approach for deductive verification of Java programs and thus gain a program verification tool non-interference specifications of JavaEE components. We apply the verification tool to a web-shop system as a proof-of-concept.

Apart from our contributions presented in this thesis, the framework was used by other researchers to build novel analysis techniques for components, e.g. based on program dependency graphs and automatic software testing approaches.

We finally discuss an extension of our framework, which allows non-interference specification of input information to depend on previous inputs to the system. In this extension, we sacrifice compositionality of non-interference for a more expressive specification language. We sketch a concrete idea for this extension by specifying and verifying a history-based non-interference property for a concrete system.

Zusammenfassung

Moderne IT Systeme müssen komplexe Funktionen umsetzten und gleichzeitig hochgradig skalierbar sein. Ebenso nimmt die Bedeutung von Sicherheitseigenschaften wie Geheimhaltung und Integrität von Informationen für solche Systeme stetig zu.

Eine gängige Technik zur Umsetzung solcher System ist komponentenbasierte Systementwicklung, in der Teile der Software auf unterschiedliche Komponenten verteilt werden. Jede Komponente realisiert einen Teil des Gesamtfunktionalität als Service, und nutzt andere Komponenten für hierfür benötigte Funktionalitäten. Komponenten können auf unterschiedliche Rechner verteilt werden, wodurch eine hohe Flexibilität beim Ausbringen des Gesamtsystems erreicht wird. Komponentenbasierte Systementwicklung unterstützt außerdem die Wiederverwendung von existierenden Komponenten in neuen Umgebungen und Anwendungsszenarien.

Nichtinterferenz ist ein beliebtes Konzept zur Beschreibung und Analyse von Geheimhaltungs- und Integritätseigenschaften von Software. Sie beschreibt eine strenge Eigenschaft für erlaubte Informationsflüsse innerhalb eines Softwaresystems. Ein- und Ausgaben eines Systems werden mittels Nichtinterferenzspezifikationen als geheim oder öffentlich markiert. Ein Programm erfüllt Nichtinterferenz, wenn öffentliche Ausgaben nicht durch geheime Eingaben beeinflusst werden.

In dieser Arbeit präsentieren wir ein neuartiges und allgemeines Rahmenwerk für Nichtinterferenz in komponentenbasierten Systemen. Wir nutzen Einschränkungen die üblicherweise für Komponenten gemacht werden um so sehr präzise Spezifikationen für erlaubte Informationsflüsse erreichen zu können. Die Aufteilung von Ein- und Ausgaben in öffentlich und geheim basiert auf Äquivalenzrelationen, so dass auch die Klassifikation von Teilinformationen, sowie die reine Existenz von Ein- und Ausgaben ermöglicht wird. Die resultierende Nichtinterferenzeigenschaft ist kompositional und erfüllt somit eine grundlegende Voraussetzung um für komponentenbasierte Systeme geeignet zu sein.

Wir präsentieren des Weiteren als Teil des Rahmenwerkes einen neuartigen Nichtinterferenz-Begriff als Service-lokale Informationsflusseigenschaft, und weisen nach, dass nichtinterferente Services zu nichtinterferenten Komponenten zusammengestellt werden können. Demnach ist es ausreichend, einzelne Service, d.h. kleine und oftmals relativ einfache Programme, zu analysieren, um Sicherheitsgarantien für ganze Komponenten und komponentenbasierte Systeme zu erhalten. Wir führen Dependency Cluster als servicelokale und angreiferunabhängige, kompositionale Bausteine für Informationsflussspezifikationen ein. Ebenso zeigen wir, wie Dependency Cluster genutzt werden können um systemweite Sicherheitsgarantien zu erhalten. Insbesondere im Kontext von evolvierenden Systemen sind Dependency Cluster nützlich, da sie den Mehraufwand für die Analyse von Sicherheitseigenschaften auf ein Minimum reduzieren, wenn bereits existierende Komponenten in einem neuen Umfeld wiederverwendet werden.

Wir zeigen den praktischen Nutzen unseres Rahmenwerkes indem wir zwei konkrete Werkzeuge auf dieser Basis entwickeln.

Einerseits erweitern wir das graphische Spezifikationswerkzeug Palladio Component Model (PCM) auf Basis unseres Rahmenwerkes. Unsere Erweiterung ermöglicht die intuitive Spezifikation von Sicherheitseigenschaften für Komponenten. Wir bilden Elemente aus dem PCM auf Definitionen aus unserem Rahmenwerk ab und zeigen so, dass Kopositionalitätseigenschaften aus dem Rahmenwerk auch für PCM Komponenten gelten. Das Spezifikationswerkzeug wurde in mehreren Fallstudien für die Spezifikation von Sicherheitseigenschaften für Systeme eingesetzt.

Ausserdem präsentieren wir einen deduktiven Software Verifikationsansatz für JavaEE Komponenten basierend auf unserem Rahmenwerk. Wie zeigen eine neuartige Spezifikationstechnik für Nichtinterferenzspezifikationen auf Service- und Komponentenebene als Erweiterung des Java Modeling Language, und aufbauend auf Dependency Clustern. Wir erweitern auch den KeY Kalkül, ein Ansatz zur deduktiven Verifikation von Java Programmen und erhalten so ein Werkzeug für die Verifikation von Sicherheitseigenschaften für JavaEE Komponenten. Wir wenden das Verifikationswerkzeug auf ein Web Shop System an.

Abschließend diskutieren wir eine Erweiterung unseres Rahmenwerkes, die es erlaubt Nichtinterferenzspezifikationen von Eingaben abhängig von vorherigen Eingaben zu formulieren. Wir opfern Kompositionalität von Nichtinterferenz um eine aussagekräftigere Spezifikationssprache für Sicherheitseigenschaften zu erhalten. Wir erörtern eine konkrete Ausgestaltung dieser Erweiterung an Hand einer Nichtinterferenzeigenschaft für ein konkretes System.

Contents

1 Introduction

1.1 Motivation

More and more, software systems become pervasive in our every-day life. They collect information which describes for individuals where they are, when they move, which movies they watch. At the same time, software more and more interact with their environment physically, for example as part of vacuum cleaners, industrial production lines, or as cars. These systems are highly distributed and highly interconnected and more often than not, they are interconnected via the internet, which makes them easily accessible for everyone. A lack of security therefore can have profound consequences on people's lives.

As a result of software becoming more pervasive, while methods for securing software do not keep pace, reports of security problems with autonomous systems become more frequent in the media. In 2017 a security company demonstrated how easy it is to turn a vacuum cleaner into a spy tool[1]. In 2016 an attack on network router[2] caused a widespread disruption of the internet in Germany and other parts of the world. In 2015 researchers demonstrated that it possible to take over control of a car[3] (even without autonomous driving abilities built-in). These examples are only few incidents which illustrate how profound of an impact on the people's privacy and health can be caused by insecure distributed systems.

[1]https://www.forbes.com/sites/thomasbrewster/2017/10/26/lg-hom-bot-robot-hoover-hacked-into-surveillance-device/

[2]https://www.wired.de/collection/tech/das-mirai-botnet-koennte-hinter-den-telekom-ausfaellen-stecken

[3]https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/

1.2 Context of this Thesis

In this thesis, we present a general framework for specification and analysis of non-interference properties for distributed component-based systems. We aim the framework to serve as a theoretical basis for security requirements in the design process of a system, for program analysis in a quality assurance process, and for specifications during the implementation phase. We demonstrate that our framework can be instantiated in a concrete model-based design approach as well as for the specification and analysis of Java implementations of distributed component-based systems.

Our work therefore touches two general areas of research. The first area of research is concerned with the question of what are properties which a secure system has to satisfy. A very basic concept of security is the CIA triad, which states that a system is secure if it satisfies the three properties confidentiality, integrity, and availability. A common method to ensure that a system provides confidentiality and integrity of information is information flow security, or the strict form, non-interference. In a nut-shell, non-interference describes which information managed by a system may influence which output. While we concentrate on confidentiality in this thesis, we would like to mention that integrity is a very similar property from theoretical point of view.

The second research area tries to answer the question of how to develop distributed systems. Component-based system engineering provides methods for how to design and build a distributed system throughout the different phases of a development process. Our framework allows to integrate non-interference into this development process.

When developing a system, security, and information flow in particular, stretches over all phases of a development process. During requirement elicitation, it has to be clarified which inputs of a system contain sensitive information and which outputs may be available to a potential attacker. During system design it has to be ensured that the composition of a system satisfies its specified information flow property due to the behavior of individual components of the system. During implementation, the developer needs a specification on the level of components and services, which allows him to understand how a functionality can be realized, without knowing every detail about the overall system. And during quality assurance it has to be clear against which information flow specification a service, component, or system has to be checked.

In the following, we briefly discuss the context of research for component-based system engineering and non-interference as far as needed to make our contributions clear.

1.2.1 Component-Based System Engineering

Component-based system engineering is a system design and development approach where the functionality of a system is distributed over several components. What constitutes a component is very loosely defined in the literature. One of the most commonly accepted definitions for software components is by Szyperski et al. [2002]: "A software component is a *unit of composition* with contractually specified *interfaces* and explicit *context dependencies* only. A software component can be deployed independently and is subject to composition by third parties". Other definitions, e.g. by Reussner et al. [2016], Meyer [2003], or Heineman and Councill [2001], go in a very similar direction. According to these definitions, components provide a block-box view of their functionality and are designed for re-use in different contexts.

A more concrete definition of components can be found in practical frameworks for implementing components. Components in Java are called beans (JavaEE [2013]), in .Net (Wigley et al. [2003]) they are called assemblies. These frameworks, among others, make restrictions on how components may be implemented, e.g. they may restrict concurrent program execution within one component or restrict components from using a shared state.

What many concepts for components have in common is that often the functionality of a component is provided in the form of services, i.e. relatively small sequential programs. Each service can call services which are provided by the environment, or, in a composed system, by other components. Especially when components are distributed in the sense that they run on physically different machines, the only allowed or possible way of communication between components is via remote service calls.

Specification languages for security in component-based systems are mainly concerned with access control (Nguyen et al. [2015]). In approaches which allow specification of information flow properties, these specifications are either local to isolated activities (Jürjens [2005]), part of the specification of the dynamic behavior of a single component (Stenzel et al. [2014]) and therefore not easily relateable to the information flow of an entire system, or they lack a theoretic semantic basis (Hafner and Breu [2009]) which would allow reasoning about whether an implementation satisfies the resulting specification.

1.2.2 Information Flow

Components are interactive programs in the sense that they communicate with their environment via sending and receiving messages.

Several approaches to formalize non-interference for interactive programs were studied in the past, for example, by Focardi and Gorrieri [1994],Ryan and Schneider,Mantel,Sabelfeld and Mantel [2002], or Pottier. Non-interference in these notions is formalized by comparing traces of messages which a system can communicate. Generally speaking, an interactive program is non-interferent in this notion, if given two streams of inputs, which have equal non-sensitive information, the program produces outputs with equal non-sensitive information.

One drawback of these non-interference notions is that they assume all input to the system is pre-computed and the environment is not able to react on outputs it was able to observe. A second problem with these notions in the context of our work is that the environment, and therefore a potential attacker, is not part of the security model. This makes it hard to map a non-interference specification to an attacker model during requirements elicitation of a system.

We therefore focus on another line of work which uses explicit environments to model non-interference (e.g. Wittbold and Johnson,O'Neill et al.,Clark and Hunt [2009], and Rafnsson et al. [2012]). In this model a program is run in two environment, which provide the same non-sensitive inputs after observing the same non-sensitive overall behavior of the program. The program is secure, if every observation of non-sensitive outputs of the program in one environment is also possible in the other environment. Somebody observing the non-sensitive inputs and outputs of the program therefore can not decide under which of these two environments the program was run.

Non-interference notions with an explicit environment, as published in the literature, do allow the specification of message existence and message content to be non-sensitive. However so far no published notion allows to specify only parts of a message content to be non-sensitive, e.g. the last four digits of a credit card number. Also, existing non-interference definitions do not allow to specify that the existence of a message is non-sensitive depending on a parameter, e.g. a bank transaction with a transaction value of less than 5.000. The second drawback is that notions using an explicit environment are overly restrictive when the existence of a message is specified sensitive. Whenever the program sends an non-sensitive message after receiving a sensitive message, the program is deemed insecure, since the non-sensitive message reveals the existence of the previous sensitive message. The consequence for a secure distributed system is that it may never receive a sensitive message.

1.3 Contributions

We provide in this thesis a novel framework for requirement elicitation, analysis and specification of non-interference for distributed component-based systems. The framework consists of a collection of definitions of non-interference properties and theorems describing properties of these non-interference definitions and how they are related to each other. Apart from this, we provide individual contributions as part of the framework itself, and as part of two instantiations of our framework which show that it is practically applicable.

The following contributions are part of our framework itself.

- We provide the first notion of non-interference for interactive programs with an explicit environment model which is compositional under interleaving parallel composition, allows declassification of information, and the specification of message existence as sensitive. (Section 3.2)

- We provide the first notion of non-interference for component-based systems which allows declassification of information and the specification of message existence as sensitive. We prove that our notion of non-interference for components is compositional under synchronized parallel composition. (Section 3.3) In particular, this notion of non-interference for components is more precise than non-interference for interactive programs in general, in that non-interferent components can send non-sensitive outputs after receiving inputs whose existence is specified sensitive. We overcome this limitation of existing work by assuming that the environment behaves according assumptions which are common in component-based systems.

- We provide a novel non-interference notion for services, which extends state-of-the-art non-interference notions for sequential programs with message passing. We prove that a component is non-interferent if all services provided by the component are non-interferent. (Section 3.4) As a result, the analysis of a component for non-interference can be limited to analyzing each service modularly instead of the entire component.

- We define the novel concept of Dependency Clusters as a specification which describes ideally small dependencies of information between inputs, outputs, and states caused by the execution of individual services. (Chapter 4) Dependency cluster are service-local, i.e. whether or not a service is non-interferent w.r.t. a Dependency Cluster solely depends on the implementation of the service, and is independent from other service, other components, or the environment a component is deployed in. This way, Dependency Cluster simplify re-use and evolution of components with non-interference requirements.

  We prove that Dependency Clusters are compositional in the sense that if a service is non-interferent w.r.t. two Dependency Clusters, it is also non-interferent w.r.t. the composition of the two Dependency Clusters. This makes Dependency Clusters useful building blocks for complex non-interference specifications. Additionally, a service can be analyzed for each Dependency Cluster individually, especially with different analysis techniques. Thus, we allow to use different analysis techniques for different Dependency Clusters depending on their strengths and weaknesses, while still gaining non-interference specifications which may not be practically analyzable by one technique alone.

In the second part of this thesis, we instantiate our framework to show that it can be used to build specification tools and program analysis tools for practically relevant component-based systems.

- We provide a novel, intuitive specification language for non-interference properties as an extension of Palladio, a graphical specification language for component-based systems. (Chapter 7) Our extension of Palladio was used in multiple case-studies to specify information flow properties and perform security analysis for systems.
- We provide a novel specification language based on Dependency Cluster for component-based systems implemented in Java as an extension of the Java Modeling Language. We further provide rules and proof obligations which allow deductive verification of our specifications in the deductive program verification tool KeY. We apply this program analysis tool an a web shop system. (Chapter 8)

1.4 Outline

This thesis consists of three parts. In Part I we present our theoretical framework. In Part II we instantiate our framework as a concrete specification language for component-based systems and as a analysis technique for concrete implementations of component-based systems. In Part III we discuss an extension of our framework which allows more expressive non-interference specifications on the example of a smart surveillance system.

1.4.1 Part I: A Framework for Non-Interference in Component-Based Systems

In Chapter 2 we formally define components as distributed service components (DSC) using Labeled Transition Systems (LTS) as the formal basis, and we define synchronous composition of components.

In Chapter 3 we discuss non-interference for DSCs from a conceptual point of view. We first define a notion of non-interference with an explicit environment for LTS, which supports declassification of information as well as the specification of message existence as sensitive. We formally prove that non-interference for LTS is compositional under asynchronous composition. After that we define non-interference for DSCs by only considering environments which adhere to assumptions that are common in component-based approaches. We prove that non-interference for DSCs is compositional. Finally, we define non-interference for services and show that service-non-interference implies non-interference for DSCs.

In Chapter 4 we switch to a constructive point-of-view on our framework. We introduce Dependency Cluster as non-interference specifications for services. Dependency cluster are modular and compositional building blocks for non-interference specifications which describe small dependencies caused by the execution of a service. We show that non-interference for DSCs can be verified by checking a first-order predicate-logic condition for Dependency Clusters of services, and that non-interference of a DSC w.r.t. a system-wide non-interference specification can be shown by checking a first order predicate logic condition for a non-interference specification of a DSC. Static analysis for non-interference specifications of component-based systems thus requires complicated program analysis only for Dependency Clusters of services, and therefore only for comparably small and simple programs w.r.t. small non-interference specifications.

In Chapter 5 we discuss work related to our framework and in Chapter 6 conclude the first part.

1.4.2 Part II: Instantiating the Framework

In the second part of this thesis we show that the theoretical framework from Part I can be used as the basis for implementing tools for system engineering and program analysis of component-based systems.

In Chapter 7, we define a specification language for non-interference as an extension of the practically used graphical specification language Palladio. By mapping specification primitives to elements of our framework, we show that results from Part I also hold for our graphical specification language. We also demonstrate that the relation between a model of potential attackers of a system and a non-interference specification is intuitive.

In Chapter 8 we extend the deductive program verification tool KeY to support reasoning about Java beans and extend the Java Modeling Language with specifications for Dependency Cluster of remote methods, the service-equivalent in Java beans. We provide proof obligations for Dependency Cluster specifications in JavaDL, the logic underlying KeY. This way we demonstrate that our framework can serve as the basis for program analysis techniques for real programming languages.

1.4.3 Part III: Beyond the Framework

In the third part, we discuss in Chapter 9 an extension of our framework which allows more expressive specifications. According to our framework, the specification of sensitive information is limited to single messages. We discuss the example of a smart surveillance system, where a domain-motivated specification of sensitive information depends on the history of previously communicated messages, not only the message itself. We provide a formal specification, and during that lift restrictions on our framework by refining definitions from Part I. Finally we verify that the example is secure w.r.t. the specification using KeY. The results in Chapter 9 should serve as an inspiration for future work which may want to extend our framework.

Part I

A Framework for Non-Interference in Component-Based Systems

2 Distributed Service Components

2.1 Introduction

Software components are defined in the literature rather informally. One of the most commonly accepted definitions for software components is by Szyperski et al. [2002]: "A software component is a *unit of composition* with contractually specified *interfaces* and explicit *context dependencies* only. A software component can be deployed independently and is subject to composition by third parties". Similar definitions for *software components* can be found for example by Reussner et al. [2016], Meyer [2003], and Heineman and Councill [2001].

The lack of concrete criteria makes it hard to formalize components in a mathematical sense. We need, however, a formal definition for a formal discussion of security properties in component-based systems. Frameworks for implementing component-based systems provide a more concrete clarification by the definition of the semantics of an implementation according to the respective programming environment. Examples for common frameworks for implementing potentially physically distributed component-based systems are the Java Enterprise Edition (EJB 3.1 Expert Group [2009]) or the .Net framework (Wigley et al. [2003]).

In this chapter, we provide a formal definition of software components. We concentrate on software components which are designed to run in a distributed fashion. We aim to provide a formal model, which is consistent with common informal definitions as well as commonly used implementation frameworks.

In the following section we introduce the computational model based on Labeled Transition Systems that we use for formalization in the first part of this thesis. In Section 2.3 we define Distributed Service Components (DSC) as our notion of components using the computational model. In Section 2.4 we define how DSCs are composed to larger systems, and after this we conclude the chapter.

The results in this chapter are based on work by the author (Greiner and Grahl [2016]).

2.2 Computational Model

A core property of DSCs in our setting is that they do not share a state and thus cannot communicate via variable manipulation. The only form of communication is via messages sent and received by a DSC (representing calls and terminations of services). Labeled Transition Systems (LTS) are a very general formalism for modeling programs communicating via messages.

The environment communicates with an LTS by passing messages over channels and receiving output messages from the LTS. We define $\mathbb{C}$ as a set of channels, over which values from a domain $\mathbb{D}$ can be communicated. We refer to the communication of a value over a channel as a message in $\mathbb{M} \subseteq \mathbb{C} \times \mathbb{D}$. An LTS can receive input messages $\mathbb{I}$ and send output messages $\mathbb{O}$, where $\mathbb{M} = \mathbb{I} \uplus \mathbb{O}$, and $\uplus$ is the disjoint union operator. We write $\alpha!v$ for a message $m \in \mathbb{O}$ communicating the value v on a channel α, $\alpha?v$ for $m \in \mathbb{I}$, and if it is not relevant whether m is an input or output, we write $\alpha.v$ for $m \in \mathbb{M}$. The transition relation $\rightarrow$ describes the transition of an LTS by communicating a message. We write $p \xrightarrow{m} p'$, if LTS p transitions to p' for some $m \in \mathbb{M}$. We write $p \xrightarrow{m}$ if there exists some p' such that $p \xrightarrow{m} p'$.

Further, we require an LTS not to discriminate on input acceptance due to the communicated value (1) and neither to provide indeterministic output (2) nor indeterministic internal behavior (3).

1. $\forall \alpha \in \mathbb{C}, v \in \mathbb{D} \cdot p \xrightarrow{\alpha?v} \implies \forall v' \in \mathbb{D} \cdot p \xrightarrow{\alpha?v'}$
2. If $p \xrightarrow{m_1} p_1$ and $p \xrightarrow{m_2} p_2$ and $m_1 \neq m_2$ then $m_1 \in \mathbb{I}$ and $m_2 \in \mathbb{I}$.
3. If $p \xrightarrow{m} p_1$ and $p \xrightarrow{m} p_2$ then $p_1 = p_2$.

The only source of indeterministic behavior is due to indeterministic input. In related work by Rafnsson et al. [2012], this restricted form of an LTS is called "input-output LTS." For simplicity, we refer to them as LTS in the remainder.

Traces $\mathbb{T}$ are finite lists of messages, where $\langle\rangle \in \mathbb{T}$ is the empty list and $\frown$ is the concatenation operator. The trace consisting of a single message m is $\langle m \rangle$. If it is clear from the context we write m instead of $\langle m \rangle$. The prefix relation is defined as $t_1 \leq t$ if $\exists t_2 \cdot t_1 \frown t_2 = t$. We say p can *communicate* a trace t, written $p \xrightarrow{t}$, if $t = \langle\rangle$ or $t = \langle m \rangle \frown t'$ and $p \xrightarrow{m} p'$ and $p' \xrightarrow{t'}$ for some p'. $\mathbb{T}(p) := \{t \mid p \xrightarrow{t} \}$ is the set of all traces, which p can communicate. For simplicity, we say a trace t contains a message m written $m \in t$, if $\exists t', t'' \cdot t' \frown m \frown t'' = t$.

For better readability, we use the following conventions: We refer to traces by t, channels by α and β, messages by m, values by v, and LTS by p and, if necessary by s. If necessary, we add to this notion indices and/or use its primed version.

2.3 Distributed Service Components and Services

We use the following intuitive understanding of components and refer to them as *Distributed Service Components* (DSCs): We assume a DSC is an entity that encapsulates its state and provides a set of functionalities in the form of services. Each service is a deterministic and sequential program and can itself call services provided by other DSCs. DSCs can communicate with each other only by calling services and providing and receiving parameters, but not via a shared state. DSCs are non-re-entrant, meaning while a service is executed by a DSC, other calls to services provided by the DSC are postponed until the current execution of the service terminates. DSCs can be composed to compositions by binding required services of one DSC to provided services of another DSC. In the rest of this section, we formalize this intuitive understanding of DSCs.

A service is defined by a name, a signature and the body of the program defining the service's behavior. The signature describes the input parameters of the service, the initial channel used for calling the service, and the termination channel used for communicating the return value. The function $\mathit{Ini}(\mathit{serv})$ defines the *initial channel* of the service serv, $\mathit{Fin}(\mathit{serv})$ the *termination channel*.

Example 2.1 (The `Cart` DSC)**.** As an example, see the following DSC modeling a shopping cart in an online shop:

```
Component Cart {
  int product, prodprice, prodamount;
  int countbuy, countpay, countcheck;

  int buy(int prod, int price, int amount) {...}
  (int, int, int) checkCart(int) {...}
  int clearCart(int) {...}
  int pay(int ccnr) {...}
  (int, int, int) getAllNums(int x) {...}
}
```

The DSC `Cart` provides services for adding a product to the cart (`buy`), checking the content of the cart (`checkCart`), removing items (`clearCart`), paying for the products in the cart (`pay`), and, for analysis and debugging purposes, receiving information on how the cart is used (`getAllNums`).

In the example, the following following initial channels are defined:
$\mathit{Ini}(\mathit{buy})$, $\mathit{Ini}(\mathit{checkCart})$, $\mathit{Ini}(\mathit{clearCart})$, $\mathit{Ini}(\mathit{pay})$, $\mathit{Ini}(\mathit{getAllNums})$ Further, the respective termination channels are defined:
$\mathit{Fin}(\mathit{buy})$, $\mathit{Fin}(\mathit{checkCart})$, $\mathit{Fin}(\mathit{clearCart})$, $\mathit{Fin}(\mathit{pay})$, $\mathit{Fin}(\mathit{getAllNums})$

The domain $\mathbb{D}$ in our example contains the natural numbers and tuples of natural numbers.

$$\textsc{Skip}\ \frac{}{\langle \mathit{SKIP};\sigma\rangle \to \langle \mathit{SKIP};\sigma\rangle} \qquad \textsc{Seq1}\ \frac{}{\langle \mathit{SKIP};c_2;\sigma\rangle \to \langle c_2;\sigma\rangle}$$

$$\textsc{Seq2}\ \frac{\langle c_1;\sigma\rangle \xrightarrow{t} \langle c_1';\sigma'\rangle \quad c_1 \neq \mathit{SKIP}}{\langle c_1;c_2;\sigma\rangle \xrightarrow{t} \langle c_1';c_2;\sigma'\rangle}$$

$$\textsc{Assign}\ \frac{\sigma(e) = v}{\langle x := e;\sigma\rangle \to \langle \mathit{SKIP};\sigma[x := \sigma(e)]\rangle}$$

$$\textsc{If1}\ \frac{\sigma(e) \neq 0}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2;\sigma\rangle \to \langle c_1;\sigma\rangle}$$

$$\textsc{If2}\ \frac{\sigma(e) = 0}{\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2;\sigma\rangle \to \langle c_2;\sigma\rangle}$$

$$\textsc{While}\ \frac{}{\begin{array}{c}\langle \texttt{while}\ e\ \texttt{do}\ c_1;\sigma\rangle \to \\ \langle \texttt{if}\ e\ \texttt{then}\ (c_1;\texttt{while}\ e\ \texttt{do}\ c_1)\ \texttt{else}\ \mathit{SKIP};\sigma\rangle\end{array}}$$

$$\textsc{Service}\ \frac{\mathit{servC} = \mathit{Ini}(\mathit{serv}) \quad \mathit{servR} = \mathit{Fin}(\mathit{serv})}{\langle x := \texttt{serv(}e\texttt{)};\sigma\rangle \to \quad \langle \texttt{write(}e \to \mathit{servC}\texttt{)};\texttt{read(}x \leftarrow \mathit{servR}\texttt{)};\sigma\rangle}$$

$$\textsc{Send}\ \frac{\sigma(e) = v}{\langle \texttt{write(}e \to \alpha\texttt{)};\sigma\rangle \xrightarrow{\alpha!v} \langle \mathit{SKIP};\sigma\rangle}$$

$$\textsc{Rec}\ \frac{e \in \mathbb{D}}{\langle \texttt{read(}x \leftarrow \alpha\texttt{)};\sigma\rangle \xrightarrow{\alpha?e} \langle \mathit{SKIP};\sigma[x := e]\rangle} \qquad \textsc{Ext1}\ \frac{\langle c_1;\sigma\rangle \xrightarrow{\alpha?x} \langle c_1';\sigma'\rangle}{\langle c_1 \sqcap c_2;\sigma\rangle \xrightarrow{\alpha?x} \langle c_1';\sigma'\rangle}$$

Figure 2.1: Semantics for the example language

The *body* of a service is a program consisting of the language primitives from Figure 2.1. We give the semantics of the language with respect to a state σ. A *state* is a function mapping program variables to values from the domain $\mathbb{D}$. In the remainder, σ and its primed and indexed counterparts refer to states.

Example 2.2 (State Variables in `Cart`). `Cart` defines the state variables `product`, `prodprice`, `prodamount`, `countbuy`, `countpay`, and `countcheck`.

We provide the simple while-language in Figure 2.1 as an illustration for how services can be implemented. We do not require services to be actually implemented in this language, but provide general properties DSCs and services have to satisfy in order to be non-interferent. These properties are language independent but require some assumptions on the implementation of services, for example that services can not spawn threads and are deterministic. These assumptions are made explicit in the further presentation. Extending the language with additional features, e.g. objects, exceptions and DSC-internal method calls not causing communication of a message, therefore does not invalidate our results presented in the first part of this thesis. However, considering a more complicated (and realistic) programming language at this point would complicate the presentation without any considerable benefit.

The rule SERVICE in Figure 2.1 defines the semantics of a service call. A service call consists of the sending of a message on the initial channel and providing some value representing a parameter. After sending, the service waits for the response of the called service on the termination channel. Semantics of sending and receiving messages is shown in Figure 2.1. We refer to the body of a service *serv* by $body_{serv}$.

Example 2.3 (Body of the Service `pay`)**.** When service `pay` is called, the state variables `paycount` is increased and the sale is registered using another service `registerSale`. During service call, the values provided as parameters for the call are taken from the respective state variables. The variable *param* holds the parameter passed to `pay` when it is called, and *param#x* refers to the x-th element of the tuple stored in *param*. The return value is written to the variable *res*.

The body of `pay` implemented in the Cart DSC according to our simple while language looks as follows:

```
$body_{pay} :=$
    $payCount++;$
    $registerSale(prodId, price, amount, param\#1);$
    $res = 1;$
```

The services defined by the language introduced above are limited to a single parameter. This limitation is not a restriction of the expressiveness of our language, since the parameter can be considered to be some encoding of several parameters. An example for this is shown in the listing for the service body of `pay` above. If a parameter encodes a tuple of values, we refer to the x-th element of the tuple p by $p\#x$.

The *handler* of a service represents the program which is executed when a service is called. Initially, the service is started by a message on channel $Ini(serv)$ and after executing the service's body, the handler writes the return value on channel $Fin(serv)$. We assume the variables *param* and *res* to be available in the state of every DSC, the variable *param* can be used by the program to access the parameter and *res* to write the return value.

Definition 2.1. The handler $handler_{serv}$ of a service *serv* is defined as

```
$handler_{serv} :=$ read($param \leftarrow Ini(serv)$); $body_{serv}$
                     write($res \rightarrow Fin(serv)$);
```

Example 2.4. The handler for the service `pay` looks as follows:

```
$handler_{pay} :=$
    read($param \leftarrow Ini(pay)$);
    $body_{pay}$
    write($res \rightarrow Fin(pay)$);
```

A service $serv_1$ *requires* a service $serv_2$, if a call to $serv_2$ is contained in the body of $serv_1$. We denote the set of services required by $serv_1$ with req_{serv_1}. A DSC c *provides* a set of services $prov_c$ to its environment. We recursively define the body of the DSC, referred to as $body_c$, with respect to the services it provides.

The DSC initially provides all services, while the environment chooses which service should be executed by sending a message to the respective initial channel. After termination of the called service, again, the environment can choose among all provided services. We formally define the body of a DSC as follows:

Definition 2.2. Let $\{serv_1, \ldots, serv_n\} = prov_c$ for some DSC c. Then the body of c is recursively defined as
$body_c := (handler_{serv_1} \sqcap \ldots \sqcap handler_{serv_n}); body_c$.

The *external choice* operator $\sqcap$ connects two programs waiting for a message on different channels. If a message is received on one of the channels, the respective program is executed and the other program is dismissed. Definition 2.2 enforces that DSCs are not re-entrant, since a message starting a service call can not be accepted by the DSC until a previously called service has terminated. The formal semantics of $\sqcap$ is shown in Figure 2.1.

Example 2.5. The required services for the `Cart` DSC[1]:
$req_{buy} = req_{checkCart} = req_{clearCart} = req_{getAllNums} = \emptyset$
$req_{pay} = \{registerSale\}$

The body of `Cart` is defined as follows:
$body_{Cart} :=$
$(handler_{buy} \sqcap handler_{checkCart} \sqcap handler_{clearCart} \sqcap$
$handler_{getAllNums} \sqcap handler_{pay}); body_{Cart}$

We assume every DSC c to have some unique initial state σ_c without explicitly specifying it. $\langle body_c; \sigma_c \rangle$ represents an LTS as defined in Section 2.2. We refer to this LTS by c_{LTS}.

Definition 2.2 implicitly states that Services provided by one DSC share a common state, i.e. information received by one service might be leaked by another, subsequently executed, service. Further, it stated that services are executed sequentially and the entire component halts while waiting for the termination of a called service.

Example 2.6 (Trace Communication for `Cart`)**.** For our running example, $\langle body_{Cart}; \sigma_{Cart} \rangle$ defines the LTS describing the behavior of the DSC. By

[1]The full implementation of the `Cart` DSC can be found in Section A.1

calls to the services `buy`, `checkCart`, `pay`, the LTS produces the following trace t:

$$\begin{aligned} t = \langle &\mathit{Ini}(\mathit{buy})?(1,2,3), \mathit{Fin}(\mathit{buy})!(2),\\ &\mathit{Ini}(\mathit{checkCart})?(0), \mathit{Fin}(\mathit{checkCart})!(1,2,3),\\ &\mathit{Ini}(\mathit{pay})?(12345678, 123),\\ &\quad \mathit{Ini}(\mathit{registerSale})!(1,2,3,123456789), \mathit{Fin}(\mathit{registerSale})?(1),\\ &\mathit{Fin}(\mathit{pay})!(1)\rangle \end{aligned}$$

The services `buy` and `checkCart` terminate without intermediate calls of other services, while between the initial and terminating messages for `pay`, the service `registerSale` is called by the DSC. The construction of the body of the `Cart` DSC ensures that the LTS is sequential, i.e. it can not accept calls to other services while a service is still executing. As a result, the following trace is not consistent with the LTS:

$$\begin{aligned} t_{not} = \langle &\mathit{Ini}(\mathit{buy})?(1,2,3), \mathit{Ini}(\mathit{checkCart})?(0),\\ &\mathit{Fin}(\mathit{buy})!(2), \mathit{Fin}(\mathit{checkCart})!(1,2,3)\rangle \end{aligned}$$

Also by construction, DSCs are not re-entrant, i.e. services can not be called while another service waits for the termination of a called service. Therefore, the following trace also can not be communicated by the LTS:

$$\begin{aligned} t_{not2} = \langle &\mathit{Ini}(\mathit{pay})?(12345678, 123), \mathit{Ini}(\mathit{registerSale})!(1,2,3,123456789),\\ &\mathit{Ini}(\mathit{buy})?(1,2,3), \mathit{Fin}(\mathit{buy})!(2),\\ &\mathit{Fin}(\mathit{registerSale})?(1), \mathit{Fin}(\mathit{pay})!(1)\rangle \end{aligned}$$

The set of services that a DSC requires is the union of the required services of the provided services, i.e. $\mathit{req}_c = \{\mathit{serv} \mid \exists s \cdot s \in \mathit{prov}_c \wedge \mathit{serv} \in \mathit{req}_s\}$.

Example 2.7 (Required Services per DSC)**.** For the DSCs in the running example[2], the set of required service for `Cart` is $\mathit{req}_{\mathit{Cart}} = \{\mathit{registerSale}\}$.

It is common in component-based system-engineering to assume a contract between a component and its (unspecified) environment. A part of this contract states that a component guarantees correct functionality of provided services, if the environment guarantees that required services are available and behave correctly. We do not discuss here in detail, what *correct* means, but limit the consideration to the property that a component can assume that all required services terminate. Since DSCs are designed as units for composition, a DSC may also have to serve as part of the environment of other DSCs. Therefore, we have to ensure that services provided by a DSC terminate. A service is a terminating service, if for every trace the service can

[2]See Section A.1 for the full implementation of all DSCs.

communicate, there exists a trace which leads to termination of the service. Components, and therefore also DSCs, are meant to be usable as black boxes, termination should not depend on unknown behavior of the environment. We therefore additionally require services to terminate independently from intermediate input from the environment.

Definition 2.3 (Terminating Service)**.** A service *serv* is *terminating* if

$$\forall t, \sigma \cdot \langle \mathit{handler}_{\mathit{serv}}; \sigma\rangle \xrightarrow{t} \implies \exists t' \langle \mathit{handler}_{\mathit{serv}}; \sigma\rangle \xrightarrow{t \frown t'} \langle \mathit{SKIP}; \sigma'\rangle$$
$$\text{and } \forall \sigma \exists n \in \mathbb{N}\ \ \forall t \cdot \langle \mathit{handler}_{\mathit{serv}}; \sigma\rangle \xrightarrow{t} \implies |t| \leq n$$

The first condition in Definition 2.3 ensures that every input provided by the environment leads to a state in which the service still can run to completion. The second condition ensures that the maximum length of a trace needed for completion only depends on the initial state, but not on intermediate input. It is possible to implement services which comply with the first condition of this definition, but can still be forced by the environment to perform infinite execution. Take for example the following program:

```
x = otherServ(0);
while x == 1 do {x = otherServ(0)};
return 1;
```

For every trace the program can communicate, there exists the case, when the environment provides 0 as a return value of *otherServ* which leads to infinite loop iterations. Therefore, we additionally require an upper bound for the length of a trace a service can communicate, which is ensured by the second condition in Definition 2.3. In combination the two conditions guarantee that the implementation of the service ensures that it terminates, independent from the environment the DSC runs in.

Example 2.8. The service `pay` is terminating according to Definition 2.3, since it calls the required service `registerSale` exactly one time and does not contain a loop in the body.

For technical reasons, we assume that a service is at most provided by one DSC, i.e. $\mathit{serv} \in \mathit{prov}_c \wedge \mathit{serv} \in \mathit{prov}_d \implies c = d$. This restriction is useful in the further presentation, but limits the expressivity of our language only marginally. If two DSC are designed to provide the same service (i.e. with the same name and the same parameter declaration), one of them can be changed by a simple renaming of the initial and terminating channels and the name of the service. It does however imply a static system in the sense that we do not allow exchanging DSCs at run-time.

$$\textsc{ParSynch1}\ \frac{p \xrightarrow{\alpha.v} p' \wedge \alpha \notin C}{p[\![C]\!]s \xrightarrow{\alpha.v} p'[\![C]\!]s} \qquad \textsc{ParSynch2}\ \frac{s \xrightarrow{\alpha.v} s' \wedge \alpha \notin C}{p[\![C]\!]s \xrightarrow{\alpha.v} p[\![C]\!]s'}$$

$$\textsc{ParSynch3}\ \frac{p \xrightarrow{\alpha!v} p' \wedge s \xrightarrow{\alpha?v} s' \wedge \alpha \in C}{p[\![C]\!]s \xrightarrow{\alpha!v} p'[\![C]\!]s'}$$

$$\textsc{ParSynch4}\ \frac{p \xrightarrow{\alpha?v} p' \wedge s \xrightarrow{\alpha!v} s' \wedge \alpha \in C}{p[\![C]\!]s \xrightarrow{\alpha!v} p'[\![C]\!]s'}$$

Figure 2.2: Inference rules for synchronized parallel composition

2.4 Composition

To compose DSCs, we define synchronized parallel composition for LTS. Composition for two LTS p and p' on a set of channels C, written $p[\![C]\!]p'$, means that communication between p and p' on some channel from C is performed by the DSCs directly without utilizing the environment. The semantics of $p[\![C]\!]p'$ is defined in Figure 2.2. If two LTS synchronize on a channel, progress can only happen if one LTS sends a message on the respective channel while the other LTS waits for a message on this channel. Internal communication of two composed LTS becomes an output of the composition such that the inter-DSC communication is observable for an environment, but the environment can not provide inputs on these channels.

We call the combination of two DSCs using synchronized parallel composition a *composition*. Composed DSCs communicate on a set of services by synchronizing on the respective initial and terminating channels. We provide a formal definition for compositions:

Definition 2.4 (Composition)**.** A DSC is a *composition*. For DSCs (or compositions) c, c' with

1. $(prov_c \cap prov_{c'}) = \emptyset$,
2. $(req_c \cap req_{c'}) = \emptyset$, and
3. $S \subseteq (req_c \cap prov_{c'})$

the composition $d = c[\![S]\!]c'$ is defined as

1. $prov_d := prov_c \cup (prov_{c'} \setminus S)$
2. $req_d := (req_c \setminus S) \cup req_{c'}$
3. $d_{LTS} := c_{LTS}[\![Ini(S) \cup Fin(S)]\!]c'_{LTS}$

To compose two compositions on a set of services, one composition has to provide the services, while the other composition has to require them. The set of provided services of the composition results from the provided services of each DSC, minus the services on which the DSCs synchronize on. Also, the set of required services is the combination of the services required by each DSC, except the services provided internally. The LTS defined by the composition results from the parallel synchronous composition of the two LTS of the DSCs.

We ensure that the channels used for synchronization represent calls and termination of required services for one DSC and provided services for the other DSC. This way, we enforce an acyclic structure in compositions, which guarantees that there are no deadlock situations caused by call-backs of the composed DSCs.

Composition removes services from the set of required services. Since requiring a service from the environment states an assumption made on the environment, composition can reduce the assumptions made about the environment that the composition runs in.

Example 2.9 (Composition of `Cart` and `Controlling` DSC)**.** The DSC `Controlling` requires the service `getAllNums`, which is provided by the `Cart` DSC. Further, `Controlling` provides the services `getBuys`, `getPays`, and `getChecks`, which serve as an interface for the Controlling department to gain information on how customers use the webshop. By composition of `Cart` and `Controlling` the two DSCs are synchronized on the initial and terminating channels for `getAllNums`.

$$\begin{aligned}
&\mathit{CartControlling} &&:= \mathit{Cart}[\![\{\mathit{getAllNums}\}]\!]\mathit{Controlling}\\
&\mathit{prov}_{\mathit{CartControlling}} &&:= \{\mathit{buy}, \mathit{pay}, \mathit{clearCart}, \mathit{checkCart}, \mathit{getBuys},\\
& && \quad \mathit{getPays}, \mathit{getChecks}\}\\
&\mathit{req}_{\mathit{CartControlling}} &&:= \{\mathit{registerSale}\}\\
&\mathit{CartControlling}_{LTS} &&:=\\
&\quad \mathit{Cart}_{LTS}[\![\{\mathit{Ini}(getAllNums), \mathit{Fin}(getAllNums)\}]\!]\mathit{Controlling}_{LTS}
\end{aligned}$$

The composition of the `CartControlling` can be composed with other DSCs in order to gain full webshop functionality accordingly.

In a composition, provided services are also removed from the set of provided services, which has the effect that at most one DSC can call a particular service provided by a DSC. Practically, one may want to make a service usable by several DSCs. This is a mere technicality, since we can always add a copy of the respective services with renamed initial and terminating channels to the DSC.

2.5 Conclusion

We introduced in this chapter the computational framework based on labeled transition systems which we use throughout this first part of this thesis. We further provided a formal definition of DSCs, services, compositions and how DSCs and compositions interact with each other.

During formalization, we made several limitations to what constitutes a DSC. By limiting DSCs to entities which do not share a state, we made the decision that for example objects in object-oriented programming languages or program libraries are in general not considered to be DSCs. Since we aim for security properties in distributed systems in this thesis, this limitation is fair. We also limited DSCs to entities which do not allow parallel execution of services. This restriction is common in programming frameworks, as we will discuss in detail in in Chapter 8 with the example of the Java Enterprise Edition.

Further, we made the limitation that a service provided by a DSC may at most be called by one other DSC in a composition. Also, different DSCs must not provide the same service. For both limitations, we stated that they are only technical, but do not limit our framework substantially. For the remainder of the first part, we ask the reader to accept this statement. In the second part of this thesis, when we apply our framework in a practical setting, it will become clear why these limitations do not have a practical impact.

3 Non-Interference in Distributed Service Components

3.1 Introduction

In this chapter we define a non-interference property specially designed for DSCs and show that non-interference for DSCs is compositional. We further provide a non-interference property for services and show that a DSC is non-interferent if all services are.

Non-interference is a program property which describes that sensitive inputs do not influence outputs which are specified not to contain sensitive information. A non-interference specification separates inputs and outputs of a program into sensitive (*high*) and non-sensitive (*low*) inputs and outputs. A program is non-interferent w.r.t. a specification, if the low output is not influenced by the high input.

Existing notions for non-interference in literature differ in the expressiveness, i.e. what a specification can state to be high or low information; in the precision, i.e. whether the non-interference property categorizes intuitively non-interferent programs as such; if non-interferent programs can be composed and result in non-interferent programs; and in the possibilities for analyzing programs for whether they are non-interferent. For an extensive review of non-interference properties in the literature, we refer the reader to related work described in Chapter 5. An additional, less technical, criterion for non-interference notions is how intuitive it is, i.e. how easy is it for a human to understand what it means that a program is non-interferent.

In this chapter, we define a non-interference property which is especially designed for DSCs. By limiting the considered programs to DSCs, we gain a very precise notion of non-interference, which also allows expressive specifications for high and low inputs and outputs. Our specifications allows to specify parts of inputs and outputs and even functions over inputs and outputs to be classified low, as well as the specification of the existence of inputs and outputs, potentially depending on their parameters.

Our non-interference notion is based on an explicit model of the environment of a program which allows an intuitive understanding of how non-interference of a DSC w.r.t. a specification relates to attacker models for a concrete program. Further, we show that non-interference for DSCs is compositional, an important property especially in the context of component-based systems.

We do not provide an explicit method for analysis of DSCs in this chapter, however, we do provide a non-interference property for services, which is similar to existing notions of non-interference for which there are analysis methods available. This way, we provide a basis for tool developers who want to design analysis methods for non-interference for DSCs.

In the next section, we introduce our specification language which is based on equivalence relations and introduce *strategies* as an explicit model for the environment of a DSC. Based on the specification language and the environment model, we define non-interference for LTS in general and show that non-interference for LTS is compositional for asynchronous parallel composition of LTS. In Section 3.3 we specialize the non-interference notion for DSCs and show that non-interference is also compositional for DSCs and compositions. In Section 3.4 we define non-interference for services and show that a DSC is non-interferent, if all services provided by the DSC are non-interferent. Finally, we conclude the chapter.

The results in this chapter are based on work previously published by the author (Greiner and Grahl [2016]).

3.2 Non-Interference with What-Declassification

To analyze information flow for an LTS, a specification of high and low information is necessary. Typically, this specification is, depending on the framework, given in the form of types of variables, parameters or channels.

Specification of high and low information for LTS with types is coarse-grained, since it does not allow to specify that only partial information contained in an input may be public (See 'What'-declassification by Sabelfeld and Sands [2009]). For example, it cannot be expressed that at most the last four digits of a credit card number may be revealed. In our case, this even does not allow different levels of confidentiality for two parameters of one and the same service call, since by construction described in the previous chapter, messages only provide one parameter (which may encode several parameters). Please note that this coarse grained specification is not inherent to type systems, since work on non-interference for batch programs allows more precise declassification of information using type systems. See Section 5.2 for a discussion.

Therefore, we introduce a specification of high and low information based on equivalence relations. If two messages are equivalent with respect to this specification, the observable behavior of the LTS should be equivalent for an adversary. This is a generalization of specification using types, allows a flexible specification of secret information, and we can express type-based specifications with our relations.

For a compact presentation, we only consider the 2-element security lattice consisting of *high* and *low* in this chapter. Nevertheless, a more complicated security lattice can easily be expressed with our notion of non-interference, but explicit consideration does not provide further insights.

3.2.1 Security Specification of Messages and Values

We assume the classification of high and low input and output for an LTS is provided by an equivalence relation $\sim \subseteq \mathbb{M} \times \mathbb{M}$ over messages as part of the specification. If two messages m_1, m_2 are equivalent with respect to $\sim$, the information that discriminates m_1 from m_2 is secret.

In order to specify that the existence of a message itself is a secret, we introduce a special placeholder $\square \in \mathbb{M}$. We call a message $m \sim \square$ *invisible*, and *visible* if $m \nsim \square$. We denote the set of all visible messages with $\blacksquare = \{m \in \mathbb{M} \mid m \nsim \square\}$ If a message m is invisible, the observable behavior of an LTS must not differ depending on whether m is provided as an input or not.

Example 3.1. We illustrate the specification using equivalence relations for the `Cart` DSC as introduced in the previous chapter. For the moment we assume a potential attacker modeling an employee in the billing department. In order to perform his job, he has to know about the products a customer ordered, at which price he bought it, and how many items he ordered. We use equivalence relations to express this information in input- and output-level.

$$\begin{aligned}\mathit{Ini}(\mathit{buy}).(\mathit{pid}, \mathit{pr}, \mathit{am}) \sim \mathit{Ini}(\mathit{buy}).(\mathit{pid}', \mathit{pr}', \mathit{am}') \ :\Leftrightarrow& \\ \mathit{pid} = \mathit{pid}' \wedge \mathit{pr} = \mathit{pr}' \wedge \mathit{am} = \mathit{am}'&\end{aligned}$$

It is, however, irrelevant for the potential attacker, whether or not the customer checked the content of his shopping cart.

$$\mathit{Ini}(\mathit{checkCart}).v \sim \square \ :\Leftrightarrow \ \mathit{true}$$

We can also specify partial information to be low, for example to state that the attacker may know the last four digits of the credit card number, as he should print this information on the bill. The pin number, however is not relevant.

$$\begin{aligned}\mathit{Ini}(\mathit{pay}).(\mathit{ccnr}, \mathit{pin}) \sim \mathit{Ini}(\mathit{pay}).(\mathit{ccnr}', \mathit{pin}') \ :\Leftrightarrow& \\ \mathit{ccnr}\%10000 = \mathit{ccnr}'\%10000&\end{aligned}$$

We assume in the remainder of this chapter some definition of $\sim$ to be given. We also assume that an attacker is always able to distinguish on which channel a visible message was communicated. Formally, this means:

$$m \sim m' \implies (m \sim \Box \wedge m' \sim \Box) \vee (m = \alpha.v \wedge m' = \alpha.v') \text{ for some } \alpha \in \mathbb{C}.$$

The equivalence relation $\sim$ implicitly defines *equivalence classes* on $\mathbb{M}$ with

$$[m] := \{m' \mid m' \sim m\}.$$

For every equivalence class $[m]$, we denote an arbitrary, but constant *representative* $[\![m]\!] \in [m]$, where $[\![\Box]\!] = \Box$.

Equivalence of messages gives rise to the equivalence of traces t, t', written $t \sim t'$. Traces t, t' are equivalent, if, after removing invisible messages, their *projection* on the representative of the equivalence classes are equal.

Definition 3.1. We define $\sim \subseteq \mathbb{T} \times \mathbb{T}$ with $t \sim t'$ if $t{\restriction_\sim} = t'{\restriction_\sim}$ where

$$\begin{aligned} \langle\rangle{\restriction_\sim} &:= \langle\rangle \\ (m \frown t){\restriction_\sim} &:= \begin{cases} t{\restriction_\sim} & \text{if } m \sim \Box \\ [\![m]\!] \frown t{\restriction_\sim} & \text{otherwise} \end{cases} \end{aligned}$$

While we introduced $\sim$ for equivalence of messages, we overload the symbol for equivalence of traces in order to avoid many different symbols. It should be clear from the context, whether $\sim$ refers to messages, traces, or sets of messages as defined below.

Example 3.2. Assume the following traces in our example:

$$\begin{aligned} t =& \langle \mathit{Ini}(\mathit{buy}).(1,2,3),\quad \mathit{Ini}(\mathit{checkCart}).(0),\quad \mathit{Ini}(\mathit{pay}).(12345678, 123)\rangle \\ t' =& \langle \mathit{Ini}(\mathit{buy}).(1,2,3),\quad \mathit{Ini}(\mathit{checkCart}).(0),\quad \mathit{Ini}(\mathit{checkCart}).(1), \\ & \quad \mathit{Ini}(\mathit{pay}).(99995678, 666)\rangle \end{aligned}$$

The projection of the two traces then provides us according to the specification in Example 3.1 with the following traces, depending on the choice of the representative of the equivalence classes:

$$\begin{aligned} t{\restriction_\sim} &= \langle \mathit{Ini}(\mathit{buy}).(1,2,3),\quad \mathit{Ini}(\mathit{pay}).(00005678, 000)\rangle \\ t'{\restriction_\sim} &= \langle \mathit{Ini}(\mathit{buy}).(1,2,3),\quad \mathit{Ini}(\mathit{pay}).(00005678, 000)\rangle \end{aligned}$$

Since $t{\restriction_\sim} = t'{\restriction_\sim}$ the original traces t and t' are equivalent w.r.t. $\sim$.

Apart from the projection operator $\cdot{\restriction_.}$ on traces, we define projection on sets and a *filter* operator $\rhd$ on traces and sets. These definitions will be useful in the remainder for formalization and the presentation of proofs.

Definition 3.2. Let $M, N \subseteq \mathbb{M}, m \in \mathbb{M}, t \in \mathbb{T}$.

- $M\restriction_{\sim} := \{[\![m]\!] \mid m \in M\} \setminus [\square]$
- $M \sim N :\Leftrightarrow M\restriction_{\sim} = N\restriction_{\sim}$
- $M \rhd N := M \cap N$
- $\langle\rangle \rhd N := \langle\rangle$
- $(m \frown t) \rhd N := \begin{cases} m \frown (t \rhd N) & \text{if } m \in N \\ t \rhd N & \text{otherwise} \end{cases}$

If two traces are equivalent, then the amount of visible messages in both traces is equal.

Lemma 3.1. $\forall t \sim t' \cdot |t \rhd \blacksquare| = |t' \rhd \blacksquare|$

Proof. Follows from definition of equivalence of traces and the definition of the filter operation for visible messages. $\lhd$

Also, an LTS is input neutral for equivalent and visible messages, i.e. if an LTS p accepts an input m and $m \sim m'$ and m and m' are visible, then p also accepts m'.

Lemma 3.2. *Given LTS p and $m, m' \in \mathbb{I}$ with $m \nsim \square$ and $m' \sim m$. Then* $p \xrightarrow{m} \implies p \xrightarrow{m'}$.

Proof for Lemma 3.2. Follows directly from definition of LTS and the restrictions on $\sim$ stating that m and m' have to be messages over the same channel. $\lhd$

As a shortcut, we define *prefix equivalence*, written $s \lesssim t$, as $\exists t_1 \leq t \cdot s \sim t_1$.

3.2.2 Strategies

The environment observes the trace an LTS communicates and provides input depending on this observation. The environment may also deny to provide further input. We model the environment as a *strategy*, a function mapping the previously communicated trace, i.e. the observation made by the environment, to a set of possible inputs provided by the environment.

Our goal is to define non-interference for LTS. Therefore, we want to ensure that detected leaks are due to an insecurity in the LTS not due to an environment leaking confidential information. So, we require the strategy to provide equivalent input for equivalent observations. We denote the set of all strategies by `Strat`.

Definition 3.3. A *strategy* is a function $\omega : \mathbb{T} \mapsto \mathcal{P}(\mathbb{I})$, such that for all $t_1, t_2 \in \mathbb{T} \cdot t_1 \sim t_2 \implies \omega(t_1) \sim \omega(t_2)$.

A trace t is *consistent* with a strategy ω, written $\omega \models t$, if all inputs in the trace are provided by the strategy. Formally $\omega \models t$, if for all $m \in \mathbb{I}$ with $t = t_1 \frown m \frown t_2$ for some t_1 and t_2, it holds that $m \in \omega(t_1)$. An LTS p produces or communicates t under ω, written $\omega \models p \xrightarrow{t}$ if ω is consistent with t and p can communicate t, formally $p \xrightarrow{t}$.

Example 3.3. We reconsider the trace of the DSC `Cart` from Example 2.6:

$$\begin{aligned} t = \langle & \mathit{Ini}(\mathit{buy})?(1,2,3),\ \ \mathit{Fin}(\mathit{buy})!(2),\ \ \mathit{Ini}(\mathit{checkCart})?(0),\\ & \mathit{Fin}(\mathit{checkCart})!(1,2,3),\ \ \mathit{Ini}(\mathit{pay})?(12345678,123),\\ & \mathit{Ini}(\mathit{registerSale})!(1,2,3,123456789),\ \ \mathit{Fin}(\mathit{registerSale})?(1),\\ & \mathit{Fin}(\mathit{pay})!(1)\rangle \end{aligned}$$

We have illustrated in Example 2.6 that this trace can be communicated by the LTS for `Cart`. The trace is also consistent with a strategy ω, if it provides the inputs in the trace, i.e. :

$$\begin{aligned} &\mathit{Ini}(\mathit{buy})?(1,2,3) && \in \omega(\langle\rangle) \wedge \\ &\mathit{Ini}(\mathit{checkCart})?(0) && \in \omega(\langle \mathit{Ini}(\mathit{buy})?(1,2,3),\ \ \mathit{Fin}(\mathit{buy})!(2)\rangle) \wedge \\ &\mathit{Ini}(\mathit{pay})?(12345678,123) && \in \omega(\langle \mathit{Ini}(\mathit{buy})?(1,2,3),\ \ \mathit{Fin}(\mathit{buy})!(2),\\ &&& \quad \mathit{Ini}(\mathit{checkCart})?(0),\\ &&& \quad \mathit{Fin}(\mathit{checkCart})!(1,2,3)\rangle) \wedge \\ &\mathit{Fin}(\mathit{registerSale})?(1) && \in \omega(\langle \mathit{Ini}(\mathit{buy})?(1,2,3),\ \ \mathit{Fin}(\mathit{buy})!(2),\\ &&& \quad \mathit{Ini}(\mathit{checkCart})?(0),\\ &&& \quad \mathit{Fin}(\mathit{checkCart})!(1,2,3),\\ &&& \quad \mathit{Ini}(\mathit{pay})?(12345678,123),\\ &&& \quad \mathit{Ini}(\mathit{registerSale})!(1,2,3,123456789)\rangle) \end{aligned}$$

If a strategy ω provides at most the input another strategy ω' provides, we say that ω *refines* ω'.

Definition 3.4 (Strategy Refinement)**.** ω *refines* ω', written $\omega \leq \omega'$, if $\omega(t) \subseteq \omega'(t)$ for all t.

A strategy ω refining ω' is at most consistent with the traces that ω' is consistent with.

Lemma 3.3. *If $\omega \leq \omega'$ then for all LTS p: $\omega \models p \xrightarrow{t} \implies \omega' \models p \xrightarrow{t}$ for all $t \in \mathbb{T}$.*

Proof for Lemma 3.3. Proof according to Clark and Hunt [2009]. In general, every trace accepted by ω is also accepted by ω', due to definition of strategy acceptance and refinement relation. $\omega \models t$, so for every $m \in \mathbb{I}$: $t' \frown m \leq t \implies m \in \omega(t')$ and since $\omega(t') \subseteq \omega'(t')$ it holds that $m \in \omega'(t')$, so $\omega' \models t$. $\lhd$

Non-interference informally states that given equivalent inputs for two runs of a program, both runs provide equivalent outputs. Since in our case, the output depends on intermediate input, which again depends on previous outputs, we consider two runs in equivalent environments instead of equivalent inputs. Two strategies are equivalent with respect to $\sim$, if they provide equivalent input after the same observation.

Definition 3.5 (Equivalence of Strategies)**.** Two strategies ω and ω' are *equivalent* with respect to an equivalence relation $\sim \subseteq \mathbb{M} \times \mathbb{M}$ if $\forall t \in \mathbb{T} \cdot \omega(t)\lceil_\sim = \omega'(t)\lceil_\sim$.

Again, we overload the symbol $\sim$ and write $\omega \sim \omega'$ for strategies that are equivalent with respect to $\sim$.

The following lemma states an equivalent formalization for strategy equivalence. In the remainder of this chapter we will use different either formalization in proofs, depending on which one is the most useful. Especially Lemma 3.4, 3 will become important in the next section, since it provides an intuitive understanding how environments and attacker are related.

Lemma 3.4. *Given $\omega, \omega' \in \mathtt{Strat}$. Then, the following notions are equivalent:*

1. $\omega \sim \omega'$
2. $\forall m \nsim \square\ \forall t \cdot \omega(t) \rhd [m] = \emptyset \Leftrightarrow \omega'(t) \rhd [m] = \emptyset$
3. $\forall t, t' \cdot t \sim t' \implies \omega(t) \sim \omega'(t')$

Proof for Lemma 3.4.
Ad 1 $\Leftrightarrow$ 2: Assume $\omega \sim \omega'$. Then, by Definition 3.5, $\forall t \cdot \omega(t) \sim \omega'(t)$, which is by definition of low-projection $\forall t \cdot \{[m] \mid \exists n \sim m \in \omega(t) \wedge m \notin [\square]\} = \{[m] \mid \exists n \sim m \in \omega'(t) \wedge m \notin [\square]\}$, which is equal to $\forall t, [m] \neq [\square] \cdot \omega(t) \rhd [m] = \emptyset \Leftrightarrow \omega'(t) \rhd [m]$ which is equivalent to $\forall [m], t \cdot [m] \neq [\square] \Leftrightarrow \omega(t) \rhd [m] \sim \omega'(t) \rhd [m]$.
Ad 1 $\implies$ 3: $\omega \sim \omega' \wedge t \sim t' \implies \omega(t) \sim \omega(t') \wedge \omega'(t) \sim \omega'(t')$ since ω and ω' are strategies. $\omega \sim \omega'$, therefore $\omega(t) \sim \omega'(t)$. And by transitivity of $\sim$: $\omega(t) \sim \omega'(t) \sim \omega'(t')$
Ad 3 $\implies$ 1: $\forall t \sim t' \cdot \omega(t) \sim \omega'(t')$, so especially $\forall t \cdot \omega(t) \sim \omega'(t)$ and by definition of $\sim$ on sets of messages: $\forall t \cdot \omega(t)\lceil_\sim = \omega'(t)\lceil_\sim$ which is the definition of $\sim$ on strategies. $\lhd$

3.2.3 Non-interference

To define non-interference for LTS, we compare different runs of an LTS. Since execution of an LTS in general requires intermediate input, the runs have to be executed in presence of an environment providing this input. We want to ensure that different behavior of runs indicating information leaks is due to leaks in the LTS, not by the environment leaking secrets. Therefore, we require the different runs to be executed under equivalent environments. We say an LTS is non-interferent, if for every trace which is consistent for the LTS under a strategy, for every other, equivalent strategy there exists an equivalent trace with which the LTS is also consistent.

Definition 3.6 (W-non-interference)**.** An LTS p is *W-non-interfering* for $W \subseteq \mathtt{Strat}$, if

$$\begin{aligned}&\forall \omega_1, \omega_2 \in W, \forall t_1 \cdot \omega_1 \sim \omega_2 \;\wedge\; \omega_1 \models p \xrightarrow{t_1} \implies \\ &\qquad\qquad \exists t_2 \cdot \omega_2 \models p \xrightarrow{t_2} \;\wedge\; t_1 \sim t_2\end{aligned}$$

The definition requires an LTS to run in two equivalent environments, i.e. both runs are executed with equivalent inputs. The resulting traces being equivalent expresses that for an attacker, who is only able to observe low information in a trace, can not distinguish the two runs. Therefore, the attacker can not distinguish under which environment the LTS was run. This property has to hold for all pairs of equivalent environments, meaning that seeing the low part of a trace, any environment could have produced the respective trace from the point of view of the attacker.

A *W-attack* is a counter example for W-non-interference.

Definition 3.7 (Attack)**.** A *W-attack* on p is a tuple $(\omega_1, \omega_2, t_1) \in W \times W \times \mathbb{T}$ with

1. $\omega_1 \sim \omega_2$ and
2. $\omega_1 \models t_1$ and
3. $\omega_1 \models p \xrightarrow{t_1}$ and
4. $\forall t_2 \cdot \omega_2 \models p \xrightarrow{t_2} \implies (t_1 \nsim t_2)$

It is easy to see that an LTS is W-non-interferent if and only if there does not exist a W-attack. We denote the set of all LTS, which are W-non-interferent with W-NI. If an LTS is W-non-interferent, it is also non-interferent with respect to all subsets of W.

Lemma 3.5. *For all $W_1, W_2 \subseteq \mathtt{Strat}$: $W_1 \subseteq W_2 \implies W_2$-NI $\subseteq W_1$-NI.*

Proof for Lemma 3.5. We have to prove that, given $W_1 \subseteq W_2$, all $p \in W_2$-NI it also holds that $p \in W_1$-NI. We show the contrapositive. Assume $p \notin W_1$-NI. Then there exists (ω_1, ω_2, t) which is a W_1-attack on p. Since $W_1 \subseteq W_2$, $\omega_1 \in W_2$ and $\omega_2 \in W_2$, it follows that (ω_1, ω_2, t) is a W_2-attack on p. ◁

Related work by Rafnsson et al. [2012] provides a non-interference notion similar to ours, however without declassification (See related work for details). They show compositionality of their non-interference notion under parallel interleaving, which models messages passing performed by the environment. While their proof can not easily be transferred to our case with declassification, we still show this kind of compositionality of our non-interference notion with declassification for better comparability of the two non-interference notions.

Theorem 3.1 (Compositionality). *$p_A, p_B \in \texttt{Strat}\text{-}NI \implies (p_A \parallel p_B) \in \texttt{Strat}\text{-}NI$, where $\parallel$ denotes asynchronous parallel composition.*

For the following proof, we need a supporting lemma. Non-interference of an LTS does not depend on feeding of invisible messages. If we can find an attack on an LTS, then there also exists an attack with a strategy that does not provide any secret input.

Lemma 3.6. *If (ω_1, ω_2, t) is an attack on p, then, for ω_2' with $\forall t \cdot \omega_2'(t) = \omega_2(t) \setminus [\Box]$, also (ω_1, ω_2', t) is an attack on p.*

Proof for Lemma 3.6. ω_2' refines by construction ω_2 according to Definition 3.4. By Lemma 3.3, this means $\forall t' \cdot \omega_2' \models t' \implies \omega_2 \models t'$. Therefore, if there would exist a trace t with $\omega_2' \models p \xrightarrow{t}$, then also $\omega_2 \models p \xrightarrow{t}$ would be true, which contradicts the original assumption of the lemma. ◁

Proof for Theorem 3.1. We prove that $(p_A \parallel p_B) \notin \texttt{Strat}\text{-NI} \implies p_A, p_B \notin \texttt{Strat}\text{-NI}$.

Since $(p_A \parallel p_B) \notin \texttt{Strat}$-NI we know that there exists an attack (ω_1, ω_2, t) on $p_A \parallel p_B$. In particular, we know by Lemma 3.6 that ω_2 does not produce invisible input.

Assume towards contradiction $p_A, p_B \in \texttt{Strat}$-NI. Then by definition we have for $k \in \{A, B\}$: $\forall \omega_{1k}, \omega_{2k} \in \texttt{Strat} \cdot \omega_{1k} \sim \omega_{2k} \implies \forall t_{1k} \cdot \omega_{1k} \models p_k \xrightarrow{t_{1k}} \implies \exists t_{2k} \cdot \omega_{2k} \models p_k \xrightarrow{t_{2k}} \wedge t_{1k} \sim t_{2_k}$

Select t_{1A}, t_{1B} such that t is an interleaving of t_{1A} and t_{1B} and $p_A \xrightarrow{t_{1A}}$ and $s_B \xrightarrow{t_{1B}}$. We now construct strategies $\omega_{1A}, \omega_{2A}, \omega_{1B}, \omega_{2B}$ such that $\omega_{1A} \sim \omega_{2A}$ and $\omega_{1B} \sim \omega_{2B}$.

We use the notion $t_1 \parallel_t t_2$ to denote that t is an interleaving of t_1 and t_2.

Let $j \in \{1, 2\}, k, k' \in \{A, B\}, k \neq k'$.

$$\begin{aligned}\omega_{jk}(t) := \{m \mid \exists t_1' t_k', t_{k'}' \cdot t \sim t_k' \wedge t_k' \parallel_{t_1'} t_{k'}' \\ \wedge t_k' \frown m \lesssim t_{1k} \wedge p_k \xrightarrow{t_k' \frown m} \\ \wedge t_{k'}' \lesssim t_{1k'} \wedge p_{k'} \xrightarrow{t_{k'}'} \\ \wedge t_1' \frown m \lesssim t_1 \wedge \omega_j \models (p_A \parallel p_B) \xrightarrow{t_1' \frown m} \}\end{aligned}$$

We have to show $\omega_{jk} \in \texttt{Strat}$, $\omega_{1k} \sim \omega_{2k}$, $\omega_{1A} \models p_A \xrightarrow{t_{1A}}$ and $\omega_{1B} \models p_B \xrightarrow{t_{1B}}$.

Then, we show that this contradicts $p_A, p_B \in \texttt{Strat}$.

Proof for $\omega_{jk} \in \texttt{Strat}$ Let $t \sim t'$. Assume $m \in \omega_{jk}(t)$. Let $t'_1, t'_k, t'_{k'}$ be the witnesses for m above. Since $t \sim t'$ and $t \sim t'_k$ also $t' \sim t'_k$. Therefore $t'_1, t'_k, t'_{k'}$ is a witness for t' and $m \in \omega_{jk}(t')$ and $\omega_{jk}(t) \sim \omega_{jk}(t')$.

Proof for $\omega_{1k} \sim \omega_{2k}$ Let t be arbitrary. Assume w.l.o.g. $m \in \omega_{1k}(t)$ and $m \nsim \square$.

Let $t'_1, t'_k t'_{k'}$ be the witness for m in the definition of ω_{1k}. Since $\omega_j \models (p_A \parallel p_B) \xrightarrow{t'_1 \frown m}$ and $\omega_1 \sim \omega_2$ there exists an $m' \in \omega_2(t'_1)$ with $m' \sim m$. By input neutrality, $(p_A \parallel p_B) \xrightarrow{t'_1 \frown m'}$. Since $t'_k \frown m \sim t'_k \frown m'$, also $t'_k \frown m' \lesssim t_{1k}$. Similar since $t'_1 \frown m \sim t'_1 \frown m'$, also $t'_1 \frown m' \lesssim t_1$. Lines 1 and 3 from the definition are independent from parameter j, therefore $m' \in \omega_{2k}(t)$.

Proof for $\omega_{1A} \models p_A \xrightarrow{t_{1A}}$ We show that $\omega_{1A} \models p_A \xrightarrow{t_{1A}}$ We already have $p_A \xrightarrow{t_{1A}}$, $p_B \xrightarrow{t_{1B}}$, $t_{1A} \parallel_{t_1} t_{1B}$, and $\omega_1 \models (p_A \parallel p_B) \xrightarrow{t_1}$.

We show $\omega_{1A} \models p_A \xrightarrow{t_{1A}}$ by induction over $n = |t_{1A} \triangleright \mathbb{I}|$

Base case ($n = 0$): Since $p_A \xrightarrow{t_{1A}}$ and t_{1A} has no inputs, trivially it holds $\omega_{1A} \models p_A \xrightarrow{t_{1A}}$.

Step case ($|t_{1A} \triangleright \mathbb{I}| = n + 1$): Assume by induction $\omega_{1A} \models p_A \xrightarrow{t'_{1A}}$ for all t'_{1A} with $|t'_{1A} \triangleright \mathbb{I}| = n$. We know $t_{1A} = t'_{1A} \frown m \frown t''_{1A}$ for some t''_{1A} with $|t''_{1A} \triangleright \mathbb{I}| = 0$ and some $m \in \mathbb{I}$.

For some $t'_{1B} \leq t_{1B}$ and $t'_1 \leq t_1$ we have $t'_{1A} \parallel_{t'_1} t'_{2B}$ and $\omega_{1A} \models p_A \parallel p_B \xrightarrow{t'_1 \frown m}$. By $p_A \xrightarrow{t_{1A}}$ and $p_B \xrightarrow{t_{1B}}$, we get $p_A \xrightarrow{t'_{1A} \frown m}$ Since $u \leq u' \implies u \lesssim u'$, we get by definition of ω_{1A}: $m \in \omega_{1A}(t'_{1A})$. Therefore $\omega_{1A} \models p_A \xrightarrow{t'_{1A} \frown m}$. And since t''_{1A} does not have inputs, it also holds that $\omega_{1A} \models p_A \xrightarrow{t'_{1A} \frown m \frown t''_{1A}}$.

The proof for $\omega_{1B} \models p_B \xrightarrow{t_{1B}}$ can be obtained by swapping A and B in the previous paragraph.

$p_A \notin$ Strat-NI or $p_B \notin$ Strat-NI We have assumed towards contradiction that $p_A, p_B \in \texttt{Strat}$-NI. Since $\omega_{1k} \sim \omega_{2k}$, there exist $t_{1k} \sim t_{2k}$ such that $\omega_{2k} \models p_k \xrightarrow{t_{2k}}$. We now show that there exists $t_2 \sim t_1$ with $\omega_2 \models (p_A \parallel s_B) \xrightarrow{t_2}$, which contradicts the original assumption that there exists an attack on $(p_A \parallel p_B)$.

We assume $|t_{2k} \rhd \mathbb{I}| > 0$. Let $t_{2k} = t'_{2k} \frown m_k \frown t''_{2k}$ with $|t''_{2k} \rhd \mathbb{I}| = 0$ and $m_k \in \mathbb{I}$. By definition of ω_{2k}, we have $m_A \in \omega_{2A}(t'_{2A})$ and $m_B \in \omega_{2B}(t'_{2B})$. Therefore, there exist t_1^A, t_1^B such that $t_1^A \frown m_A \lesssim t_1$ and $t_1^B \frown m_B \lesssim t_1$ and $\omega_2 \models p_A \parallel p_B \xrightarrow{t_1^A \frown m_A}$ and $\omega_2 \models p_A \parallel p_B \xrightarrow{t_1^B \frown m_B}$. Since $m \in \omega_2(t) \implies m \nsim \square$, we know $t_1^A \frown m_A \rhd \mathbb{I} \ \rhd \blacksquare = t_1^A \frown m_A \rhd \mathbb{I}$ and $t_1^B \frown m_B \rhd \mathbb{I} \ \rhd \blacksquare = t_1^B \frown m_B \rhd \mathbb{I}$.

By definition, we either get $t_1^A \frown m_A \lesssim t_1^B \frown m_B \lesssim t_1$ or $t_1^B \frown m_B \lesssim t_1^A \frown m_A \lesssim t_1$. W.l.o.g. $t_1^B \frown m_B \lesssim t_1^A \frown m_A \lesssim t_1$. Therefore, we get $t_1^A \frown m_A \rhd \mathbb{I} \sim t_1 \rhd \mathbb{I}$. Thus, there also exists t'_1, t''_1 such that $t'_1 \frown t''_1 = t_1$ and $t''_1 \rhd \mathbb{I} \sim \langle\rangle$ and $t'_1 \sim t_1^A \frown m_A$. Now there is some u''_1 with $u''_1 \sim t''_1$ and $t''_{2A} \parallel_{u''_1} t''_{2B}$. Since $|u''_1 \rhd \mathbb{I}| = 0$ and $\omega_2 \models (p_A \parallel p_B) \xrightarrow{t_1^A \frown m_A}$, we also get $\omega_2 \models (p_A \parallel p_B) \xrightarrow{t_1^A \frown m_A \frown u''_1}$. But $t_1^A \frown m_A \frown u''_1 \sim t_1$, which contradicts the original assumption in this proof. Thus, either $p_A \notin \texttt{Strat}$-NI or $p_B \notin \texttt{Strat}$-NI. ◁

Theorem 3.1 shows that non-interference with what-declassification is compositional for *LTS* under asynchronous parallel composition, as is non-interference for *LTS* without declassification defined by Rafnsson et al. [2012].

3.3 Cooperative Non-interference

In this section, we show how our notion of non-interference relates to DSCs. In contrast to the previous section, where we considered arbitrary LTS, we limit our focus in this section to DSCs. Composition of DSCs is done by synchronized parallel communication between DSCs and we define and assume cooperative environments. Further, we show that non-interference for DSCs is compositional.

Applying non-interference as defined in the previous section would be sound, however it would not be very precise. The non-interference notion as defined above would reject may DSCs we intuitively consider secure. We illustrate this with the following example.

Example 3.4. We revisit Example 2.3. For a reminder, this is the service's implementation.

```
read($param \leftarrow Ini(pay)$);
$payCount++$;
$registerSale(prodId, price, amount, param\#1)$;
$res = 1$;
write($res \rightarrow Fin(pay)$);
```

For this example, we consider all messages (and their contents) caused by calls and termination of `pay` to be low and messages on `registerSale` to be invisible. Intuitively, the service can not leak any information, since one output (parameters provided to `registerSale` is invisible and the other output (the return value) is constant 1. However, the service is not secure w.r.t. our non-interference definition in the previous section.

We can construct a strategy which executes the service to termination and another strategy providing the same input, except never providing a termination message for `registerSale`. The second strategy then would block execution of `pay` after the call of `registerSale`. Since this termination message is invisible, the two strategies are equivalent according to Definition 3.5 and we have found an attack.

However, since we assume for DSCs all services to terminate, the environment has to provide a termination message for every service call. So considering the program above to be insecure due to a not-provided termination message would be an over-approximation.

We stated that DSCs only guarantee correctness, if the environment is cooperative. Therefore, we also assume cooperative environments when defining non-interference for DSCs and compositions. An environment is *cooperative*, if it satisfies three conditions.

1. Every service required by a composition is provided by the environment.
2. Every service called by a composition terminates.
3. A service terminates with a visible message if and only if it was called with a visible message.

We model the environment as strategies, therefore cooperative environments can be modeled as a subset of all strategies. Condition 1 is trivially satisfied by any strategy since the call of a service is an output message sent by the composition and strategies cannot refuse outputs.

Condition 2 ensures that if a composition calls a required service, the environment provides a message on the terminating channel. This is a real restriction on strategies.

Finally Condition 3 ensures that the information whether or not a DSC called a service is not leaked by the strategy. Assume a DSC calls a service with an invisible message. If the environment answers this call with a visible message, the environment leaks the information whether the service was called. Since we do not consider leaks caused by the environment, we rule out this kind of leak by definition.

We call a strategy satisfying Conditions 2 and 3 a *cooperative* strategy and formalize this in Definition 3.8.

Definition 3.8 (Cooperative Strategies)**.** Given composition c providing the services $prov_c$, $\omega \in \texttt{Strat}$ is a *cooperative strategy* for c, written $\omega \in \texttt{Coop}_c$, if for all $t, t', serv, \sigma, v$ such that $serv \in req_c$, and $Fin(serv) \notin t'$, and $\omega \models c_{LTS} \xrightarrow{t \frown Ini(serv)!v \frown t'}$, it holds

$$\exists t'' \cdot \omega \models c_{LTS} \xrightarrow{t \frown Ini(serv)!v \frown t' \frown t''} \wedge \tag{3.1}$$
$$Fin(serv)?v \in \omega(t \frown Ini(serv)!v \frown t' \frown t'') \tag{3.2}$$

and

$$Fin(serv)?v' \in \omega(t \frown Ini(serv)!v \frown t') \implies \tag{3.3}$$
$$Ini(serv)!v \sim \Box \Leftrightarrow Fin(serv)?v' \sim \Box \tag{3.4}$$

The first restriction in Definition 3.8 formalizes Condition 2. We ensure that for every trace which is consistent with a cooperative strategy and a composition, and contains the call of a service, there also exists a trace (Line 3.1) after which the called service terminates (Line 3.2). For DSCs, the termination has to be communicated right after the call of the service, because by construction of DSCs, no other traces are accepted by the DSC. Especially, this also ensures that a strategy cannot block execution of a DSC by not providing invisible termination messages. As a consequence, it may be a secret, whether the environment calls a service, but not whether a service terminates if it was called.

The second restriction formalizes visibility preserving execution of services required by the DSC (Condition 3). It ensures that, if a cooperative strategy provides a terminating message for a trace, which contains the initial message for the service (Line 3.3), then this terminating message is visible if and only if the initial message was visible (Line 3.4). This way, we avoid that the strategy leaks the information that an invisible service call happened by revealing the call through the termination message.

The set of cooperative strategies for a DSC or composition c only limits the set of strategies in the case that the composition requires services and therefore expects a cooperative environment. Directly from this, it follows that if the composition does not require services, it is $\texttt{Strat}$-NI.

Lemma 3.7. *For a composition c with $req_c = \emptyset$ it holds that $\texttt{Coop}_c = \texttt{Strat}$ and $c_{LTS} \in \texttt{Coop}_c$-NI $\Leftrightarrow c_{LTS} \in \texttt{Strat}$-NI.*

Remember that composition as defined earlier possibly reduces the set of required services. Thus, by composition the dependency of a DSC on the cooperation of the environment can be reduced and finally removed.

The definition of non-interference for DSCs and compositions now is straight-forward. If the composition does not leak any information in the presence of cooperative environments, it is non-interferent.

Definition 3.9 (Cooperation-non-Interference)**.** A composition c is *non-interferent*, if $c_{LTS} \in \mathtt{Coop}_c$-NI.

When composing two DSCs, one DSC becomes part of the environment of the other DSC. Therefore, we have to ensure that the restrictions we make on environments also hold for DSCs. Since all services are assumed to terminate (Definition 2.3), we only have to ensure that the visibility of service calls is preserved by their termination.

If a service is called with an invisible message, but terminates with a visible message, the termination of a service reveals the initial message. Hence, we require a service not to reveal its call by a visible termination message. By the same argument, we make sure that the call of a service does not reveal the upcoming termination of the service.

Definition 3.10 (Visibility-preserving Services)**.** A service *serv* is *visibility-preserving* if

$$\forall \sigma, \sigma', v, v', t \cdot \langle handler_{serv}; \sigma\rangle \xrightarrow{Ini(serv)?v \frown t \frown Fin(serv)!v'} \langle SKIP; \sigma'\rangle \implies (Ini(serv)?v \sim \square \Leftrightarrow Fin(serv)!v' \sim \square)$$

A DSC c is *visibility-preserving* if all services in $prov_c$ are visibility-preserving.

Cooperation-non-interference is compositional for visibility-preserving DSCs under synchronized parallel composition.

Theorem 3.2 (Composition Non-interference)**.** *For a composition d with $d = p_A[\![s]\!]p_B$, $p_A \in \mathtt{Coop}_{p_A}$-NI , $p_B \in \mathtt{Coop}_{p_B}$-NI, and p_A, p_B* visibility-preserving *then $d \in \mathtt{Coop}_d$-NI*

Proof. We prove the contrapositive. Let $d \notin \mathtt{Coop}_d$-NI. Therefore, it exists an attack $(\omega_1, \omega_2, t_1)$ on d with $\omega_1, \omega_2 \in \mathtt{Coop}_d$, $\omega_1 \models d_{LTS} \xrightarrow{t_1}$ and $\forall t_2 \cdot \omega_2 \models d_{LTS} \xrightarrow{t_2} \implies (t_2 \not\approx t_1)$.

Then, there exist traces t_{1A}, t_{1B} such that $(t_{1A}[\![s]\!]t_{1B}) = t_1$ and $p_{A_{LTS}} \xrightarrow{t_{1A}}$ and $p_{B_{LTS}} \xrightarrow{t_{1B}}$.

As in proof for Theorem 3.1, we construct strategies which then result in attacks on p_A or p_B.

First we construct strategies for p_A and p_B: Let $j \in \{1,2\}$, $k, k' \in \{A, B\}$, $k \neq k'$. We define strategies ω'_{jk}:

$$\begin{aligned}
\omega'_{jk}(t) := \{m \mid \exists t'_1, t'_k, t'_{k'} \cdot t \sim t'_k \wedge t'_k[\![s]\!]t'_{k'} = t'_1\\
\wedge\, t'_k \frown m \lesssim t_{1k} \wedge s_k \xrightarrow{t'_k \frown m}\\
\wedge\, t'_{k'} \lesssim t_{1k'} \wedge s_{k'} \xrightarrow{t'_{k'}}\\
\wedge\, t'_1 \frown m \lesssim t_1 \wedge \omega_j \models p_A \parallel p_B \xrightarrow{t'_1 \frown m} \}
\end{aligned}$$

We extend the strategies ω_{jk} with terminating events such that ω_{jk} are cooperative strategies.

$$\begin{aligned}
&\omega_{jk}(t) := \omega'_{jk}(t) \cup\\
&\{m \mid \exists t', t'', v, w, serv \cdot t = t' \frown \mathit{Ini}(serv)!v \frown t' \wedge\\
&(\mathit{Fin}(serv)?w' \notin t') \wedge \mathit{Ini}(serv)!v \sim \Box \wedge\\
&m = \mathit{Fin}(serv)?w \wedge m \sim \Box \wedge p_k \xrightarrow{t \frown m} \wedge\\
&\neg(\exists u, w'' \cdot (\mathit{Fin}(serv)?w' \notin u) \wedge\\
&\quad \mathit{Fin}(serv)?w' \in \omega'_{jk}(t \frown u) \wedge \omega'_{jk}(t) \models p_k \xrightarrow{t \frown u})\}
\end{aligned}$$

We have to show $\omega_{jk} \in \texttt{Strat}$, $\omega_{jk} \in \texttt{Coop}_{p_k}$ and $\omega_{1k} \sim \omega_{2k}$.

Proof for $\omega_{jk} \in \texttt{Strat}$ See proof of Theorem 3.1.

Proof for $\omega_{jk} \in \texttt{Coop}_{p_k}$ $\omega_{jk} \in \texttt{Strat}$, so it is left to show two properties from Definition 3.8.
Lines 3.1, 3.2: We have to show

$$\begin{aligned}
&\forall t, t' \forall serv \forall \sigma \forall v \cdot serv \in req_c \wedge\\
&\quad \omega_{jk} \models p_k \xrightarrow{t \frown \mathit{Ini}(serv)!v \frown t'} \wedge \mathit{Fin}(serv) \notin t'\\
&\quad\quad \Longrightarrow \exists t'' \cdot \omega_{jk} \models p_k \xrightarrow{t \frown \mathit{Ini}(serv)!v \frown t' \frown t''} \wedge\\
&\quad \mathit{Fin}(serv)?v' \in \omega_{jk}(t \frown \mathit{Ini}(serv)!v \frown t' \frown t'')
\end{aligned}$$

Select $t, t', serv, v$ such that $serv \in req_{p_k}$,
$\omega_{jk} \models p_k \xrightarrow{t \frown \mathit{Ini}(serv)!v \frown t'}$, $\mathit{Fin}(serv) \notin t'$.

Case 1: $k = A$: Since $\omega_{jA} \models s_A \xrightarrow{t \frown \mathit{Ini}(serv)!v \frown t'}$, there exists witnesses $t'_1, t'_k, t'_{A'}$, such that $t \frown \mathit{Ini}(serv)!v \frown t' \sim t'_A$. Therefore, it exists $t'_A = u \frown \mathit{Ini}(serv)!v' \frown u'$ such that $t \sim u, \mathit{Ini}(serv)!v \sim \mathit{Ini}(serv)!v', t' \sim u'$. Therefore, u' can contain $\mathit{Fin}(serv)!w$ only if $\mathit{Fin}(serv)!w \sim \Box$.

Case 1.1: $(\mathit{Ini}(serv)!v \nsim \Box)$: Since $\omega_j \in \texttt{Coop}_d$ and $(\mathit{Ini}(serv)!v \nsim \Box)$ (Second condition of Definition 3.8), $\mathit{Fin}(serv)!w$ is not in u'. Again, since $\omega_j \in \mathtt{Coop}_d$, there exists u'', such that $\omega_j \models d \xrightarrow{u \frown \mathit{Ini}(serv)!w \frown u' \frown u''}$ $\wedge\ \mathit{Fin}(serv)?w' \in \omega_j(u \frown \mathit{Ini}(serv)!w \frown u' \frown u'')$. Also, due to $\omega_j \in \mathtt{Coop}_d$ and $(\mathit{Ini}(serv)!w \not\sim \Box)$, it holds: $(\mathit{Fin}(serv)?w' \not\sim \Box)$.

Similar to above, we can again split the trace u in u_A, u_B such that it is accepted by p_A. Since $u \sim t, \mathit{Ini}(serv)!v \sim \mathit{Ini}(serv)!v'$ and $t' \sim u'$, also $(t'_1, t'_k, t'_{A'})$ is a witness for $\mathit{Fin}(serv)!w' \in \omega_{jA}$.

Case 1.2: $\mathit{Ini}(serv)!v \sim \Box$: Due to the extension of ω'_{jk} to ω_{jk} as constructed above, there is a trace u such that the terminating message can be consumed.

Case 2: $k = B$: Proof is similar to Casees 1.1 and 1.2.

Definition 3.8, lines 3.3, 3.4 states:

$$\begin{aligned} \forall t, t', serv, \sigma \cdot serv \in req_c \wedge \omega_{jk} \models p_k \xrightarrow{t \frown \mathit{Ini}(serv)!v \frown t'} \wedge \\ \mathit{Fin}(serv) \notin t' \wedge \mathit{Fin}(?)v' \in \omega_{jk}(t \frown \mathit{Ini}(serv)!v \frown t') \\ \implies (\mathit{Ini}(serv)!v \sim \Box \Leftrightarrow \mathit{Fin}(serv)?v' \sim \Box) \end{aligned}$$

This follows by construction of ω_{jk}. ω'_{jk} is constructed from ω_j and it is ensured during construction of ω'_{jk} that visible calls are terminated visibly. Since ω'_{jk} is only extended with invisible termination events when constructing ω_{jk} , if the original call was also invisible, the claim holds.

Proof for $\omega_{1k} \sim \omega_{2k}$ See proof of Theorem 3.1. The definition of ω'_{jk} is essentially the same. Note that the extension to ω_{jk} only adds invisible events, therefore $\omega'_{jk} \sim \omega_{jk}$.

Form the four ω_{jk}, there results an attack p_A or p_B similar to the proof for Theorem 3.1. $\lhd$

Theorem 3.2 allows us to compose non-interferent DSCs and gain a non-interferent composition.

In Definition 3.9, we have presented a compositional non-interference property for DSCs which allows expressive specifications, including what-declassification and the specification of high message existence. The non-interference property is timing-insensitive and termination-insensitive. In an LTS which is non-interferent w.r.t a timing-insensitive non-interference property, an attacker may potentially be able to deduce high input from the order of observable messages in combination with knowledge about the typical execution time of services in the case of DSCs running in parallel. Termination-insensitivity means that an attacker is able to deduce from observable outputs whether or not a DSC terminated. Since DSCs by definition can not terminate, termination-insensitivity does not provide any additional knowledge to an attacker.

Further, our non-interference property is based on strategies as an explicit model for the environment a component runs in. This allows an intuitive understanding of the relation of non-interference of a component and a notion of confidentiality of high inputs in presence of an attacker as part of the environment.

3.4 Non-interference for Services

Analyzing non-interference for entire DSCs according to Definition 3.9 may be tedious, especially if $\sim$ contains complicated declassification terms. In particular, we do expect that complex definitions of $\sim$, although useful for specification, are hard to analyze. It would therefore be beneficial, if we could utilize an additional dimension of modularity.

The next natural level of modularity are services. Services are rather simple programs, in that they are deterministic and terminating; and only in combination with each other, they result in complex DSCs and compositions. In this section, we provide a non-interference property for services, which allows combining services to non-interferent DSCs and thus, re-using the compositionality result shown in Theorem 3.2.

Non-interference properties for terminating, deterministic programs can be found in literature in many different shapes (see discussion in Chapter 5). The notion presented here is inspired by non-interference for batch programs. In this notion, a program, or service, is non-interferent, if it terminates in equivalent post-execution states after being started in equivalent pre-execution states. We extend non-interference for batch programs with message passing. Especially, our definition of non-interference for services does not require the concept of strategies, but is restricted to trace properties.

We define the equivalence of states σ, σ' by an equivalence relation $\approx$ which defines a partitioning of the state in a high and a low part.

Example 3.5 (State Equivalence for Services)**.** Again, we revisit `pay` in Example 2.3. The service uses the variables *payCount*, *prodId*, *price*, and *amount*.

In order to specify that the state variables `payCount` and `price` may only contain low information, while all other variables may contain high information, we define $\approx$ as $\sigma \approx \sigma' \Leftrightarrow \sigma(\mathit{payCount}) = \sigma'(\mathit{payCount}) \wedge \sigma(\mathit{price}) = \sigma'(\mathit{price})$.

While the definition of high and low information in a state in literature often is considered as a specified security property of a program, we merely consider this partition as a technical necessity. The specification of high information is already given in our context by the equivalence relations on messages, i.e. on input and output level. The equivalent relation over states is only necessary in our context to ensure that high information stored in the state by service is not leaked by subsequent service calls. Therefore, we do not put any focus on the intended meaning of a partitioning defined by

$\approx$, neither do we discuss an attacker model, since we assume an attacker not to have direct access to the state at all. As long as there exists some suitable equivalence relation over states, it is sufficient for the purposes.

In order to define non-interference for services, we require a service not to reveal its execution by changes to the low part of the state, whenever the environment does not reveal it.

Definition 3.11 (Strictly Visibility-preserving Service)**.** A service *serv* is *strictly visibility-preserving* with respect to $\sim$ and $\approx$ if *serv* is visibility preserving according to Definition 3.10 and

$$\begin{aligned}
\forall \sigma, \sigma', t, t' \cdot \langle handler_{serv}; \sigma\rangle \xrightarrow{t \frown t'} \langle SKIP; \sigma'\rangle \implies && (3.5)\\
(t \rhd \mathbb{I} \sim \langle\rangle \implies t \rhd \mathbb{O} \sim \langle\rangle \;\wedge && (3.6)\\
t \frown t' \rhd \mathbb{I} \sim \langle\rangle \implies \sigma \approx \sigma') && (3.7)
\end{aligned}$$

Line 3.6 in Definition 3.11 states that if all inputs provided by the environment to the service for any prefix of a terminating trace, the service may only provide invisible outputs. This condition is a more strict version of *Visibility-Preserving Services* as defined in Definition 3.10. Additionally to the condition that a service called invisibly also terminates invisibly, Definition 3.11 requires all intermediate service calls to be invisible.

However, the service may provide visible output after visible input, which means that the environment answered an invisible service call with a visible terminating message. In this case, it is not the service under analysis which is insecure but the environment is not cooperative. In the context of cooperative strategies, a DSC cannot be non-interferent, if a service makes a visible service call after being called with an invisible initial message. Since we are about to define a non-interference definition for services, which is independent from strategies, we have to make this property explicit.

Additionally, we require the service to terminate in a poststate which is equivalent w.r.t. $\approx$ to the prestate if only invisible inputs were sent to a service (Line 3.7). This property formalizes that a service must not leak the fact that it was called to the state. This condition is necessary in order to provide sequential compositionality for services.

Example 3.6. Assume for the moment the messages on initial and terminating channels for service `pay` of DSC `Cart` and for the required service `registerSale` to be specified to be invisible. After a call to the service, `Cart` changes the value of variable `payCount` on the state. Since the state, in our model, is not directly observable by the attacker, these changes can not directly provide any information to the attacker. Since all messages sent and received during execution of `pay` are invisible, the attacker is assumed not to be able to directly observe them either.

However, in a subsequent call of another service, the value of `payCount` may be revealed as a return value or parameter. If the respective message is observable by the attacker, he may be able to indirectly see the changed value of `payCount` and can therefore deduce that `pay` was called.

We define non-interference for a service according to the classic definition of non-interference for batch programs and add the consideration of equivalent traces.

Definition 3.12 (Service Non-interference)**.** A Service *serv* is non-interferent with respect to $\sim$ and $\approx$, written $serv \in \mathtt{SNI}_{\sim}^{\approx}$ if it is *strictly visibility-preserving* with respect to $\sim$ and $\approx$ and

$$
\begin{aligned}
&\forall \sigma_1, \sigma_2, \sigma_1', \sigma_2', t_1, t_2 \cdot \sigma_1 \approx \sigma_2 \wedge && (3.8)\\
&\quad \langle handler_{serv}; \sigma_1\rangle \xrightarrow{t_1} \langle SKIP; \sigma_1'\rangle \quad \wedge && (3.9)\\
&\quad \langle handler_{serv}; \sigma_2\rangle \xrightarrow{t_2} \langle SKIP; \sigma_2'\rangle && (3.10)\\
&\Longrightarrow\\
&\quad (t_1 \rhd \mathbb{I} \ \sim \ t_2 \rhd \mathbb{I} \implies \sigma_1' \approx \sigma_2') \quad \wedge && (3.11)\\
&\quad (\forall t_1' \leq t_1, t_2' \leq t_2 \cdot \ t_1' \rhd \mathbb{I} \ \sim \ t_2' \rhd \mathbb{I} \implies && (3.12)\\
&\qquad \exists t_1', t_2'' \cdot t_1' \frown t_1'' \leq t_1, t_2' \frown t_2'' \leq t_2 \wedge && (3.13)\\
&\qquad\quad t_1' \frown t_1'' \ \sim t_2' \ \frown t_2'') && (3.14)
\end{aligned}
$$

A service started in two equivalent states (Line 3.8) has to terminate (Line 3.10) in equivalent states, if the input provided by the environment is equivalent for both runs (Line 3.11). Implicitly, this condition encodes a well-behaving environment in the sense that we assume the environment not to leak information.

The second condition ensures that the service does not leak information by providing non-equivalent output to the environment after receiving equivalent input. (Lines 3.12) to (3.14) ensure that t_1 and t_2 are equivalent up to the first non-equivalent input. For all prefixes of the two traces produced during execution which got provided equivalent input (Line 3.12), the traces either are equivalent, or at least there are further events in the traces such that the traces can become equivalent (Line 3.14). We give in the condition the possibility that both prefixes can be extended. In fact, it is sufficient to only extend the prefix whose equivalence projection is shorter. But phrasing this formally does not result in a simpler formula.

Example 3.7. The implementation of our example, the DSCs `Cart` manages the state variables `product`, `prodprice`, `prodamount`, `numbuys`, `numPays`, and `numCheck`. The low information, from the point of view of the billing department, are the ordered product, the product's price and the ordered amount. However, the information of the number of calls to the respective services are irrelevant to the billing department.

We can define an equivalence relation over states the following way:

$$\begin{aligned}\sigma \approx \sigma' \Leftrightarrow & \sigma(product) = \sigma'(product) \wedge \\ & \sigma(prodprice) = \sigma'(prodprice) \wedge \\ & \sigma(prodamount) = \sigma'(prodamount)\end{aligned}$$

All services provided by `Cart` are non-interferent w.r.t. the previously discussed equivalence relation $\sim$ in Example 3.1 and $\approx$.

We characterize DSCs that only have non-interferent services with respect to the equivalence relation $\approx$.

Definition 3.13 (DSC State Non-interference)**.** A DSC c is $(\sim, \approx)$-NI if $\forall serv \in prov_c \cdot serv \in \mathtt{SNI}^{\approx}_{\sim}$.

If a DSC only provides non-interferent services w.r.t the same equivalence relation over states, and thus has the property in Definition 3.13, every execution of a service ensures that there is no leak of previous secret input due to an output. Neither is there a leak of previous secret input into the low part of the DSC's state. So Definition 3.13 implies Cooperation-Non-Interference for a DSC.

Theorem 3.3. *Given an equivalence relation $\approx$ such that the DSC c is $(\sim, \approx)$-NI. Then $\langle body_c; \sigma_0\rangle \in \mathtt{Coop}_c$-NI.*

Before we can prove the theorem, we need the following lemma.

Lemma 3.8. *Given $\omega_1 \sim \omega_2 \in \mathtt{Coop}_c$, a DSC c, traces p_1 and p_2, such that $p_1 \sim p_2$, and states $\sigma_1, \sigma_2, \sigma_1'$ with $\sigma_1 \approx \sigma_2$. Let further and $\omega_1 \models \langle body_c; \sigma_0\rangle \xrightarrow{p_1} \langle body_c; \sigma_1\rangle$ and $\omega_2 \models \langle body_c; \sigma_0\rangle \xrightarrow{p_2} \langle body_c; \sigma_2\rangle$. Also, let serv be a non-interferent service provided by c. Then*

$$\begin{aligned}&\forall t_1 \cdot \langle handler_{serv}; \sigma_1\rangle \xrightarrow{t_1} \langle SKIP; \sigma_1'\rangle \implies \\ &\quad (t_1 \sim \langle\rangle \wedge \sigma_1 \approx \sigma_1') \vee \\ &\quad \exists t_2, \sigma_2' \cdot \langle handler_{serv}; \sigma_2\rangle \xrightarrow{t_2} \langle SKIP; \sigma_2'\rangle \wedge \\ &\qquad t_1 \sim t_2 \wedge \sigma_1' \sim \sigma_2'\end{aligned}$$

The lemma states that for all pairs of executions of a series of non-interferent services, such that the resulting traces p_1 and p_1 are equivalent and the poststate of the last service executions σ_1 and σ_2, for every execution of a non-interferent service to its termination, there exists a second execution of that service started in the second poststate, such that the then resulting traces and poststates again are equivalent.

Proof for Lemma 3.8. Let t_1 be an arbitrary trace with first message m. We make a case distinction over the visibility of m.

Case 1: $m \sim \square$: Since *serv* is non-interferent, *serv* is also visibility-preserving. By definition Definition 3.11, $t_1 \sim \langle\rangle$ and $\sigma_1 \sim \sigma_1'$.

Case 2: $m \nsim \square$: We show that for all traces $t_1' \leq t_1$ exists a trace t_2 such that $t_1' \sim t_2'$ and $\omega_2\langle handler_{serv}; \sigma_2\rangle \xrightarrow{t_2'}$. We prove the property by induction over $n = |t_1'|$.

Start: $n = 1$: Since m is visible, $\omega_1 \sim \omega_2$ and $m \in \omega_1(p_1)$ and $p_1 \sim p_2$, there exists $m_2 \in \omega_2(p_2)$ with $m \sim m_2$. Therefore $t_2' = m_2$.

Step: $n+1$: We know by induction $t_2' \sim t_1'$ and choose m by $t_1' \frown m \leq t_1$. We make a case distinction over the visibility of m.

Case 2.1: $m \sim \square$: Then t_2' is the witness for the hypothesis.

Case 2.2: $m \nsim \square$: Again, we make a case distinction, this time over m being input or output.

Case 2.2a: $m \in \mathbb{I}$: Therefore there exists some t_1'' such that $t_1' = t_1'' \frown o_1$, $o_1 \in \mathbb{O}$. Since $\omega_1 \in \texttt{Coop}_c$, and o_1 being the call of a service, which is terminated by ω_1 visibly, $(o_1 \nsim \square)$. Since $t_1' \sim t_2'$ and $\omega_2 \in \texttt{Coop}_c$, there exists the prefix t_2'' with $t_2' = t_2'' \frown o_2$ and $(o_2 \nsim \square)$. Again, since $m \in \omega_1(p_1 \frown t_1')$ and $\omega_1 \sim \omega_2$ and $p_1 \frown t_1' \sim p_2 \frown t_2'$, there exists $m' \in \omega_2(p_2 \frown t_2')$ with $m' \sim m$. Therefore $t_1' \frown m \sim t_2' \frown m'$ and $\omega_2 \models \langle handler_{serv}; \sigma_2\rangle \xrightarrow{t_2' \frown m'}$.

Case 2.2b: $m \in \mathbb{O}$: We know $t_1' \sim t_2'$, thus $t_1' \frown m \rhd \mathbb{I} \sim t_2' \rhd \mathbb{I}$. Since *serv* is non-interferent and $\sigma_1 \approx \sigma_2$, there exists t_2'', m' with $t_2' \frown m' \frown t_2'' \leq t_2$ with $t_1' \frown m \sim t_2' \frown m' \frown t_2''$ and $\langle handler_{serv}; \sigma_2\rangle \xrightarrow{t_2' \frown m'}$.

If m' is visible, then $m' \sim m$ and $m' \in \mathbb{O}$, and therefore we know $\omega_2 \models \langle handler_{serv}; \sigma_2\rangle \xrightarrow{t_2' \frown m'}$.

If $m' \sim \square$ and $m' \in \mathbb{O}$, then m' represents the invisible call of a required service. Since $\omega_2 \in \texttt{Coop}_c$ there exists the message terminating the service call m'' such that $m'' \sim \square$ and $m'' \in \omega_2(p_2 \frown t_2' \frown m')$ and therefore $\omega_2 \models \langle handler_{serv}; \sigma_2\rangle \xrightarrow{t_2' \frown m' \frown m''}$. Since $m' \sim m'' \sim \square$, it holds $t_1' \frown m \rhd \mathbb{I} \sim t_2' \frown m' \frown m'' \rhd \mathbb{I}$. We redefine t_2 from the induction hypothesis as $t_2' := t_2' \frown m' \frown m''$.

We now recursively apply this proof until case 2.2b is not applied anymore. Since *serv* is terminating and t_2' initially starts with a visible event, at some point m' will be visible and therefore this recursion terminates. Therefore, $t_1' \frown m \sim t_2' \frown m'$ and $\omega_2 \models \langle handler_{serv}; \sigma_2\rangle \xrightarrow{t_2' \frown m'}$, which is the induction hypothesis for $n + 1$.

In conclusion, since t_1 terminates on a visible output (Definition 3.10) and $t_2 \sim t_1$, t_2 is a terminating trace. Since *serv* is non-interferent $\sigma_1' \sim \sigma_2'$ holds. This is equivalent with the last two lines of Lemma 3.8. $\lhd$

Now, we can prove Theorem 3.3.

Proof for Theorem 3.3. We show the contrapositive, i.e. we assume an attack to exist which contradicts $c \in (\sim, \approx)$-NI.

Given a DSC c and a $\texttt{Coop}_c - \mathit{Attack}$ (ω_1, ω_2, t), i.e. $\omega_1, \omega_2 \in \texttt{Coop}_c$, $\omega_1 \sim \omega_2$, $\omega_1 \models \langle body_c; \sigma_0\rangle \xrightarrow{t}$ and $\forall t' \cdot \omega_2 \models \langle body_c; \sigma_0\rangle \xrightarrow{t'} \implies (t \not\approx t')$.

We select t_1 as the longest prefix of t for which an equivalent trace exists, which is consistent with ω_2 and $\langle body_c; \sigma_0\rangle$. We select t_2 such that it is consistent with ω_2 and $\langle body_c; \sigma_0\rangle$ and equivalent to t_1 and has the most visible events in the trace among the candidates. And finally, among those, we select the longest possible trace, meaning, there are no invisible events following t_2 for c under ω_2.

Formally let t_1, t_2 such that:

1. $t_1 \le t \land \forall t' \cdot t' \le t \implies (\exists t'' \cdot (t'' \sim t' \land \omega_2 \models \langle body_c; \sigma_0\rangle \xrightarrow{t''}) \implies |t'| \le |t_1|)$.
2. $t_2 \sim t_1$
3. $\omega_2 \models \langle body_c; \sigma_0\rangle \xrightarrow{t_2}$
4. $\forall t' \cdot (t' \lesssim t \land \omega_2 \models \langle body_c; \sigma_0\rangle \xrightarrow{t'}) \implies |t' \rhd \blacksquare| \le |t_2 \rhd \blacksquare|$
5. $\forall t' \cdot (t_2 \le t' \land t' \sim t_1 \land \omega_2 \models \langle body_c; \sigma_0\rangle \xrightarrow{t'}) \implies t' = t_2$

We split t_i with $i \in \{1,2\}$, into t_{ia} and t_{ib} such that the last event of t_{ia} represents the termination of a provided service and t_{ib} does not contain an event representing termination of a service provided by c:

$t_i =: t_{ia} \frown t_{ib}$ with $i \in \{1,2\}$ such that $\langle body_c; \sigma_0\rangle \xrightarrow{t_{ia}} \langle body_c; \sigma_i\rangle$ and $\langle body_{serv_i}; \sigma_i\rangle \xrightarrow{t_{ib}} \langle rest_i; \sigma_i'\rangle$.

This means, that there is an event m in t following t_1. and some event m', which might be consumed by c and provided by ω_2. More formally, let m, m' such that $t_1 \frown m \le t$ and $\langle rest_2; \sigma_2'\rangle \xrightarrow{m'}$. The event m' must be visible, because otherwise there would have existed a longer trace t_2, such that t_2 satisfies condition 5 above.

We make a case distinction over $m \in \mathbb{I}$ and $m \in \mathbb{O}$ and show that both cases result in a contradiction to the original assumption.

Case 1: $m \in \mathbb{I}$:Since $\omega_1 \sim \omega_2$ and m visible, we know that there exists an $m'' \in \omega_2(t_2)$ such that $m'' \sim m$. If m represents the call of a provided service, then m'' is a call to the same service. Otherwise, m represents the termination of a called service, meaning the last event in t_1 represents the call of this service and the call is visible. Since $t_1 \sim t_2$, t_2 also ends with a visible call to the same service. Therefore, $\langle body_c; \sigma_0\rangle \xrightarrow{t_2 \frown m'}$. So there exists a trace t_2' such that $t_2' \sim t_1$ and $\omega_2 \models \langle body_c; \sigma_0\rangle \xrightarrow{t_2'}$, i.e. $\omega_2 \models \langle body_c; \sigma_0\rangle \xrightarrow{t_2' \frown m''}$. But this means due to Lemma 3.1, that $|{t_2}' \frown m' \rhd \blacksquare| > |t_2 \rhd \blacksquare|$, which contradicts the construction of t_2.

Case 2: $m \in \mathbb{O}$: First we show, that $m' \in \mathbb{O}$. Since $serv_1$ is terminating, there exists a trace t_1' with $t_{1a} \le t_1'$ such that $\langle body_{serv_1}; \sigma_1\rangle \xrightarrow{t_1'} \langle SKIP; \sigma_1''\rangle$. Further, by induction and with Lemma 3.8, we also know $\sigma_1 \approx \sigma_2$. Since

$serv_1 \in \mathtt{SNI}^{\approx}_{\sim}$ we know that for all traces t_2' such that $\langle body_{serv_2}; \sigma_2 \rangle \xrightarrow{t_2'} \langle SKIP; \sigma_2'' \rangle$ it holds that $t_1' \rhd \mathbb{I} \sim t_2' \rhd \mathbb{I} \implies t_1' \sim t_2'$

$serv_2$ can not be blocked due to missing termination of a called service. Therefore, if $m' \in \mathbb{I}$, $m \in Ini(prov_c)$. Since $m \in \mathbb{O}$, $(t_{1b} \neq \langle\rangle)$, but $t_{1b} \sim \langle\rangle$, especially the initial message is invisible. This contradicts Definition 3.11.

Therefore $m' \in \mathbb{O}$ and by definition $\omega_2 \models \langle body_c; \sigma_0 \rangle \xrightarrow{t_2 \frown m'}$. Due to the construction of t_2, m' is visible.

According to Definition 2.3, there exists a trace t_2', such that $t_2' \rhd \mathbb{I} \sim t_1' \rhd \mathbb{I}$. Therefore, $t_2' \sim t_1'$, and it has to hold that $m' \sim m$, therefore $t_1 \frown m \sim t_2 \frown m'$, which is a contradiction to the construction of t_2. $\lhd$

Definition 3.12 provides us with a non-interference property for services which we assume to be easier to analyze than non-interference for an entire DSC. Theorem 3.3 states that it is sufficient to show non-interference for all services provided by a DSC in order to show non-interference for a DSC. Together with compositionality of non-interference for DSCs in Theorem 3.2, we can reduce non-interference analysis for a component-based system to non-interference analysis of each provided service.

3.5 Conclusion

In this chapter, we defined non-interference for LTS using strategies as explicit models for the environment. Strategies can also be seen as a combination of an attacker and non-attacking users of a system, where the attacker may know low input and can observe low output. Our non-interference property then states that the attacker can not distinguish between regular user behavior, which differs on high input, since every observation the attacker makes could be caused by any user behavior.

Our specifications of high and low information is based on equivalence relations which allows very expressive declassification of information, including parameter-dependent classification of the existence of messages. While allowing expressive specifications, we did prove that non-interference for LTS is compositional.

We further specialized the non-interference notion for DSC by limiting the environment to cooperative environments which comply to the standard assumption in component-based systems that all services terminate. This specialization makes our non-interference notion more precise than other non-interference definitions in literature, since we rule out the case that the execution of a DSC is blocked due to the environment not providing high, terminating messages for services called by the DSC. We showed that non-interference for DSCs is also compositional under synchronized parallel composition.

Finally, we provided a non-interference property for services as a combination of trace-based and state-based non-interference. For both types of non-interference, a selection of analysis approaches exist for different programming languages, which gives us the confidence that several different analysis methods can be implemented for our non-interference property for services. We describe one analysis method for object oriented programs in Chapter 8 in the second part of this thesis as an example.

The biggest limitation of the work we have presented in this chapter is that in order to show non-interference for a DSC, one non-interference specification has to be identified such that all services are non-interferent w.r.t this specification. While the specification of inputs and outputs is usually (at least in parts) domain driven, the equivalence relation over states has to be identified by an expert such that it is valid for all services. This requires knowledge about the internals of the entire component and identifying it in practice is a very cumbersome task. In the next chapter we provide a constructive approach to find non-interference specifications for services which can be combined to gain one component-global non-interference specification.

4 Modular Specification with Dependency Clusters

4.1 Introduction

In Chapter 3 we developed a powerful approach for specification of non-interference properties in DSCs and formally defined non-interference. The additional notion of non-interference for services allows in general to build analysis techniques for checking non-interference in DSCs. We assume a specification of low inputs and outputs to be given a priori. This specification practically is derived by some domain expert who decides what a potential attacker can observe and input information he should be able to know.

The presentation in the previous chapter can be seen as a top-down approach. Given a domain-driven non-interference specification we discussed how a system consisting of DSCs can be shown to be non-interferent w.r.t. the specification.

Components in modern systems are designed to be re-used in different contexts and its implementation is tailored to fit special requirements in a new context. Each time the context or some part of the implementation changes, a new specification has to be found and the component has to be re-analyzed for non-interference.

In this chapter, we change our perspective and consider information flow as a property of a program, independently from the environment it is run in and attackers that the domain expert may identify. The result in this chapter can be seen as a bottom-up approach, where first modular dependencies of services are identified and then combined to gain a security guarantee for a domain-driven specification for DSCs and systems consisting of DSCs.

Dependencies between inputs, outputs, and the state of a DSC are inherent to the implementation of the services.

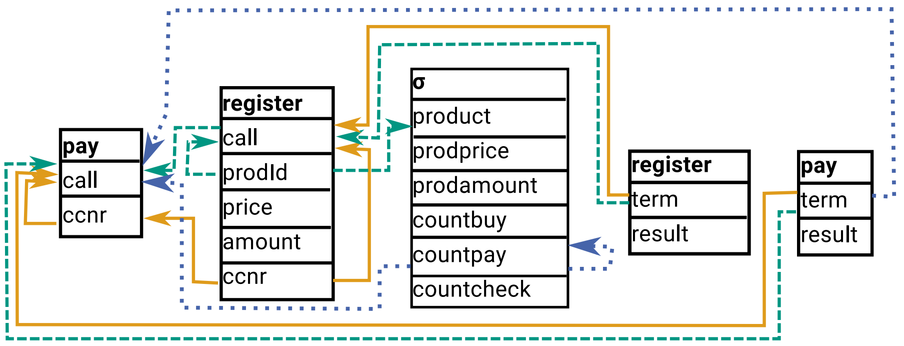

Figure 4.1: Simple information dependencies in DSC `Cart` caused by the service `pay`

Example 4.1. We explain what we mean with dependencies being inherent to the implementation of a service using an example. Figure 4.1 illustrates some flows caused by the implementation of the service `pay` in DSC `Cart` implemented as in Example 2.3. Different colors of arrows indicate different clusters of dependencies between information contained in messages and state variables. The figure illustrates three different sets of dependencies of information, which are independent from each other and purely caused by the implementation of `pay`. The solid orange arrows indicate that the information passed as parameter `ccnr` during the service call on `registerSale` depends on whether the service is called and on the information passed as parameter `ccnr` when `pay` is called. This information again depends on the existence of the call of `pay` itself. Additionally, the existence of the termination messages of `pay` and `registerSales` depend on the existence of the respective initial messages.

The dashed green arrows indicate that the information passed as parameter `prodId` during the call of `registerSales` depends on the information stored in the state variable `product` and on the existence of the call to `registerSales`. The existence of this call depends on the existence of the call to `pay`. Again, the existence of the termination messages of each service depends on the existence of the initial message.

The third set, indicated by blue dotted arrows, illustrates the dependencies for the value of the state variable `countpay`. For one, it depends on the value of `countpay` before the service call as well as the existence of the call to `pay` itself. The existence of the termination message for `pay` depends on the existence of the service call.

In this chapter, we formalize service-local dependencies as *Dependency Clusters*. We show that Dependency Clusters can be combined to bigger, more complicated information flow specifications. Further, we show how Dependency Clusters can be combined in order to gain non-interference specifications for DSCs and ultimately to gain information flow specifications for compositions. We show that only Dependency Clusters have to be analyzed using program analysis methods, while combinations of Dependency Clusters for DSC-wide and system-wild non-interference specifications can be verified by solving simple first order logic proof obligations. Overall, we show in this chapter how Dependency Clusters can be used as building blocks for domain-driven security specifications and illustrate how Dependency Clusters are robust against changing environments and evolving components.

In the next section we introduce a simple specification language which is more intuitive than equivalence relations. We use this specification language for examples throughout this chapter. In Section 4.3 we formally define Dependency Clusters and show that different Dependency Clusters of one service can be combined to a new non-interference specification. In Section 4.4 we show that Dependency Clusters of different services within one DSC can be combined to gain an information flow specification for a complete DSC by ensuring one first order logic (FOL) condition per service. In Section 4.5 we show that non-interference specifications for a DSC can be checked against a system-global non-interference specification, again by solving a simple FOL condition. Finally, we conclude the chapter.

The results in this chapter are based on work previously published by the author in (Greiner et al. [2017b] and Greiner et al. [2017a]).

4.2 A List-based Specification Language

We will use several different non-interference specifications to illustrate the theorems and insights concerning Dependency Clusters in the following. Directly defining these specifications as equivalence relations as in the previous chapter leads to confusing and verbose definitions. We therefore introduce an alternative, list-based, notion for non-interference specifications. We only indicate the formal semantics for the language, since it is for presentation purposes only.

A list-based non-interference specification consists of a list *Vis* expressing the visibility specification for messages, a list *LowIO* expressing the low information in messages and a list *LowIO* expressing the low part of the state. The grammar of for the specification is as follows:

$$\begin{array}{ll} \mathit{Vis}: & \langle T(serv_1).(bexpr_1), \ldots, T(serv_m).(bexpr_m)\rangle \\ \mathit{LowIO}: & \langle T(serv_1).(pexpr_{11}, \ldots, pexpr_{1n}), \ldots, \\ & \quad T(serv_m).(pexpr_{m1}, \ldots, pexpr_{mo})\rangle \\ \mathit{LowState}: & \langle sexpr_1, \ldots sexpr_n\rangle \end{array}$$

where $T \in \{\mathit{Ini}(.), \mathit{Fin}(.)\}$, $bexpr_i$ are boolean expressions either over parameters or r for the return value of service, $pexpr_{ij}$ is an expression over parameter of a service or r and $sexpr_i$ are expressions over state variables.

A message $\mathit{Ini}(\mathit{serv}).p$ is visible, if there exists an entry $\mathit{Ini}(\mathit{serv}).(\mathit{bexpr})$ in *Vis*, such that *bexpr* evaluates to true for value p, and analogous for the termination channel.

Two messages $\mathit{Ini}(\mathit{serv}).p$ and $\mathit{Ini}(\mathit{serv}).p'$ are equivalent, if either both are invisible, or there exists an entry $\mathit{Ini}(\mathit{serv}).(\mathit{pexpr}_1, ...\mathit{pexpr}_n)$, such that all pexpr_i evaluate to the same values for p and p', and analogous for termination channels.

Two states σ and σ' are equivalent, iff all entries in *LowState* evaluate to the same value in the states σ and σ'.

Example 4.2. In order to specify that the initial and terminating messages of the services `pay` and `registerSales` are visible, we write

$$\begin{aligned}\mathit{Vis}_1 := \langle &\mathit{Ini}(\mathit{pay}).(\mathit{true}),\ \mathit{Fin}(\mathit{pay}).(\mathit{true}),\\ &\mathit{Ini}(\mathit{registerSales}).(\mathit{true}),\ \mathit{Fin}(\mathit{registerSales}).(\mathit{true})\rangle\end{aligned}$$

The resulting equivalence relation is as follows:

$$\begin{aligned}\alpha.v \not\approx \square \Leftrightarrow &\alpha \in \{\mathit{Ini}(\mathit{pay}), \mathit{Fin}(\mathit{pay}), \mathit{Ini}(\mathit{registerSales}),\\ &\quad \mathit{Fin}(\mathit{registerSales})\}\\ &\wedge ((\alpha = \mathit{Ini}(\mathit{pay}) \wedge \mathit{true}) \vee (\alpha = \mathit{Fin}(\mathit{pay}) \wedge \mathit{true}) \vee\\ &\quad (\alpha = \mathit{Ini}(\mathit{registerSales}) \wedge \mathit{true}) \vee\\ &\quad (\alpha = \mathit{Fin}(\mathit{registerSales}) \wedge \mathit{true}))\end{aligned}$$

Messages not mentioned in the list are specified invisible by default.

In the lists *LowIO*, we specify the equivalence relation over messages in case a message is visible. To express that the credit card number provided as a parameter of `pay` is low, and the parameters `prodId` (the ID of a product) and `ccnr` (the credit card number) of the service `registerSale` are low, we specify:

$$\begin{aligned}\mathit{LowIO}_1 := \langle &\mathit{Ini}(\mathit{pay}).(\mathit{ccnr}), \mathit{Fin}(\mathit{pay}).(0),\\ &\mathit{Ini}(\mathit{registerSales}).(\mathit{prodId}, \mathit{ccnr}), \mathit{Fin}(\mathit{registerSales}).(\mathit{0})\rangle\end{aligned}$$

The list in parenthesis is a list of expressions over the parameters or r for the return value, which can be evaluated for two concrete messages. Two messages on the respective services are equivalent, if the lists evaluate to the same values.

Formalized directly as equivalence relation, the specification expresses:

$$\begin{aligned}\alpha.v \sim \beta.v' \Leftrightarrow & (\alpha.v \sim \square \wedge \beta.v' \sim \square)\vee \\ & (\alpha = \beta \wedge \\ & \quad ((\alpha = \mathit{Ini}(\mathit{pay}) \wedge v = \mathit{ccnr} \wedge v' = \mathit{ccnr}' \\ & \qquad \wedge\, \mathit{ccnr} = \mathit{ccnr}')\vee \\ & \quad (\alpha = \mathit{Fin}(\mathit{pay}) \wedge v = r \wedge v' = r' \wedge 0 = 0)\vee \\ & \quad (\alpha = \mathit{Ini}(\mathit{registerSales})\wedge \\ & \qquad v = (\mathit{prodId}, \mathit{price}, \mathit{amount}, \mathit{ccnr})\wedge \\ & \qquad v' = (\mathit{prodId}', \mathit{price}', \mathit{amount}', \mathit{ccnr}')\wedge \\ & \qquad \mathit{prodId} = \mathit{prodId}' \wedge \mathit{ccnr} = \mathit{ccnr}')\vee \\ & \quad (\alpha = \mathit{Fin}(\mathit{registerSales}) \wedge v = r \wedge v' = r' \wedge 0 = 0)))\end{aligned}$$

Providing a constant as an expression specifies that all communicated content for this message is high.

Similar the list *LowState* specifies the low part of the state as a list of expressions over state variables. To express that the variable managing the bought product (`product`), we specify:

$$\mathit{LowState}_1 := \langle \mathit{product} \rangle$$

The resulting equivalence relation then is as follows:

$$\sigma \approx \sigma' \Leftrightarrow \sigma(\mathit{product}) = \sigma'(\mathit{product})$$

Together, the lists $\mathit{Vis}_1, \mathit{LowIO}_1$, and $\mathit{LowState}_1$ provide us with the specification tuple $(\sim_1, \approx_1)$. Note that `pay` is in $\mathtt{SNI}^{\approx_1}_{\sim_1}$.

The lists only serve as a compact representation of equivalence relations. All theorems and proofs provided in the remainder are based on our original notion using equivalence relations. Thus, the list notion does not change the generality of the framework.

4.3 Dependency Clusters and Services

A non-interference specification of a service describes dependencies between inputs, outputs and parts of a state managed by a DSC. We call service-local, implementation-specific non-interference specifications *Dependency Cluster*. Each of the dependencies with the same color illustrated in Figure 4.1 represent a simple Dependency Cluster.

Definition 4.1 (Dependency Cluster for Services)**.** A Dependency Cluster for a service *serv* is a tuple $(\sim, \approx)$ such that $\mathit{serv} \in \mathtt{SNI}^{\approx}_{\sim}$.

Example 4.3. We provide here two different Dependency Clusters for the service `pay`.

$$\begin{aligned}
\mathit{Vis}_2 :=\ & \langle \mathit{Ini}(\mathit{pay}).(\mathit{true}),\ \mathit{Fin}(\mathit{pay}).(\mathit{true}),\\
& \ \mathit{Ini}(\mathit{registerSales}).(\mathit{true}),\ \mathit{Fin}(\mathit{registerSales}).(\mathit{true})\rangle\\
\mathit{LowIO}_2 :=\ & \langle \mathit{Ini}(\mathit{pay}).(\mathit{ccnr}),\ \mathit{Fin}(\mathit{pay}).(0),\ \mathit{Ini}(\mathit{registerSales}).(\mathit{ccnr}),\\
& \ \mathit{Fin}(\mathit{registerSales}).(\mathit{0})\rangle\\
\mathit{LowState}_2 :=\ & \langle 0\rangle
\end{aligned}$$

$$\begin{aligned}
\mathit{Vis}_3 :=\ & \langle \mathit{Ini}(\mathit{pay}).(\mathit{true}),\ \mathit{Fin}(\mathit{pay}).(\mathit{true}),\\
& \ \mathit{Ini}(\mathit{registerSales}).(\mathit{true}),\ \mathit{Fin}(\mathit{registerSales}).(\mathit{true})\rangle\\
\mathit{LowIO}_3 :=\ & \langle \mathit{Ini}(\mathit{pay}).(0), \mathit{Fin}(\mathit{pay}).(0),\ \mathit{Ini}(\mathit{registerSales}).(\mathit{prodId}),\\
& \ \mathit{Fin}(\mathit{registerSales}).(\mathit{0})\rangle\\
\mathit{LowState}_3 :=\ & \langle \mathit{product}\rangle
\end{aligned}$$

The Dependency Cluster indexed by 2 represents the information flow illustrated in Figure 4.1 with the solid yellow arrows. The Dependency Cluster indexed by 3 is displayed in Figure 4.1 with the dashed green arrows.

Since we allow in our specification a powerful way of declassification, there does not exist something as a “smallest” Dependency Cluster. We can construct arbitrarily many different Dependency Clusters for any service. Therefore, we can not provide a set of Dependency Clusters for a service which describe all dependencies of information in the service.

Example 4.4. We used $\mathit{LowState}_3 := \langle \mathit{product}\rangle$ in the example, but we could also specify $\mathit{LowState}_3 := \langle \mathit{product}\%2\rangle$, $\mathit{LowState}_3 := \langle \mathit{product}\%4\rangle$ and so on. Also any combination of parameters can be used for defining Dependency Cluster, i.e. instead of $\mathit{Ini}(\mathit{registerSales}).(\mathit{prodId}, \mathit{ccnr})$, we could specify $\mathit{Ini}(\mathit{registerSales}).(\mathit{prodId} + \mathit{ccnr})$.

Interestingly, Dependency Clusters are compositional, in the sense that given two Dependency Clusters $(\sim_1, \approx_1)$ and $(\sim_2, \approx_2)$ for a service *serv*, their combination $(\sim_1, \approx_1) + (\sim_2, \approx_2) := (\sim_1 \cap \sim_2, \approx_1 \cap \approx_2)$ is also a Dependency Cluster for *serv*.

Theorem 4.1 (Dependency Cluster Compositionality)**.** *Let* $d_1 = (\sim_1, \approx_1)$ *and* $d_2 = (\sim_2, \approx_2)$ *be Dependency Clusters for a service serv. Then* $d_1 + d_2$ *is a Dependency Cluster for serv.*

Proof for Theorem 4.1. Assume *serv* is non-interferent with respect to $(\sim_1, \approx_1)$ and $(\sim_2, \approx_2)$. Let $\sim := \sim_1 \cap \sim_2$ and $\approx := \approx_1 \cap \approx_2$.

We first show that *serv* is visibility-preserving w.r.t. $(\sim, \approx)$. Select arbitrarily σ, σ', t, t' such that $\langle \mathit{handler}_{\mathit{serv}}; \sigma\rangle \xrightarrow{t \frown t'} \langle \mathit{SKIP}; \sigma'\rangle$ and $t \rhd \mathbb{I} \ \sim\ \langle\rangle$.

By definition of $\sim$ and $\approx$, this implies $t \rhd \mathbb{I} \ \sim_1 \ \langle\rangle$ and $t \rhd \mathbb{I} \ \sim_2 \ \langle\rangle$. Since d_1 and d_2 are Dependency Clusters for *serv*, this means $t \rhd \mathbb{O} \ \sim_1 \ \langle\rangle$ and $t \rhd \mathbb{O} \ \sim_2 \ \langle\rangle$, and by definition of $\sim$ also $t \rhd \mathbb{O} \ \sim \ \langle\rangle$.

Now, we assume $t \frown t' \rhd \mathbb{I} \ \sim \ \langle\rangle$. Again, by definition of $\sim$ and $\approx$, this implies $t \frown t' \rhd \mathbb{I} \ \sim_1 \ \langle\rangle$ and $t \frown t' \rhd \mathbb{I} \ \sim_2 \ \langle\rangle$ and since d_1 and d_2 are Dependency Clusters $\sigma \approx_1 \sigma'$ and $\sigma \approx_2 \sigma'$ which implies by definition of $\approx$ also $\sigma \approx \sigma'$. Therefore *serv* is visibility preserving w.r.t. $\sim$ and $\approx$.

Second we show non-interference.

Select $\sigma_1, \sigma_2, \sigma_1', \sigma_2', t_1, t_2$ arbitrarily such that $\sigma_1 \approx \sigma_2$ and $\langle \mathit{handler}_{serv}; \sigma_1\rangle \xrightarrow{t_1} \langle \mathit{SKIP}; \sigma_1'\rangle$ and $\langle \mathit{handler}_{serv}; \sigma_2\rangle \xrightarrow{t_2} \langle \mathit{SKIP}; \sigma_2'\rangle$

Now assume $t_1 \rhd \mathbb{I} \ \sim \ t_2 \rhd \mathbb{I}$, which, by definition of $\sim$ means $t_1 \rhd \mathbb{I} \ \sim_1 \ t_2 \rhd \mathbb{I}$ and $t_1 \rhd \mathbb{I} \ \sim_2 \ t_2 \rhd \mathbb{I}$.

Since *serv* is non-interferent w.r.t. $(\sim_1, \approx_1)$ and $(\sim_2, \approx_2)$, we know that $\sigma_1' \approx_1 \sigma_2'$ and $\sigma_1' \approx_2 \sigma_2'$, and therefore by definition of $\approx$ also $\sigma_1' \approx \sigma_2'$.

With an argument of the same structure, we also get $\forall t_1' \leq t_1, t_2' \leq t_2 \cdot \ t_1' \rhd \mathbb{I} \ \sim \ t_2' \rhd \mathbb{I} \implies \exists t_1' \frown t_1'' \leq t_1, t_2' \frown t_2'' \leq t_2 \cdot t_1' \frown t_1'' \ \sim t_2' \ \frown t_2'')$

And therefore $d_1 + d_2$ is a Dependency Cluster for *serv*. $\lhd$

The composition of the specifications in Example 4.3 is again the Dependency Cluster indexed by 1 from our initial example in Example 4.2.

According to Theorem 4.1, it is sufficient to show two Dependency Clusters independently in order to gain a composed, potentially more complicated, Dependency Cluster. Ideally, it is possible to identify simple Dependency Clusters (in the sense of an applied analysis method) which describe information flows inherent to the implementation of a service. More complicated clusters, which are usually necessary to compare information flow to an attacker-motivated security policy, can be generated by composing these simple Dependency Clusters.

The composition of two Dependency Clusters is a weaker non-interference specification than requiring the service to be non-interferent w.r.t. both Dependency Clusters.

Example 4.5. Assume the state equivalence specifications $\mathit{LowState}_a := \langle a\rangle$ and $\mathit{LowState}_b := \langle b\rangle$ and their composition $\mathit{LowState}_{ab} := \langle a, b\rangle$ The program $a := a + 1; b := b + b$ produces equivalent poststates, if executed in equivalent prestates all three Dependency Cluster. However the program $a := a + b$ satisfies the specification $\mathit{LowState}_{ab}$, but not $\mathit{LowState}_a$.

Since Dependency Clusters are properties only depending on services, they hold independent from the environment a DSC is deployed in and other services interacting on the same state.

4.4 Dependency Clusters and Components

We can use Dependency Clusters for services to abstractly describe dependencies of information caused by the execution of a service. However, we cannot directly use Dependency Clusters for services to describe the flow of information caused by the execution of a DSC, i.e. arbitrary sequential compositions of services provided by a DSC. Definition 3.9 states that we require an equivalence relation such that all services are non-interferent w.r.t. this DSC-global equivalence relation. Dependency Clusters for a service, however, do not consider all state variables of the state or even the state variables used by the service. Similarly, we require an equivalence relation over messages for all services provided by a DSC. A Dependency Cluster does not necessarily provide us with the information if high information is provided as a parameter during a service call.

Example 4.6. See for example the Dependency Clusters we defined so far for the service `pay` in Example 4.2 and Example 4.3. None of the Dependency Clusters mention the state variable `countbuy`. In this case the reason is that this variable is not mentioned in the code of the service, it is never involved in any calculation and therefore its classification is irrelevant for the service. Practically, it would be surprising, if a variable not mentioned in the implementation of a service would be mentioned in a service-local, attacker-independent specification. Also the variable `prodprice` is not mentioned. In this case, however, the reason is that there was no Dependency Cluster identified in which the variable was specified to contain low information. The variable is involved in other flows caused by the service.

Similarly, a Dependency Cluster may not mention a service for the same two reasons. And for the same two reasons, maybe respective messages are irrelevant for the service implementation or it is just not mentioned in the specification.

A typical approach in program analysis to deal with irrelevant parts of states, and similar with irrelevant services required by a DSC, is called framing (Kassios [2006]). Framing uses an abstract description of an upper bound of relevant variables and the services required by a particular service. An *assignable set* describes the variables which may at most be changed by a service and a *callable set* for a service describes the services that are at most required by the service. The assignable set indirectly describes for all variables not in the set that the content of the variable, and therefore the security level of their contents, is not changed.

Definition 4.2 (Assignable Set for Services)**.** $\mathbb{F} \subseteq \mathcal{V}$ is an assignable set for a service *serv*, iff
$\forall \sigma, \sigma', t \cdot \langle handler_{serv}; \sigma\rangle \xrightarrow{t} \langle SKIP; \sigma'\rangle \implies \forall v \in \mathcal{V} \cdot v \in \mathbb{F} \vee \sigma(v) = \sigma'(v)$

When making arguments about pre- and poststates of a service with a given assignable set, we use the anonymization function $anon : \mathbb{S} \times \mathcal{P}(\mathcal{V}) \times \mathbb{S} \mapsto \mathbb{S}$. The function $anon(\sigma, V, \sigma')$ yields a state σ_{anon} such that for all variables v $\sigma_{anon}(v)$ evaluates to $\sigma'(v)$ if $v \in V$ and to $\sigma(v)$ otherwise.

Similar to the assignable set, a *callable set* specifies a list of services which can at most be called by a service. Every service not in this set is irrelevant for the information flow caused by the service.

Definition 4.3 (Callable Set for Services)**.** $\mathcal{C} \subseteq \mathcal{S}$ is a callable set for a Service *serv*, iff
$\forall \sigma, \sigma', t, t', m \cdot \langle handler_{serv}; \sigma\rangle \xrightarrow{t \frown m \frown t'} \langle SKIP; \sigma'\rangle \implies$
$((\exists v \in \mathbb{D}, c \in \mathbb{C} \cdot m = c!v \wedge c = Ini(serv')) \implies serv' \in \mathcal{C})$

Example 4.7. In our example, an assignable set for *pay* is $\langle countcheck\rangle$, for *buy* it is $\langle product, prodprice, prodamount, countbuy\rangle$. A callable set for *pay* is $\langle registerSales\rangle$ and for *buy* it is the empty set.

We can use Dependency Clusters for services, combined with assignable and callable sets to check for a potential DSC-global non-interference specification $(\sim_g, \approx_g)$, if $serv \in \mathtt{SNI}^{\approx_g}_{\sim_g}$, solely based on known Dependency Clusters of the service.

Theorem 4.2. *Let $\mathcal{C}$ be a callable set for service serv and $\mathbb{F}$ an assignable set for serv. A tuple $(\sim_g, \approx_g)$ is a Dependency Cluster for serv if there exists a Dependency Cluster $(\sim_{serv}, \approx_{serv})$ for serv, such that*

$$\begin{aligned}
&\forall m, m' \cdot m \sim_g m' \implies m \sim_{serv} m' && (4.1)\\
&\wedge \forall \sigma, \sigma' \cdot \sigma \approx_g \sigma' \implies \sigma \approx_{serv} \sigma' && (4.2)\\
&\wedge \forall m, m'(m \sim_{serv} m' \wedge m \in \mathcal{C}) \implies m \sim_g m' && (4.3)\\
&\wedge \forall \sigma, \sigma', \sigma_p, \sigma'_p \cdot (\sigma \approx_g \sigma' \wedge \sigma_p \approx_{serv} \sigma'_p)\\
&\quad \implies anon(\sigma, \mathbb{F}, \sigma_p) \approx_g anon(\sigma', \mathbb{F}, \sigma'_p). && (4.4)
\end{aligned}$$

Proof. Assume we have $\sim_g, \approx_g$ and $\sim_{serv}, \approx_{serv}$, such that conditions 4.1 to 4.4 hold.

We first show that *serv* is visibility-preserving with respect to $\sim_g, \approx_g$. Let σ, σ_p, t, t' such that $\langle handler_{serv}; \sigma\rangle \xrightarrow{t \frown t'} \langle SKIP; \sigma_p\rangle$, $t \rhd \mathbb{I} \sim_g \langle\rangle$. So, for all input messages m in t it holds that $m \sim_g \square$ and since condition 4.1 holds, it also holds $m \sim_{serv} \square$, and therefore $t \rhd \mathbb{I} \sim_{serv} \langle\rangle$. Since $(\sim_{serv}, \approx_{serv})$ is Dependency Cluster for *serv*, we know $t \rhd \mathbb{O} \sim_{serv} \langle\rangle$. Since $\mathcal{C}$ is callable set for *serv*, we know for all output messages m in t by Definition 4.3 and Theorem 4.2, condition 4.3 we also know $m \sim_{serv} \square \implies m \sim_g \square$ and therefore $t \rhd \mathbb{O} \sim_g \langle\rangle$.

Further, we know $\sigma \approx_{serv} \sigma_p$ and since $\mathbb{F}$ is an assignable set, we know by Definition 4.2 that $\sigma_p = anon(\sigma, \mathbb{F}, \sigma_p)$.

Since $\approx_g$ is an equivalence relation, we know $\sigma \approx_g \sigma$ and since $(\sim_{serv}, \approx_{serv})$ is a Dependency Cluster for *serv*, we know $\sigma \approx_{serv} \sigma_p$. With Theorem 4.2 condition 4.4, we get $anon(\sigma, \mathbb{F}, \sigma) \approx_g anon(\sigma, \mathbb{F}, \sigma_p)$, i.e. $\sigma \approx_g \sigma_p$.

Now we show equivalence of the poststates: Assume $\sigma \approx_g \sigma'$ and furthermore $\langle handler_{serv}; \sigma\rangle \xrightarrow{t} \langle SKIP; \sigma_p\rangle$ and $\langle handler_{serv}; \sigma'\rangle \xrightarrow{t'} \langle SKIP; \sigma'_p\rangle$. Due to Theorem 4.2, 4.2, also $\sigma \approx_{serv} \sigma'$ and since $(\sim_{serv}, \approx_{serv})$ is a Dependency Cluster for *serv*, also $\sigma_p \approx_{serv} \sigma'_p$ holds. Since $\mathbb{F}$, we know due to Definition 4.2 that $\sigma_p = anon(\sigma, \mathbb{F}, \sigma_p)$ and $\sigma'_p = anon(\sigma', \mathbb{F}, \sigma'_p)$ and therefore with Theorem 4.2 condition 4.4 we know $\sigma_p \approx_g \sigma'_p$.

Finally we show equivalence of the communicated traces: Assume $t_1 \leq t$ and $t'_1 \leq t'$ such that $t_1 \rhd \mathbb{I} \sim_g t'_1 \rhd \mathbb{I}$. Due to Theorem 4.2 condition 4.1, it also holds that $t_1 \rhd \mathbb{I} \sim_{serv} t'_1 \rhd \mathbb{I}$ and since $(\sim_{serv}, \approx_{serv})$ is Dependency Cluster for *serv*, there exists t_2, t'_2 such that $t_1 \frown t_2 \leq t$ and $t'_1 \frown t'_2 \leq t'$ and $t_1 \frown t_2 \sim_g t'_1 \frown t'_2$. Since $\mathcal{C}$ is a callable set for *serv*, we know according to Definition 4.3 for all m, m' in $t_1 \frown t_2$ and $t'_1 \frown t'_2$ respectively $m, m' \in \mathcal{C}$ and by Theorem 4.2 condition 4.3 we know $m \sim_{serv} m' \implies m \sim_g m'$ and therefore $t_1 \frown t_2 \sim_g t'_1 \frown t'_2$.

So combined, $(\sim_g, \approx_g)$ is Dependency Cluster for *serv*. $\lhd$

Condition (4.1) states that input messages which are equivalent w.r.t. the DSC-global equivalence relation, are also equivalent w.r.t. the service-local equivalence relation. Condition (4.2) ensures that if two states are equivalent w.r.t. the global state equivalence relation, then they are also equivalent w.r.t. the service-local relation. Indirectly, this ensures that if all other services provided by a DSC ensure equivalence w.r.t. the global equivalence relation for their poststate, then *serv* is guaranteed to be executed in prestates which are equivalent w.r.t. the service-local specification.

Condition (4.3) states that it is guaranteed that all output messages of a service are considered equivalent service-locally if they are also globally equivalent. However, the service-local equivalence relation may be more relaxed for outputs which are not generated by the service. In a similar fashion, the service guarantees in condition (4.4) that the part of the state, which is actually changed by the service, is changed such that this part is also equivalent w.r.t. the DSC-global state-equivalence relation.

Example 4.8. Remember the Dependency Cluster indexed by 1 in Example 4.2. We know this Dependency Cluster is, according to Theorem 4.1 a Dependency Cluster for `pay`, since it is the intersection of the Dependency Clusters indexed by 2 and 3 (see Example 4.3).

Assume the following Dependency Cluster, which adds specifications for the service `buy`.

$$\begin{aligned}
\mathit{Vis}_4 :=\ & \langle \mathit{Ini}(\mathit{pay}).(\mathit{true}),\ \mathit{Fin}(\mathit{pay}).(\mathit{true}),\\
& \mathit{Ini}(\mathit{registerSales}).(\mathit{true}),\ \mathit{Fin}(\mathit{registerSales}).(\mathit{true}),\\
& \mathit{Ini}(\mathit{buy}).(\mathit{true}),\ \mathit{Fin}(\mathit{buy}).(\mathit{true})\rangle\\
\mathit{LowIO}_4 :=\ & \langle \mathit{Ini}(\mathit{pay}).(\mathit{ccnr}),\ \mathit{Fin}(\mathit{pay}).(0),\\
& \mathit{Ini}(\mathit{registerSales}).(\mathit{prodId}, \mathit{ccnr}),\\
& \mathit{Fin}(\mathit{registerSales}).(\mathit{0}),\\
& \mathit{Ini}(\mathit{buy}).(\mathit{prod}),\ \mathit{Fin}(\mathit{buy}).(\mathit{true})\rangle\\
\mathit{LowState}_4 :=\ & \langle \mathit{product}, \mathit{countbuy}\rangle
\end{aligned}$$

Additional to the specifications for `pay`, this specification states that the parameter `prod` of `buy` is low, as is the variable `countbuy`. While the specification of `countbuy` actually is irrelevant for service `pay` (i.e. neither is it required to contain low information by `pay`, nor is it in the assignable set such that `pay` might store high values in the variable), in this Dependency Cluster the specification states that all values stored in `product` has to only contain low information.

We can use Theorem 4.2 to show for service `pay` that it is non-interferent w.r.t. this specification. Since we know that `pay` is non-interferent w.r.t. the Dependency Cluster indexed by 1 and we know by specification the assignable set and the callable set for the service, no program analysis is necessary, but a first order logic verification is sufficient.

Theorem 4.2 allows us to check non-interference of a service against a DSC-global non-interference specification only using specifications of the service. Practically, we first have to find a DSC-global specification. One way to achieve this is to use relevant Dependency Clusters of all services provided by a DSC and intersect them. As a result, we get a non-interference specification for the entire DSC, and we have to check the condition from the theorem for each service. An example how this can be applied is discussed in Chapter 8.

Theorem 4.2 makes Dependency Cluster especially useful for evolving DSCs. If a DSC is deployed into a new context, i.e. the considered attackers change, also the domain-motivated security specification changes. This might cause changes to the implementation of single services, and only these changed services have to be re-verified against new Dependency Clusters. All Dependency Clusters of the other services are still valid and we can re-use them for construction of relevant information flow specifications. If the implementation of services is only optimized, i.e. the actual behavior of a service is unchanged and only the implementation is changed, only the already existing Dependency Clusters of the changed service has to be re-verified. The composition of Dependency Clusters still is valid.

Example 4.9. We consider here two use cases. In the first case, assume the *Cart* DSC should be re-used in a new context. Due to a changed attacker model, it is required that only the last four digits of the credit card number are low, instead of the entire credit card number. To realize this, the service *pay*, implemented as in Example 2.3 has to be edited and instead of providing the parameter *ccnr* to the service `registerSales`, the respective code is changed:

```
temp = ccnr - (ccnr/10000)*10000;
registerSales(product, productprice, productamount, temp)
```

Where `temp` is a local variable. Since the code has changed, the Dependency Clusters for *pay* have to be re-verified. We also require a new Dependency Cluster for `pay` which expresses that only the last four digits are low. However the Dependency Clusters for the other services can be re-used without program analysis when they are verified against a new DSC-global tuple $(\sim_g, \approx_g)$ according to Theorem 4.2.

In a second case of evolution, we assume that the DSC is used in the same context as in the previous example, but the implementation should be optimized with the same functionality. The line we just added in *pay* above is replaced by

```
temp = ccnr%10000
```

This time, we do not need to specify a new Dependency Cluster, since the change was only an optimization which did not change the observable behavior of the service. However, since the code has changed, the Dependency Clusters for *pay* have to be re-verified. The Dependency Clusters for all other services are still valid by definition, and even $(\sim_g, \approx_g)$ is still valid, such that we do not require program analysis for services that are unchanged or a new proof according to Theorem 4.2.

4.5 Weakening Specifications

The equivalence relations on inputs and outputs which we gain by analyzing dependencies in services as shown above do not necessarily match a security policy for a DSC provided by a domain expert for the system under analysis. The specification provided by a domain expert will mainly be motivated by the attacker model gained from a threat analysis of a system.

As a result, a domain expert may decide that certain input is not necessarily of high confidentiality, although the implementation of the DSC does not provide this information in any way as an attacker-observable output. At the same time, the domain expert may decide that some output is not accessible directly or indirectly by a possible attacker and therefore may hold high information, although the DSC does not actually provide any secret information via the respective channel.

In this case, the equivalence relation we gain from composing Dependency Clusters does not exactly match the equivalence relation required by specification, but is more strict than actually necessary. We can relax an equivalence relation over messages by allowing to accept low input, although high input is expected, and we can allow the environment to treat low output of the DSC as high output. We call an equivalence relation $\sim_w$ with this property a *weakening* of $\sim$.

Definition 4.4 (Specification Weakening)**.** An equivalence relation $\sim_w$ is a weakening of $\sim$, iff

$$\begin{aligned} & m_1 \sim_w m_2 \ \wedge\ m_1, m_2 \in \mathbb{I} && \implies m_1 \sim\ m_2 \\ \wedge\ & m_1 \sim\ m_2 \ \wedge\ m_1, m_2 \in \mathbb{O} && \implies m_1 \sim_w m_2 \end{aligned}$$

Example 4.10. Assume the DSC `Cart` from Example 2.3 with the changes made in Example 4.9 is deployed in a an environment. The domain expert may provide us with an attacker-motivated specification expressing that the cashier may know the last five digits of the credit card number. The specification we gained from bottom-up program analysis, however, provides us with a more strict specification allowing at most the last four digits to be visible to the cashier. In this case, we can not directly use our bottom-up specification as an argument for security against this attacker. However, the attacker-motivated specification is a weakening of the bottom-up specification.

If a service is non-interferent w.r.t. a specification $(\sim, \approx)$, then it is also non-interferent w.r.t. any weakening of $\sim$.

Theorem 4.3. *Let serv be a service with serv* $\in \mathit{SNI}_{\sim}^{\approx}$ *and* $\sim_w$ *a weakening of* $\sim$*. Then serv* $\in \mathit{SNI}_{\sim_w}^{\approx}$*.*

Proof. Follows directly from Definition 3.12. We strengthen the left hand side of the inner implication and weaken the right hand side. ◁

On first sight Theorem 4.3 seems to be a technicality. However, the theorem serves as an important connection between bottom-up specifications, which Dependency Clusters are, and top-down specifications, gained from context- and attacker-motivated analysis. It frees the system engineer from finding non-interference specifications for already implemented DSCs which exactly fit the domain-driven idea of secrecy. Thus it serves as a glue which allows flexibility when bringing together domain expertise and context-independent program analysis.

Theorem 4.3 can easily be extended to DSCs. If all services are non-interferent w.r.t. $(\sim, \approx)$, they also are non-interferent w.r.t. $(\sim_w, \approx)$ and therefore the DSC is non-interferent w.r.t. $(\sim_w, \approx)$ according to Definition 3.9. We can verify with this theorem that our evolved `Cart` DSC is secure against the attacker, although the required and verified information flows differ.

4.6 Conclusion

In this chapter we introduced the notion of Dependency Clusters which describe ideally small dependencies between inputs, outputs and parts of a state. Dependency Clusters can be used as building blocks to gain complex non-interference specifications for services, DSCs, and compositions. Since bottom-up non-interference specifications, which Dependency Clusters are, most likely do not exactly match a domain-driven non-interference specification, we provided the weakening theorem which states under which conditions a bottom-up non-interference specification implies the domain-driven security specification.

Program analysis techniques are only required to show that a non-interference specification is a Dependency Cluster for a service, while combining Dependency Clusters either comes for free (within a service), or only FOL conditions have to be shown to be valid (within components and for the domain-driven non-interference specification).

Dependency Clusters have two major advantages. For one, we have illustrated that Dependency Clusters lead to robust non-interference specifications in the case of a changing environment or when DSCs evolve.

Second, Dependency Clusters are also useful to combine different program analysis tools. For example in previous publications (Greiner et al. [2017b] and Greiner et al. [2017a]) we combined a fully automatic tool with a tool for deductive program verification to identify simple Dependency Clusters for all services in a system. For more complicated Dependency Clusters, i.e. Dependency Clusters using some form of declassification, we used a program analysis tool based on deductive verification, which requires manual interaction, but is very precise. We discuss the interactive tool in detail and the combination of the tools for a case study in Chapter 8.

The automatic tool alone would not have been able to verify the system as secure due to declassification. With the interactive tool alone verification would have been a very laborious. However the combination allowed the verification with a reasonable effort.

5 Related Work

Non-interference in general is a well-researched security property. The origins go back to *strong dependency* by Cohen [1977] and the definition of non-interference by Goguen and Meseguer [1982]. State-of-the-art research, related to the results presented in the first part of this thesis, can be separated into (1) non-interference for interactive programs exchanging parametrized messages with its environment, where secrets are inputs and outputs during the run of a program; and (2) non-interference for batch-programs, where partly high and partly low input is provided as a state at the start of the program and the security property has to hold in the state after termination of the program.

5.1 Non-Interference in Interactive Programs

Work on non-interference for programs with intermediate communication with its environment is manifold. Non-interference is discussed using event systems, for example by Mantel and Sabelfeld and Mantel [2002], or process calculi, e.g. Focardi and Gorrieri [1994], Ryan and Schneider, and Pottier. In both contexts, the environment is not modeled explicitly, but as traces or streams of input and output events.

The work closest to ours uses labeled transition systems as representations of programs and explicitly takes the environment of a program in the form of strategies into consideration. Modeling the environment by using strategies was pioneered by Wittbold and Johnson. Here, the environment is separated into high and low users of a system, each modeled as a strategy, and providing high and low input respectively. O'Neill et al. present a formal analysis of non-interference for interactive programs in the presence of strategies. Clark and Hunt [2009] show that for deterministic programs, it is sufficient to model the environment as input-streams. Input-streams, in contrast to strategies, do not take the observation of an execution of the program into consideration, i.e. all input is predetermined.

Rafnsson et al. add an additional dimension to the specification of parameters of messages as low and high type. Rafnsson et al. [2012] define the presence of messages as a possible secret, which leads to strategies, which are able to block program execution by not providing further input on a channel. The resulting non-interference property is very restrictive. Nearly all programs receiving intermediate secret messages as inputs followed by low outputs are considered insecure according to this property. Take, for example the program $\texttt{read}(x \leftarrow \alpha)\texttt{;}\texttt{read}(y \leftarrow \beta)\texttt{;}\texttt{write}(1 \rightarrow \gamma)$, where an event on channel γ reveals that an event on channel β previously was provided to the program. If the presence of events on channel β is high, the observable communication has to be equivalent, independent from communication on this channel.

By using cooperative environments, we make the non-interference property more applicable in cases where we have some reason to assume a cooperative environment. More concretely, we employ contracts which are assumed in component-bases system engineering to hold for every environment the component is deployed in. While the call of a service is still considered a possible secret, the termination of a called service is not and therefore the program can not be blocked by receiving the service termination as a high input. After composition of DSCs, the fact that a service is called still is a secret and can not be observed by an attacker.

Further, we extend the work by Rafnsson et al. [2012] with declassification of information, according to Sabelfeld and Sands [2009] in the *what-dimension* by using equivalence relations for the specification of secret information instead of previously used type systems. This extension allows us to specify parts of communicated parameters as high and low and preserve this specification over interface boundaries between components. Our extension is a generalization of the previously used three dimensions of high and low for communicated content and the secrecy of the existence of a message. We can express specifications according to the type system proposed by Rafnsson et al. with our equivalence relations. A channel H of type *high*, specifying that the existence of messages on H is high, can be modeled with the equivalence relation $H.v \sim \square$ for all v; a medium channel M specifying that the existence of a message on M is low, while the content is high is modeled by $M.v \nsim \square$ for all v and $M.v \sim M.v'$ for all v, v'; and a low channel L, where the message's existence as well as the content is low is specified by $L.v \nsim \square$ for all v and $L.v \sim L.v'$ if $v = v'$. Nevertheless, the compositionality proof performed by Rafnsson et al. [2012] does not apply in our more general case, so we provide a new compositionality proof for Theorem 3.1.

Vanhoef et al. provide a similar notion of declassification which allows declassification of partial information using a *project* function for specification. Vanhoef et al. additionally allow the declassification of aggregated information over a history of events, making the declassification policy stateful. Their work builds on results by Sabelfeld and Sands [2001], who allow the specification of information flow properties using partial equivalence relations for sequential batch programs. For enforcement of the policy, they propose a dynamic approach based on secure multi-execution, but do not provide a compositionality result for their non-interference property.

In contrast to Vanhoef et al. we aim for re-usability of components in different contexts, possibly with several different security lattices. In this setting a dynamic enforcement of non-interference using secure multi-execution is not practical, since we expect the cost of multi-execution to be too high in this case. We aim for a static and reusable analysis of components, which makes a compositional non-interference property necessary. Therefore, we prove compositionality for our extension of the strategy-based approach by Rafnsson et al. In contrast to Sabelfeld and Sands, we consider interactive programs instead of batch programs.

Clark and Hunt [2009] show that for deterministic LTS a strategy-based notion of non-interference is equivalent to a stream-based non-interference notion, if the specification is limited to high and low channels. Rafnsson et al. [2012] show that this is also the case for deterministic LTS, if additionally the specification of the presence of messages as high or low is allowed. We assume that noth non-interference notions are also equivalent if equivalence relations are allowed for specification, as presented here, although, we do not provide a proof. Nevertheless, we presented non-interference for components using a strategy-based notion. For one, we consider this strategy-based notion to be more intuitive when communicating security properties of systems and components to stakeholders, which do not have a formal security background. Second, we find the idea of cooperative environments, which we require for cooperative non-interference for components, to be easier to comprehend than abstract limitations of potential streams to be considered.

5.2 Non-Interference in Batch Programs

When considering compositionality of services, we extend non-interference for batch programs with intermediate message passing. A discussion on non-interference in batch programs can be found in Barthe et al., Joshi and Leino [2000], Amtoft and Banerjee [2004], and Darvas et al.. Here two runs of a program started in two equivalent, but underspecified, initial states are compared. The program is secure, if the terminal states of both runs are equivalent with respect to some specification of secrets in the state.

Sabelfeld and Sands [2001] propose the PER model for information declassification using partial equivalence relations, but without interactive message passing with the environment. Sabelfeld and Myers [2004] introduce escape hatches as a specification mechanism for precise semantic declassification designed for analysis of batch programs without message passing using type systems. Amtoft and Banerjee [2007] present a notion of non-interference for batch programs where state equivalence is specified using an agree operator. Two states are equivalent, if they agree on on a set of expressions, i.e. if the expressions evaluate to the same values in both states. This technique allows conditional declassification of information, and also supports different equivalence relations over states in the pre- and the poststate.

We disallow by definition different equivalence relations for the pre- and the poststate for service non-interference in order to ensure that non-interferent services are sequentially compositional. We extend the common notion of non-interference for batch programs with intermediate event communication and we relate it to non-interference for interactive programs by providing a compositionality result for components, which states that non-interferent batch programs result in non-interferent interactive programs in the case of components.

5.3 Rely-Guarantee Style Non-Interference

Rely-Guarantee style non-interference is a technique where parts of a program are analyzed individually while relying on some assumption on the environment of the partial program. The analysis then guarantees a non-interference property for the partial program, if it can rely on the assumption made. For composing several partial programs it has to be checked whether assumptions of partial programs are consistent with guarantees provided by the other programs it is composed with. Our non-interference property for LTS and DSC can be considered as a rely-guarantee approach in that a DSC assumes other components in the environment to be non-interferent w.r.t. the non-interference specification.

Traditionally, rely-guarantee style non-interference is used for modular non-interference properties for concurrent programs communicating over a shared state, e.g. multi-threaded programs with each thread being considered a partial program. Mantel et al. [2011] propose an approach where assumptions and guarantees express for threads whether or not the access state variables by writing them or reading from them. Murray et al. [2016] extend the approach to allow state-dependent assumptions for shared resources, i.e. their work allows assumptions like a variable x may be accessed, if another variable y has the value 0.

5.4 Compositional Specifications

Composing specifications is a frequently used technique in functional verification. For example the conjunction and disjunction rules (Reynolds [1982]) in Hoare logic (Hoare [1969]) allow a conjunctive and disjunctive composition of Hoare triples, i.e. specifications. A common application of this rule can be found in design by contract approaches. For example, the specification language JML uses the `also` keyword to specify several pre- and postcondition tuples for one method. Analysis can be limited to each tuple individually, while the combination of the contracts can be used when applying the contract. Composing functional contracts is also useful in the context of class inheritance.

Separation logic (Reynolds [2002]) is a technique for reasoning about local changes on the state caused by a program while gaining global properties. This can be seen as another application of composing specifications for programs. The frame rule allows to separate parts of pre- and postconditions of a program into one part which is modified by the program and one part which is not modified. We use a similar technique for Theorem 4.2.

Our framework provides the first notion of non-interference for services where the specification is compositional. In fact, we make heavy use of composing non-interference specifications when composing Dependency Clusters and thus gain DSC-wide and system-wide non-interference specifications.

Framing as an abstract specification for relevant parts of a state w.r.t. a program (i.e. service) was introduced by Kassios [2006]. We use their result for consistency checks when composing non-interferent services to non-interferent DSCs. Since framing allows us not to require a specification of every variable on the state, we gain a more flexible condition to be checked and we even gain a first-order logic condition which does not require program analysis for consistency checks.

6 Conclusion

In the first part of this thesis, we formalized components as DSCs, services, and compositions. Further, we defined a non-interference property for DSCs using an explicitly model for the environment a DSC runs in. Building on these definitions, we introduced Dependency Cluster as modular building blocks for specification and verification of non-interference in DSCs. Figure 6.1 illustrated how the different contributions relate to each other.

In Chapter 2 we provided a formalization of components as DSCs. A DSC is an LTS which provides its functionality as services to its environment, ensures that services are executed sequentially and all services terminate. DSCs can be composed to compositions which communicate with each other via message passing. We deliberately use the abstract notion of LTS to describe DSCs in order to provide results independent from concrete programming languages and technologies for implementing distributed systems.

In Chapter 3 we discussed non-interference for DSCs from a conceptual point of view. Specification of high and low input and output information of a DSC or composition is given in the form of equivalence relations, which makes the specification language expressive. It allows nearly arbitrary declassification of input information either encoded in parameters of services or in the existence of messages. We use strategies as an explicit model of the environment a DSC runs in. Strategies allow an intuitive understanding of the relation between non-interference specifications and security properties. A potential attacker is a part of the environment who can observe the low output information and may know the low input information. A DSC is non-interferent if an attacker can not distinguish between two environments providing the same low input.

We formally state this intuitive definition of non-interference in Definition 3.6 for interactive systems in general. Thus, we provide the first compositional non-interference notion for interactive programs which supports what-declassification and specification of message existence as low information. In Definition 3.9 we provide a non-interference notion specially

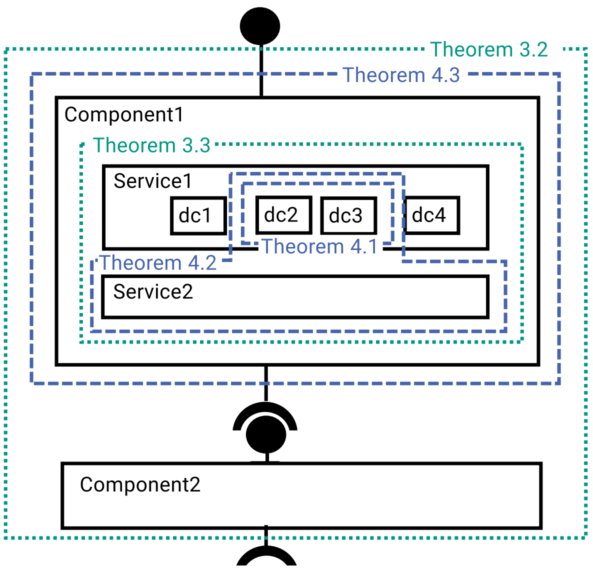

Figure 6.1: Definitions and Theorems in the Framework (green, dotted: conceptual; blue, dashed: constructive)

designed for components, i.e. DSCs. By making the assumption that the environment adheres to basic properties which are common in component-based system engineering, i.e. all called services terminate and the environment does not leak high information, we gain a more precise notion of non-interference. In Theorem 3.2 we show that non-interference for DSCs is compositional, an important property for component-based systems.

In Definition 3.12 we combine non-interference for batch programs and interactive programs to gain a novel non-interference notion for services. We assume services to be easier to analyze since they are in general rather simple programs. We show that it is sufficient to show that all service provided by a DSC are non-interferent in order to show that the entire DSC is non-interferent (Theorem 3.3).

In Chapter 4 we take a more constructive point of view on non-interference specification and analysis. While in Chapter 3, we provide a top-down approach, i.e. given a non-interference specification, what does it mean for a DSC to be non-interferent, in Chapter 4 we discuss a bottom-up approach, i.e. given a component what information does depend on each other, and how can a security guarantee against an attacker be constructed from this information.

We introduce Dependency Clusters as a novel concept for modular descriptions of information dependencies in services, DSCs and compositions. Dependency Clusters are compositional specifications (Theorem 4.1) in the sense that we can combine different Dependency Clusters of a service and gain a new specification of information dependencies without requiring an additional program analysis. Whether or not a specification is a Dependency Cluster for a service only depends on the implementation of the service, in particular it is independent from other services provided by a DSC or the environment a DSC runs in. As a result Dependency Clusters are robust specifications for evolving components and components deployed in new environments. Dependency Clusters, as non-interference specifications in Chapter 3, support expressing what-declassification and dependency on message existence.

Furthermore we show that we can use Dependency Clusters to build non-interference specifications for DSCs. After selecting relevant Dependency Clusters of different services provided by a component, only a rather simple FOPL property has to be shown to be valid Theorem 4.2. Dependency Clusters can be used as building blocks for complex non-interference specification for DSCs, while program analysis techniques are only necessary to analyze services. Compositionality is shown without program analysis.

A bottom-up approach, which is supported by Dependency Clusters, most likely does not lead to a non-interference specification which exactly expresses security against an attacker, for example identified by a domain analysis. We therefore show in Theorem 4.3 that it is sufficient to show for a bottom-up specification a FOPL condition to show that it is compatible against an attacker-motivated specification and therefore the DSC is secure against this attacker.

The framework as described in the first part of this thesis is meant to serve as a basis for specification approaches and program analysis techniques for practically used technologies for distributed systems. We show in the second part of the thesis how our framework can be used to design a specification language for non-interference of systems integrated in a graphical specification language for distributed systems. We also show that based on our framework we can implement program analysis tools for distributed systems implemented in the object-oriented programming language Java.

Part II

Instantiating the Framework

7 Model-Based Non-Interference Specification

In this chapter, we instantiate the framework from the first part of this thesis as a specification language for information flow properties in a model-based system-engineering process.

The term model, as used in this chapter, refers to a graphical and intuitive representation of a system under design. A model typically has different stakeholders at different phases of a system design process. During requirements elicitation the model serves as a communication tool between the system designer and domain experts As a result of this phase, the model makes requirements for a system explicit and ideally unambiguous. The system architect describes in the model the internal architecture of the system and thus documents which components and compositions are responsible for realizing parts of the functionality of the overall system. During implementation, the model provides the requirements for each component which allows the programmer to implement the functionality of a single component without being distracted from overall system requirements. During quality assurance, the model provides black-box specifications for components and compositions against which integration tests can be written.

A model describes one system throughout the entire development process. For each stakeholder *views* show a particular part of the entire model such that the focus is on the information necessary this stakeholder, while irrelevant information is hidden. Thus, the model can describe the same subsystem as black-box or white-box, depending on the purpose of a particular view.

Several properties of our framework as described in Part I are useful in this scenario. For one, we can describe non-interference of a subsystem from a black-box point of view, since the information flow specification $\sim$ only considers inputs and outputs of a system. Thus, the specification does not require information about the internal workings of the subsystem. Second, a non-interference specification $\sim$ directly provides a specification for a single component, which can be used by a programmer directly as a requirement for the component he is about to implement. Third, the compositionality properties of our non-interference notion allows a system architect to decide early in the development process whether a particular architecture describes an overall secure system.

In this chapter, we describe a graphical specification language for non-interference properties for component-based systems as an extension of the Palladio Component Model (PCM). By giving formal semantics to the graphical representation of the system in the model, we show that the properties of non-interference (e.g. compositionality) also hold for non-interference specifications in the model. In the next section, we introduce the Palladio Component Model as far as needed in the context of this chapter and show how formal constructs from Part I map to elements in a model. In Section 7.2, we give a brief introduction of an extension of the PCM for security specifications in which non-interference specifications are embedded. In Section 7.3, we discuss syntax and semantics of non-interference specifications in the PCM in detail. We also show in Section 7.3 that non-interference as specified in a PCM model is compositional. After this, we present related work and finally conclude.

The results presented in this chapter are an extension of parts of the work by the author in previous publications (Kramer et al. [2014] and Kramer et al. [2017]).

7.1 Palladio

"Palladio is a software component modeling approach that focuses on the prediction of quality attributes of a software architecture" (Reussner et al. [2016]). The approach provides the Palladio Component Model, a meta model which describes the syntax of a graphical specification language for software architectures, and the deployment of software to hardware. Additionally, the PCM provides structures for modeling software behavior and tools for analysis of expected response time, throughput, resource utilization, and others for a given model of a component-based system. The approach aims for early analysis of quality of service properties of software architectures before they are actually implemented and deployed.

In this section, we introduce the meta-model PCM, which describes the syntax of the graphical specification language. The PCM defines model elements that can be used in Palladio and how they may be related with each other. We concentrate on components in the PCM. For further information on behavior modeling, deployment and other topics, we refer the interested reader to Reussner et al. [2016].

The presentation of the PCM in this section is based on Reussner et al. [2016], and we additionally provide a formalization of the elements using the notion from Part I in order to make clear how PCM components and DSCs as defined in Chapter 2 are related.

7.1.1 Meta Model in Palladio

The PCM is a meta model and thus describes all possible models that can be created using the PCM. We call an instantiation of the PCM, i.e. an actual model, a PCM model. The central entity of a PCM model is the *repository*, which contains *Datatypes*, *Interfaces* and *Components*. Components are abstract units encapsulating functionality and are subject to composition. *Datatypes* are used as types of parameters for interfaces, while an interface describes a set of functionalities, which can then be used by components. *Interfaces* are "abstract descriptions of units of software. They can be used as points of interaction between components"(Reussner et al. [2016]). According to the PCM, an interface has a *name* and contains a list of service signatures. Each *signature* has a *name*, a return data type and a list of *parameters*, each again having a name and a data type.

A *Component* "is a contractually specified building block for software, which can be composed, deployed, and adapted without understanding its internals". (Reussner et al. [2016]) As part of the contract, for example, a component has to guarantee that it provides a specified set of services, if the services it is specified to require are provided by the environment it runs in. Additional contracts can also be given for more precise functional properties using pre- and postconditions, or for non-functional properties like response time and resource usage. A non-interference property, as discussed later in this chapter, can also be seen as a contract.

According to the PCM, a *Component* has a *name* and a set of *required roles* and *provided roles*. Each role again has a *name* and relates the component it belongs to with an *interface*. Note that components do not directly *have* interfaces, but are only related to interfaces which exist independent from whether or not a component actually requires or provides it. This allows that the same interface (as a model element) can be required and provided by different components or even by the same component multiple times. The PCM differentiates *Basic Components* and *Composite Components*, while the first is a basic building block for systems and the latter consists of an assembly of other components (i.e. a composite component represents a composition of components).

Example 7.1. Figure 7.1 shows a repository diagram illustrating the different entities of our shop example. The model contains the basic components *Cart*, *Sales*, and *Controlling*. Each of the components have provided and required roles, indicated by the arrows from the components to the interfaces.

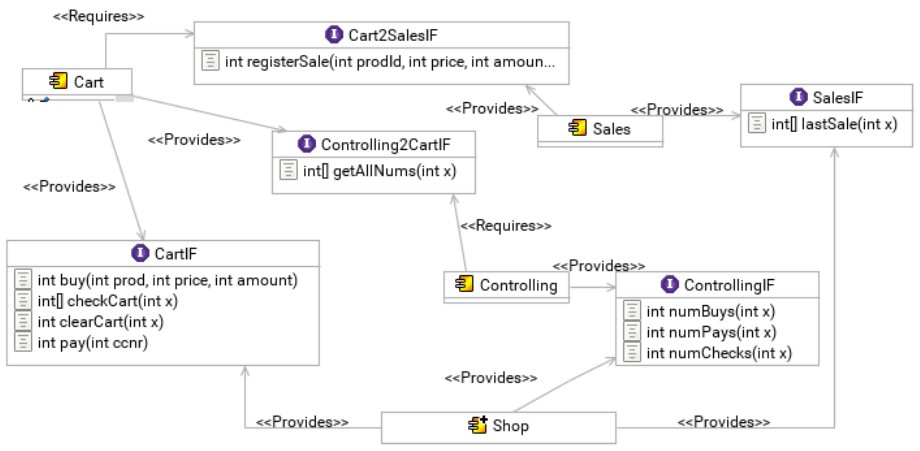

Figure 7.1: PCM Repository Diagram for the Shop example

For example, the component *Cart* provides the interfaces *Controlling2CartIF* and *CartIF*, and requires *Cart2SalesIF*. Each interface declares a set of service signatures which are contained in the respective interface element in the diagram. The composite component *Shop* is related via provided roles to the interfaces *CartIF*, *SalesIF*, and *ControllingIF*.

A PCM component can be seen as a *construction plan* for deployed software units, comparable to *classes* in object-oriented programming languages. In order to gain an instance of a component (comparable to an object in in object-oriented programming languages), the component has to be instantiated. In a PCM model *Assembly Contexts* serve as instantiations of components. An *Assembly Context* has a *name* and a reference to the *Component* it instantiates. An assembly context also provides and requires the interfaces which the instantiated component provides and requires via its roles. Each assembly context can only be embedded into exactly one composite component. In order to deploy a component into more than one composite component or more than one instances of a component into the same composite component, each of these instances are modeled as their own assembly context.

Assembly contexts are used to model the internal structure of a composite component. Additionally to the properties discussed above, a composite component has a set of assembly contexts. The set of assembly contexts describes the instances of components contained in the composite component. *Assembly Connectors* relate required roles to provided roles of assembly contexts. *Delegation Connectors* relate required roles of an assembly context to a required role of the containing composite component, and provided roles accordingly. Each required role of an assembly context has to be related to exactly one provided role of another assembly context or exactly one required role of

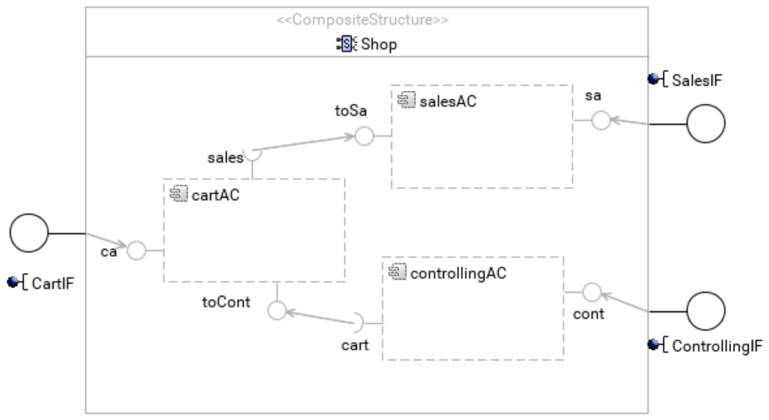

Figure 7.2: PCM assembly view for the Shop example

the composite component expressing which component's functionality is requested if the component related to the assembly context uses one of the services described by the interface. Related roles also have to match in the sense that the interfaces have to be *compatible*. We do not discuss how compatibility is defined in this context, but refer the interested reader to Reussner et al. [2016]. Instead, we require related interfaces to be the same interfaces.

Example 7.2. In Figure 7.2 the internal structure of the composite component *Shop* is shown. The components *Cart*, *Sales*, and *Controlling* are instantiated by the assembly contexts *cartAC*, *salesAC*, and *controllingAC* respectively. The internal required and provided roles are *wired* with assembly connectors, while the *Shop*'s provided roles are connected to provided roles of assembly contexts via delegation connectors (e.g. the role *CartIF* with role *ca*).

A PCM model formally defines the structure of a component-based system from the overall architecture, to its internal components, to single service signatures. In the following subsection, we additionally provide a formal model for the behavior of components and thus relate PCM components to DSCs as defined in Part I.

7.1.2 From Palladio Components to Distributed Service Components

In Chapter 2 we intuitively described components as DSCs in our framework as follows: A DSC consists of an underspecified initial internal state and a program. The program defines a sequential execution of a set of provided services which can read and write from and to the internal state, and call required services. Service calls are synchronous, i.e. after calling a service, the DSC's execution halts until the called service has terminated. All services provided by a DSC have to terminate if all intermediately called services terminate. Service calls and return values are communicated via messages, a combination of channels and values.

In the following we show how our formal notion of DSCs relates to PCM components. The PCM does not provide strict semantics or limitations for the behavior of components in general, but limits the definition to what a component guarantees based on provided and required roles. In order to consider PCM components as DSCs, we need to make additional restrictions to the behavior of a component.

Definition 7.1. A *BasicComponent bc* is a DSC, if

1. it is ensured that executions of all services of interfaces provided by *bc* are sequential,
2. all calls to required services are synchronous,
3. all services implemented by *bc* terminate,
4. all parameters of all services of all provided and required interfaces are passed by value, not by reference, and
5. the internal state of *bc* is neither write- nor readable by the environment.

Item 5 actually is more strict than necessary in order to map a Palladio component to DSCs from our framework. It may be allowed that several components are able to access shared constants. However, this can be seen as each component having its own constants available in its internal state as a copy.

A *Composite Component cs* is a composition (see Definition 2.4) if all encapsulated components of all child assembly contexts of *cs* are either *Basic Components* and DSCs or *Composite Components* and compositions.

We do not define whether these restrictions have to be ensured by the implementation of the component or by the environment the component runs in. This is a practical decision, since there are many frameworks used for realizing PCM models as programs, each dealing with these kinds of limitations to the programming model differently. For example, in widely used programming frameworks, like the Java Enterprise Edition (see Chapter 8 for details), the application container as the execution environment provides the necessary guarantees. In other cases, for example relational databases, these guarantees may be more implementation driven. Whether the guarantees mentioned above are realized by implementation or the environment may also depend on the granularity of the perspective. For example, it may differ whether a database system is considered abstractly or if the actual implementation of the database is considered.

In the remainder of this chapter, we assume all *Basic Components* and all *Composite Components* to have the properties described above. In the remainder of this subsection, we describe how a component specified in the PCM can be translated into our notion of DSCs as defined in Chapter 2.

Formalization of *Basic Component*

First we describe how *Basic Components* can be translated into our framework. According to the PCM, a *Basic Component bc* has a set of provided and required roles, which we refer to by *bc.provides* and *bc.requires* respectively. For a role *r* we refer to the set of services declared in the respective interface of the role by *r.services*. We define

$$\begin{aligned} prov_{bc} &:= \{r.s \mid r \in \mathit{bc.provides} \wedge s \in \mathit{r.services}\} \\ req_{bc} &:= \{r.s \mid r \in \mathit{bc.requires} \wedge s \in \mathit{r.services}\} \end{aligned}$$

For each service, we require an initial and a terminating channel according to Chapter 2. The channel name has to be unique for each service. While services are unique within an *Interface*, the same *Interface* can be provided and required by one component, or provided or required via several different roles. In order to distinguish these channels, we encode the role under which services are required and provided into the channel. Further, we have to distinguish call and termination channels. We therefore define channels as tuples consisting of the client component, the server component, the service name and the direction:

Definition 7.2.

$$\begin{aligned} Ini(r.s) &:= (r, this, s, CALL) && \text{if } r.s \in prov_{bc} \\ Fin(r.s) &:= (r, this, s, TERM) && \text{if } r.s \in prov_{bc} \\ Ini(r.s) &:= (this, r, s, CALL) && \text{if } r.s \in req_{bc} \\ Fin(r.s) &:= (this, r, s, TERM) && \text{if } r.s \in req_{bc} \end{aligned}$$

The channel name encodes a calling component, the called component of a service call, the called service, and whether the channel is the initial or terminating channel. If the component provides the respective service, the calling component, which at this point is not defined, is represented by the role over which the service is called and the called component is represented by the keyword *this*. If the component requires the service, the calling component, i.e. *bc*, is represented by *this* and the called component is represented by the respective role name.

In the example shown in Figure 7.1 and Figure 7.2 the channels defined for the component *Controlling* are

$$\begin{aligned}
&(\mathit{cont}, \mathit{this}, \mathit{numBuys}, \mathit{CALL}), && (\mathit{cont}, \mathit{this}, \mathit{numBuys}, \mathit{TERM}),\\
&(\mathit{cont}, \mathit{this}, \mathit{numPays}, \mathit{CALL}), && (\mathit{cont}, \mathit{this}, \mathit{numPays}, \mathit{TERM}),\\
&(\mathit{cont}, \mathit{this}, \mathit{numChecks}, \mathit{CALL}), && (\mathit{cont}, \mathit{this}, \mathit{numChecks}, \mathit{TERM}),\\
&(\mathit{this}, \mathit{cart}, \mathit{getAllNums}, \mathit{CALL}), && (\mathit{this}, \mathit{cart}, \mathit{getAllNums}, \mathit{TERM}).
\end{aligned}$$

In Chapter 2 we left the domain of values underspecified. Given a PCM model, we can make the domain concrete. The PCM defines primitive data types as *Primitive Type* for integers, rational numbers, boolean and similar. For each *Primitive Type* the domain $\mathbb{D}$ contains a set of respective values, i.e. integers for *INT*, rational numbers for *DOUBLE*, $\{\mathit{true}, \mathit{false}\}$ for *BOOL*. The PCM allows a model to define *Collection Data Types*, a list of values of a certain type. For each *Collection Data Type*, the domain contains all tuples $(\mathit{name}, \mathit{list}(l_1, ..., l_n))$, with name being the name of the collection data type, and $l_1, ..., l_n$ each being a value of the inner type of the collection data type. To model structured data types, i.e. data types which contain several values of different types, the PCM provides *Composite Data Types*. Each composite data type has a name and a set of inner variables. For each *Composite Data Type*, the domain contains tuples $(\mathit{name}, (\mathit{id}_1, v_1) \dots (\mathit{id}_n, v_n))$ with name being the name of the *Composite Data Type*, id_i the name of the i-th inner variable and v_i being a value of the inner variable.

Messages are, as in Chapter 2, tuples of channels and values, while we assume for a cooperative environment to only supply well-typed input messages.

Combining Definition 7.1 and Definition 7.2, we can represent the LTS defining the behavior of a component according to Chapter 2 as a configuration of the initial state and the body of the component.

Definition 7.3. For a *Basic Component bc* which provides the set of services $\mathit{prov}_{bc} = \{r_1.s_1, ...r_1.s_k, ..., r_m.s_1, ..., r_m.s_n\}$, we define the corresponding DSC as

$$\begin{aligned}
\mathit{handler}_{r.s} &:= \mathtt{read(}p \leftarrow \mathit{Ini}(r.s)\mathtt{)}\,;\mathit{body}_{r.s};\mathtt{write(}\mathit{res} \rightarrow \mathit{Fin}(r.s)\mathtt{)}\\
\mathit{body}_{bc} &:= \mathit{handler}_{r_1.s_1} \sqcap \dots \sqcap \mathit{handler}_{r_m.s_n}\\
\mathit{LTS}_{bc} &:= \langle \mathit{body}_{bc}; \sigma_{bc} \rangle
\end{aligned}$$

where $r.s \in \mathit{prov}_{bc}$, $\mathit{body}_{r.s}$ is the body of the service $r.s$ for an implementation of *bc*, and σ_{bc} is the initial state of an instantiation of *bc*.

We reuse the program constructs for reading and writing channels from Chapter 2. However, we do not require the implementation of the body of a service to actually use the simple while language defined there.

Formalization of *Composite Component*

In Chapter 2 composition of DSCs makes use of the fact that required and provided services use the same initial and terminating channels. Additionally, we make the assumption that at most two components communicate over the same channel, one by calling the respective service, the other one by providing the respective service. In a composite component, however, a provided interface of one assembly context may be related by assembly connectors to several assembly contexts requiring the interface. Or several delegation connectors may relate several provided roles of a composite component to one provided roles of an assembly context.

As a first step, we remove these $n : 1$ relations for provided roles by a program transformation. As stated in Section 2.4 if more than one DSCs use the same service provided by another DSC, we can just add a copy of that service to the body of the DSC with renamed initial and terminating channels. In order to be consistent with the channel naming above, additionally to duplicating the service, we also duplicate the provided role of the component and thus gain a role with a fresh name. The LTS for a basic component after adding the additional interfaces is then defined as in Definition 7.3. As a result, we can assume in the remainder that each provided interface of an assembly context within a composite component is at most related to one required interface via an assembly connector or one provided interface via a delegation connector.

In the PCM the allocation of required and provided roles of components communicating with each other is defined by *Assembly Connectors*. In order to formalize a composite structure as a composition of DSCs according to Definition 2.4 in our framework, we have to ensure that services provided and required by assembly contexts and related via assembly connectors within a composite component use the same initial and terminating channels. We achieve this by defining a renaming function ρ for channels which depends on the allocation contexts and allocation connectors of a composite component.

Definition 7.4. Let c be a *Composite Component*. We define the renaming function $\rho_c : \mathbb{C} \mapsto \mathbb{C}$, such that for all assembly contexts a and b embedded in c and all *Assembly Connectors* binding role p of the component embedded in a and role r of the component embedded in b, and all *Delegation Connectors* binding role i of a component embedded in an *Assembly Context* to the role e

of c such that

$$
\begin{aligned}
\rho_c(p, \mathit{this}, s, \mathit{CALL}) &:= (a.r, b.p, s, \mathit{CALL})\\
\rho_c(p, \mathit{this}, s, \mathit{TERM}) &:= (a.r, b.p, s, \mathit{TERM})\\
\rho_c(\mathit{this}, r, s, \mathit{CALL}) &:= (a.r, b.p, s, \mathit{CALL})\\
\rho_c(\mathit{this}, r, s, \mathit{TERM}) &:= (a.r, b.p, s, \mathit{TERM})\\
\rho_c(i, \mathit{this}, s, \mathit{CALL}) &:= (e, \mathit{this}, s, \mathit{CALL})\\
\rho_c(i, \mathit{this}, s, \mathit{TERM}) &:= (e, \mathit{this}, s, \mathit{TERM})\\
\rho_c(\mathit{this}, i, s, \mathit{CALL}) &:= (\mathit{this}, e, s, \mathit{CALL})\\
\rho_c(\mathit{this}, i, s, \mathit{TERM}) &:= (\mathit{this}, e, s, \mathit{TERM})\\
\rho_c(\alpha) &:= \alpha \ \mathit{otherwise.}
\end{aligned}
$$

The renaming function ρ_c implies a renaming function $\rho_c : \mathbb{T} \mapsto \mathbb{T}$ for traces defined as

$$
\begin{aligned}
\rho_c(\langle\rangle) &:= \langle\rangle\\
\rho_c(\alpha.v \frown t) &:= \rho_c(\alpha).v \frown \rho_c(t)
\end{aligned}
$$

The renaming function replaces the roles and *this* used as placeholders above by the respective instantiation of the components, i.e. the assembly contexts, and the roles. If a provided interface of an assembly context is delegated to a provided interface of the composite component, the role used as a placeholder for the calling component in the channel is replaced by the respective role of the composite component. Analogous, the renaming function replaces roles for required interfaces of an assembly context delegated to a required role of the composite component.

The renaming function for channels implies a canonical renaming function for traces, for which we overload the function operator. Renaming a trace with $\rho_c : \mathbb{T} \times \mathbb{T}$ amounts to renaming all channels in a trace according to $\rho_c : \mathbb{C} \times \mathbb{C}$. In the remainder of this chapter it should be clear from the context if a renaming is applied to a channel or a trace.

Example 7.3. In the example shown in Figure 7.1 and Figure 7.2, the renaming for the channels is according to Definition 7.4 as follows:

$$
\begin{aligned}
&\rho_{Shop}((cont, this, numBuys, CALL)) \\
&\qquad = (this.ControllingIF, controllingAC.cont, numBuys, CALL), \\
&\rho_{Shop}((cont, this, numBuys, TERM)) \\
&\qquad = (this.ControllingIF, controllingAC.cont, numBuys, TERM), \\
&\rho_{Shop}((cont, this, numPays, CALL)) \\
&\qquad = (this.ControllingIF, controllingAC.cont, numPays, CALL), \\
&\rho_{Shop}((cont, this, numPays, TERM)) \\
&\qquad = (this.ControllingIF, controllingAC.cont, numPays, TERM), \\
&\rho_{Shop}((cont, this, numChecks, CALL)) \\
&\qquad = (this.ControllingIF, controllingAC.cont, numChecks, CALL), \\
&\rho_{Shop}((cont, this, numChecks, TERM)) \\
&\qquad = (this.ControllingIF, controllingAC.cont, numChecks, TERM), \\
&\rho_{Shop}((this, cart, getAllNums, CALL)) \\
&\qquad = (controllingAC.cart, cartAC.toCont, getAllNums, CALL), \\
&\rho_{Shop}((this, cart, getAllNums, TERM)) \\
&\qquad = (controllingAC.cart, cartAC.toCont, getAllNums, TERM).
\end{aligned}
$$

Using the renaming function for traces, we can define a renaming of the LTS defining the behavior of a component after channel renaming.

Definition 7.5. Let c be a component instantiated by assembly context a in composite component d. Let further c_{LTS} be the LTS defining the behavior of c. We define the renaming function $\rho_{LTS,d} : LTS \mapsto LTS$ for a respective renaming function $\rho_d : \mathbb{T} \times \mathbb{T}$ such that $\forall t \cdot \rho_{LTS,d}(c_{LTS}) \xrightarrow{\rho_d(t)} \Leftrightarrow c_{LTS} \xrightarrow{t}$.

The renaming function for an LTS can be seen as modeling the instantiation of a component with an assembly context within a composite component. Note that for all messages that can be communicated by a component d and all messages the translated LTS can communicate, ρ_c is bijective. Also, for all traces that d_{LTS} can communicate and all traces $\rho_c d_{LTS}$ can communicate, the renaming function over traces ρ_c is bijective.

We can now define the behavior of a composite component as a composition of the assembly contexts within the composite component by synchronizing their LTS on their shared channels.

Definition 7.6. Let c be a composite component with assembly contexts $a_1, \ldots, a_n$ instantiating components $c_1, \ldots, c_n$. Let further $\rho_c : \mathbb{C} \times \mathbb{C}$ be the renaming function for c according to Definition 7.4 and $\rho_{LTS,c} : LTS \times LTS$

be the renaming function for c according to Definition 7.5. The LTS for c is defined as

$$\begin{aligned} c_{LTS} := (((\rho_{LTS,c}(c1_{LTS}) \; & \llbracket synch(c,(c1),(c2)) \rrbracket \; \rho_{LTS,c}(c2_{LTS})) \\ & \llbracket synch(c,(c1,c2),(c3)) \rrbracket \; \rho_{LTS,c}(c3_{LTS})) \\ & \ldots \\ & \llbracket synch(c,(c1,...,cn-1),(cn)) \rrbracket \; \rho_{LTS,c}(cn_{LTS})) \end{aligned}$$

with

$$\begin{aligned} & synch(c,(c_1,...,c_n),(d_1,...,d_m)) := \\ & \quad \{(c_i.r, d_j.p, s, CALL), (c_i.r, d_j.p, s, TERM) \mid \\ & \qquad \exists \textsf{\textit{AssemblyConnector}} \text{ binding role } r \text{ of } c_i \text{ to role } p \text{ of } d_j, \text{ and} \\ & \qquad\; i \in \{1 \ldots n\}, j \in \{1 \ldots m\} \text{ and} \\ & \qquad\; s \text{ is declared by the } \textsf{\textit{Interface}} \text{ which } r \text{ is a type of}\} \end{aligned}$$

The composition for a composite component is constructed by iteratively composing the translated LTS of each component, i.e. the LTS of the assembly contexts, and synchronizing them on their shared channels.

Example 7.4. In the running example, the composition for the assembly contexts *cartAC* and *controllingAC* is according to Definition 7.6 as follows:

$$\begin{aligned} & \rho_{LTS,Shop}(cartAC_{LTS}) \\ & \quad \llbracket (controllingAC.cart, cartAC.toCont, getAllNums, CALL), \\ & \qquad (controllingAC.cart, cartAC.toCont, getAllNums, TERM) \rrbracket \\ & \rho_{LTS,Shop}(controllingAC_{LTS}) \end{aligned}$$

7.2 Security Specification as an Extension of the PCM

Recently the PCM was extended to support explicit security specifications for systems. See Kramer et al. [2017] for details. The presentation this section is the result of the author of this thesis in close cooperation with other researchers and should not be considered a contribution of the author. In this section, we present the basics of the overall security extension of Palladio in order to provide the context for information-flow specifications in the next section. Security specifications are provided on three levels: explicit attacker specification, physical deployment and securing mechanisms, and information flow specifications for software components.

Attacker types are explicitly provided in the model as entities that should not use the system at all as well as entities which have a legitimate reason to use the system. For each attacker it is specified which physical locations he can access, while it is up to the domain expert to decide and argue why the attacker is not able to access other areas. Reasons for access restrictions may be due to not-modeled security mechanisms like camera surveillance within a building or locks on doors to certain areas. Also, for each attacker, the domain expert makes a judgment on types of tamper protections an attacker type is willing and able to overcome. For example, an entity not related to a company may be willing to destroy seals on hardware, while an employee of the company may not take this risk in an area under video surveillance. Additionally, for each attacker it can be specified that he should be allowed to have access to a certain type of information (see *datasets* in the following section), because he requires this information to fulfill his role.

From a physical point of view it can be specified for hardware resources at which physical locations they are deployed and which mechanisms are provided to secure the hardware against tampering. For hardware resources connecting parts of a system it can be specified which information transmitted over the links is secured by encryption.

The explicit attacker model and the specification of hardware in combination with software components deployed on the hardware, it can be deduced which attacker has physical access to information exchanged between software components. In order to decide whether this access poses a security risk, it is necessary to know if information this attacker should not know about is exchanged via accessible interfaces. Judging whether this is the case is twofold. For one, it has to be specified which categories of information are passed via accessible interfaces, and two, it has to be ensured that information passed via these interfaces indeed only contains information of the respective category. Access to information can be checked as a combination of physical access of an attacker to an interface and specification of the information passed via the interfaces. The condition that information passed via an interfaces indeed only contains information of the specified category is a non-interference property for a software component.

In the following section we introduce an information flow specification language for PCM models that relates input and output information with categories of information and states non-interference requirements for components.

7.3 Information Flow Specification

In this section we apply information flow specifications as annotations of services. An information flow specification serves two different purposes in the context of the security framework as introduced above, useful in different phases of system design. From a technical point of view, an information flow specification states a non-interference property as a requirement for

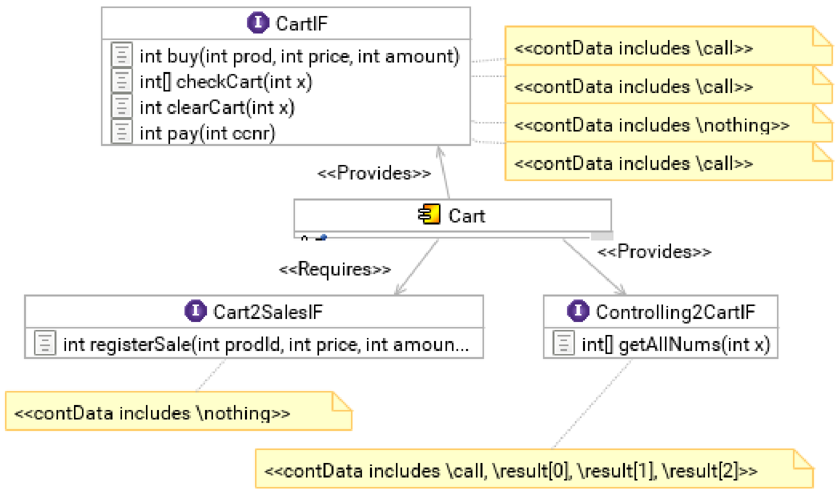
CartIF
int buy(int prod, int price, int amount)
int[] checkCart(int x)
int clearCart(int x)
int pay(int ccnr)
<<contData includes \call>>
<<contData includes \call>>
<<contData includes \nothing>>
<<contData includes \call>>
<<Provides>>
Cart
<<Requires>>
<<Provides>>
Cart2SalesIF
int registerSale(int prodId, int price, int amoun...
Controlling2CartIF
int[] getAllNums(int x)
<<contData includes \nothing>>
<<contData includes \call, \result[0], \result[1], \result[2]>>

Figure 7.3: Information flow specification for *Cart* component

the behavior of a component. The specification states which outputs of a component are influenced by which inputs. From a domain-oriented point of view, an information flow specification states which attacker or stakeholder is intended to know which input information and which outputs an attacker may potentially be able to actually observe.

We first introduce the syntax of non-interference specifications the extension of the PCM, and define its semantics using the formalization of components as LTS from earlier in this chapter. Following syntax and semantics, we show that compositionality properties for non-interference as shown in Part I also hold for non-interference specifications in PCM models.

7.3.1 Syntax and Semantics

Typically, a realistic component-based system manages information from different sources intended for different stakeholders used for different purposes. It is easy to see that a simple lattice separating information into *high* and *low* is not sufficient to describe intended or actual flows of information in a system or component. We therefore introduce *Datasets* as abstract descriptions for a collection of information (the term “information” is used informally here). Each dataset defines a lattice of high and low, stating that information which is *in* a dataset is low and information not in the dataset is high. If output information in a dataset is at most influenced by information in the same dataset, clearly this output information at most contains information in that dataset.

Example 7.5. Figure 7.3 shows an information flow specification for the *Cart* component. The example shows the dataset *contData* expressing the dataset representing information which an entity working in the controlling department may gain access to from a domain point of view.

A dataset is a model element and has as only property a name. In the following, we typically refer to a dataset by its name. In order to specify that some input or output of a component is specified to be in a dataset, we annotate the respective service signature with $\ll$*D includes L*$\gg$, where D is a dataset and L is a list of expressions, each expression either ranging over the parameters of the signature, the return value of the signature (here denoted by $\backslash result$), or the keyword $\backslash call$. The keyword $\backslash call$ states that the existence of a call to the respective service, as well as its termination, is low.

Example 7.6. The specification shown in Figure 7.3 states for the services *buy*, *checkCart*, and *pay* declared in the interface *CartIF* the existence of a call (and its termination) to the respective service is an element of *contData*, i.e. an entity which may know *contData*, also may know whether these services are called. The parameters, however, are not mentioned in the annotations meaning that the information transmitted as values of the parameters is high. For the service *clearCart*, the call itself is high with respect to *contData*, as is *registerSale* in the interface *Cart2SalesIF*. For service *getAllNums* in the interface *Controlling2CartIF* all return values are low, as is the call and its termination.

Note that the information flow specification is a property of an interface, and therefore all components providing or requiring the interface via a role have to satisfy the resulting non-interference property. Applying the specification to the interface, not the role, makes compositionality, as described later in this chapter, easier. On the other hand, this approach reduces re-usability of interfaces and components compared to applying the specification to a role, which would make the specification a property of the component instead of a the interface.

We do not fix here a concrete language for expressions used in a non-interference specification, except for the keywords $\backslash result$ and $\backslash call$. Depending on the purpose of the model and the domain of the modeled system, different languages may be a reasonable choice. The objects constraint language (OCL) (Warmer and Kleppe [1999]) may be a useful language for expressions in a general setting, while JML may be useful if the model is used as a basis for code generation of Java programs. The choice of language may also depend on other tools to which parts of the model are an input for further analysis. We assume, however, that there exists some kind of evaluation function *eval*, which allows evaluation of expressions given concrete values for parameters and return values. For a list of expressions L ranging over the parameter p, and p having value v we refer to the evaluation of L with

$eval_v(L)$. Lists of expressions are evaluated by evaluating each expression in the list separately, yielding a list of values.

We use JML as a specification language for expressions in examples in the remainder of this chapter.

Given a list L of expressions as introduced above, we can split the list in elements either concerning information in the call event of a service and elements concerning information in the termination event.

$$L_{call} := \langle\rangle \text{ iff } L = \langle\rangle \tag{7.1}$$
$$(\langle e_1\rangle \frown L)_{call} := L_{call} \text{ iff } e_1 \text{ ranges over } \backslash result \tag{7.2}$$
$$(\langle e_1\rangle \frown L)_{call} := \langle e_1\rangle \frown L_{call} \text{ otherwise} \tag{7.3}$$
$$L_{term} := \langle\rangle \text{ iff } L = \langle\rangle \tag{7.4}$$
$$(\langle e_1\rangle \frown L)_{term} := L_{term} \text{ iff } e_1 \text{ ranges over parameters} \tag{7.5}$$
$$(\langle e_1\rangle \frown L)_{term} := \langle e_1\rangle \frown L_{term} \text{ otherwise} \tag{7.6}$$

Example 7.7. In Figure 7.3, the specification for *getAllNums* defines the lists L, L_{call}, and L_{term} as

$$\begin{aligned} L \quad &= \backslash call, \backslash result[0], \backslash result[1], \backslash result[2] \\ L_{call} \quad &= \backslash call \\ L_{term} \quad &= \backslash call, \backslash result[0], \backslash result[1], \backslash result[2] \end{aligned}$$

The semantics of the $\ll D$ *includes* $L\gg$ annotation is defined from a component's point of view. Each dataset defines in combination with the $\ll D$ *includes* $L\gg$ specifications an equivalence relation over messages, i.e. tuples of channels and values.

Definition 7.7 (Dataset Equivalence Relation)**.** Given a component c and a dataset D. The equivalence relation for messages w.r.t. the dataset D for component c is defined as

$$
\begin{align}
&(p, r, s, dir).v \sim_{c,D} \square \Leftrightarrow \tag{7.7}\\
&\quad (p \neq \mathit{this} \wedge r \neq \mathit{this}) \vee \tag{7.8}\\
&\quad (r = \mathit{this} \wedge c \text{ does not provides } s \text{ via role } p) \vee \tag{7.9}\\
&\quad (p = \mathit{this} \wedge c \text{ does not require } s \text{ via role } r) \vee \tag{7.10}\\
&\quad (s \text{ has annotation with } D \vee \tag{7.11}\\
&\quad (s \text{ has annotation with } D, L \wedge \tag{7.12}\\
&\qquad dir = \mathit{CALL} \wedge \backslash \mathit{call} \notin \mathit{eval}_v(L_{\mathit{call}})) \vee \tag{7.13}\\
&\quad (s \text{ has annotation with } D, L \wedge \tag{7.14}\\
&\qquad dir = \mathit{TERM} \wedge \backslash \mathit{call} \notin \mathit{eval}_v(L_{\mathit{term}})) \tag{7.15}\\
&(p, r, s, dir).v \sim_{c,D} (p', r', s', dir').v' \Leftrightarrow \tag{7.16}\\
&\quad ((p, r, s, dir).v \sim_{c,D} \square \wedge (p', r', s', dir').v' \sim_{c,D} \square) \vee \tag{7.17}\\
&\quad (p = p' \wedge r = r' \wedge s = s' \wedge dir = dir' \wedge \tag{7.18}\\
&\qquad ((r = \mathit{this} \wedge c \text{ provides } s \text{ via role } p \wedge \tag{7.19}\\
&\qquad\qquad s \text{ has annotation with } D, L_i) \vee \tag{7.20}\\
&\qquad\quad (p = \mathit{this} \wedge c \text{ requires } s \text{ via role } r \wedge \tag{7.21}\\
&\qquad\qquad s \text{ has annotation with } D, L)) \wedge \tag{7.22}\\
&\qquad ((dir = \mathit{CALL} \wedge \mathit{eval}_v((L)_{\mathit{call}}) = \mathit{eval}_{v'}((L)_{\mathit{call}})) \vee \tag{7.23}\\
&\qquad\quad (dir = \mathit{TERM} \wedge \mathit{eval}_v((L)_{\mathit{term}}) = \mathit{eval}_{v'}((L)_{\mathit{term}})))) \tag{7.24}
\end{align}
$$

First, we define the invisible events (Line 7.7). All messages neither sent nor received by the component are invisible (Line 7.8), as are all messages communicated on services neither provided (Line 7.9) nor required (Line 7.10) by the component. Messages are also invisible w.r.t. the dataset D, if there are annotations for the service adding any information to the dataset (Line 7.11). For those services, which do have an annotation with D, the message is invisible, if it represents a service call and the evaluated list L, limited to calling expressions does not contain the marker $\backslash \mathit{call}$ (Line 7.12). Similar, a termination message is invisible, if the list L, limited to termination expressions, does not contain the marker $\backslash \mathit{call}$ (Line 7.14).

Note that this specification language only allows to specify calls to be either high or low. It does not allow to specify a service call to be high or low depending on values of parameters, as allows our framework in Part I. This design decision was made to avoid very long textual specifications in a model, even if this means limiting the expressiveness of the specification language. If in the future specifying services invisible depending on parameters is deemed useful, the specification language can be extended in a straight forward way.

Two events are equivalent according to the specification (Line 7.16), if both events are invisible (Line 7.17). Further, two events are equivalent if they refer to the same caller and callee, both messages are sent because of the same service, and both messages have the same the same communication direction (i.e. call or termination) (Line 7.18). Further, the service referenced in the channel is provided by the component and the component is the receiver of the call (Line 7.19), or the service is required by the component and the service is called by the component (Line 7.21). Finally, the specification lists L_{call} and L_{term} respectively evaluate with the communicated values to the same lists (Lines 7.23 and 7.24).

Example 7.8. In Figure 7.3 the equivalence relation for the component *Cart* for dataset *contData* is defined (in part) as follows. We limit the presentation of the equivalence relation to some examples of channels due to its verbosity.

$$
\begin{aligned}
&(\mathit{ca}, \mathit{this}, \mathit{buy}, \mathit{CALL}).v \sim \square && \Leftrightarrow \mathit{false}\\
&(\mathit{ca}, \mathit{this}, \mathit{buy}, \mathit{TERM}).v \sim \square && \Leftrightarrow \mathit{false}\\
&(\mathit{ca}, \mathit{this}, \mathit{clearCart}, \mathit{CALL}).v \sim \square && \Leftrightarrow \mathit{true}\\
&(\mathit{ca}, \mathit{this}, \mathit{clearCart}, \mathit{TERM}).v \sim \square && \Leftrightarrow \mathit{true}\\
&(\mathit{this}, \mathit{sales}, \mathit{registerSale}, \mathit{CALL}).v \sim \square && \Leftrightarrow \mathit{true}\\
&(\mathit{this}, \mathit{sales}, \mathit{registerSale}, \mathit{TERM}).v \sim \square && \Leftrightarrow \mathit{true}\\
&(\mathit{toCont}, \mathit{this}, \mathit{getAllNums}, \mathit{CALL}).v \sim \square && \Leftrightarrow \mathit{false}\\
&(\mathit{toCont}, \mathit{this}, \mathit{getAllNums}, \mathit{TERM}).v \sim \square && \Leftrightarrow \mathit{false}\\
&(\mathit{ca}, \mathit{this}, \mathit{buy}, \mathit{CALL}).v \sim (\mathit{ca}, \mathit{this}, \mathit{buy}, \mathit{CALL}).v' && \Leftrightarrow \mathit{true}\\
&(\mathit{this}, \mathit{sales}, \mathit{registerSale}, \mathit{CALL}).v \sim (\mathit{this}, \mathit{sales}, \mathit{registerSale}, \mathit{CALL}).v'\\
&\qquad\qquad \Leftrightarrow \mathit{true}\\
&(\mathit{toCont}, \mathit{this}, \mathit{getAllNums}, \mathit{TERM}).v \sim (\mathit{toCont}, \mathit{this}, \mathit{getAllNums}, \mathit{TERM}).v'\\
&\qquad\qquad \Leftrightarrow \mathit{eval}(v[0]) = \mathit{eval}(v'[0]) \wedge \mathit{eval}(v[1]) = \mathit{eval}(v'[1])\\
&\qquad\qquad\quad \wedge \mathit{eval}(v[2]) = \mathit{eval}(v'[2])
\end{aligned}
$$

A component c satisfies its non-interference specification for a dataset D specified in the model with annotations of the form $\ll D$ *includes* $L\gg$, if the LTS of the component is non-interferent w.r.t. $\sim_{c,D}$, i.e. $c_{LTS} \in \mathtt{Coop}_c$-NI w.r.t. $\sim_{c,D}$ according to Definition 3.9.

7.3.2 Soundness of Composition of Palladio Components

We have introduced the syntax for non-interference specifications for PCM components and provided semantics for the specification using our framework discussed in Part I. However, compositionality results are not directly applicable. First of all, if a PCM component is non-interferent w.r.t. its specification, this is not trivially true for the composition of components, since channels and traces are renamed for assembly contexts, i.e. instantiations of components. Second, the non-interference specification for a composite component is different than the specification of each component and their composition. It is not directly clear that if each instantiated component embedded in a composite component is non-interferent w.r.t. its own specification, it is also non-interferent w.r.t. the composite component's specification.

We show in the following that if a component is non-interferent w.r.t. its specification, there exists a non-interference specification such that the instantiation of the component is non-interferent w.r.t. this specification. After this, we discuss non-interference in composed assembly contexts. We show that if two assembly contexts are non-interferent w.r.t. their own specification, there exists a non-interference specification such that both assembly contexts are non-interferent w.r.t. this specification. Finally, we discuss non-interference for composite components. We show that if assembly context contained in the composite component is non-interferent w.r.t. its own specification, then the composite component is non-interferent w.r.t. its specification.

First we show that non-interference is robust w.r.t. renaming of the channels as described in Definition 7.4. In order to do this, we first define a renaming function for equivalence relations w.r.t. a renaming function for channels.

Definition 7.8. Given a *Composite Component* c, its renaming function ρ_c according to Definition 7.4 and an equivalence relation $\sim$. We define a renaming function $\rho_c : (\mathbb{M} \times \mathbb{M}) \mapsto (\mathbb{M} \times \mathbb{M})$ such that $\alpha.v \sim \beta.w \Leftrightarrow \rho_c(\alpha).v \rho_c(\sim) \rho_c(\beta).w$

The renamings of two messages are equivalent w.r.t. the translated equivalence relation iff the original messages are equivalent to the original equivalence relation. In the remainder, we write $\sim_c$ instead of $\rho_c(\sim)$.

Theorem 7.1. *Given a component d embedded in an* AssemblyContext *in a, embedded in a composite component c, and an equivalence relation $\sim$ such that $d_{LTS} \in$* $\texttt{Coop}_d$*-NI w.r.t. $\sim$. It holds $\rho_c(d_{LTS}) \in$* $\texttt{Coop}_{\rho_c(d_{LTS})}$*-NI w.r.t. $\sim_d$*

Proof. We proof the contra-positive. Assume $\rho_c(d_{LTS})$ is not non-interferent w.r.t. $\sim_d$. That means, there exists an attack (Definition 3.7) (ω_1, ω_2, t) on $\rho_c(d_{LTS})$, such that $\omega_1 \models \rho_c(d_{LTS}) \xrightarrow{t}$ and for all t_2 with $\omega_2 \models \rho_c(d_{LTS}) \xrightarrow{t_2}$ it holds $t \nsim_d t_2$.

We now construct an attack on d_{LTS} w.r.t. $\sim$.
Let $\omega_1'(s) = \{\alpha_1.v_1, \ldots, \alpha_n.v_n\} :\Leftrightarrow \omega_1(\rho_c(s)) = \{\rho_c(\alpha_1).v_1, \ldots \rho_c(\alpha_n).v_n\}$ and ω_2' analogous. Due to the definition of $\nsim_d$ in Definition 7.8 it directly follows $\omega_1' \sim \omega_2'$. Since the renaming function over channels is bijective, there exists a trace s such $\rho_c(s) = t$. By definition of the renaming function for LTS in Definition 7.5, we get $\omega_1' \models d_{LTS} \xrightarrow{s}$. Since $d_{LTS} \in \texttt{Coop}_d$-NI w.r.t.

$\sim$, there exists a trace s_2 such that $\omega_2' \models d_{LTS} \xrightarrow{s_2}$. Again, by Definition 7.5 and additionally by definition of ω_2', we get $\omega_2 \models \rho_c(d_{LTS}) \xrightarrow{\rho_c(s_2)}$. And finally, by Definition 7.8 and $s \sim s_2$, we know that $\rho_c s \sim_c \rho_c(s_2)$, and therefore $\rho_c(s_2)$ is the witness contradiction the assumption that (ω_1, ω_2, t) is an attack. ◁

We now show that if two components are non-interferent w.r.t. their own specification, the instantiation of the two components within a composite component are also non-interferent w.r.t. the intersection of the translated equivalence relations.

Theorem 7.2. *Let a be a Composite Component with renaming function ρ_a, let D be a dataset, and c and d components embedded in a. Let further be $\sim_{c,D}$ and $\sim_{d,D}$ be the equivalence relations according to the specification of c and d gained by applying Definition 7.7 and let $c_{LTS} \in \mathtt{Coop}_{c_{LTS}}$ w.r.t. $\sim_{c,D}$ and $d_{LTS} \in \mathtt{Coop}_{d_{LTS}}$ w.r.t. $\sim_{d,D}$. Let $\sim := \rho_a(\sim_{c,D}) \cap \rho_a(\sim_{d,D})$. Then it holds $\rho_a(c_{LTS}) \in \mathtt{Coop}_{\rho_a(c_{LTS})}$ w.r.t. $\sim$ and $\rho_a(d_{LTS}) \in \mathtt{Coop}_{\rho_a(d_{LTS})}$ w.r.t. $\sim$.*

Proof. We know according to Theorem 7.1 that $\rho_a(c_{LTS}) \in \mathtt{Coop}_{\rho_a(c_{LTS})}$ w.r.t. $\rho_a(\sim_{c,D})$ and $\rho_a(d_{LTS}) \in \mathtt{Coop}_{\rho_a(d_{LTS})}$ w.r.t. $\rho_a(\sim_{d,D})$.

We assume towards contradiction that $\rho_a(c_{LTS}) \notin \mathtt{Coop}_{\rho_a(c_{LTS})}$ w.r.t. $\rho_a(\sim)$, i.e. there exists an attack: $\omega_1 \sim \omega_2$ and $\omega_1 \models \rho_a(c_{LTS}) \xrightarrow{t_1}$ and for all t_2 with $\omega_2 \models \rho_a(c_{LTS}) \xrightarrow{t_2}$ it holds $t_1 \nsim t_2$. We show that there does exist such a trace t_2.

By definition of $\sim$, we know that for all messages $m_1 \sim m_2$ it also holds $m_1 \rho_a(\sim_{c,D}) m_2$ and therefore also for all traces it also holds $t_1 \sim t_2 \implies t_1 \rho_a(\sim_{c,D}) t_2$, and thus $\omega_1 \rho_a(\sim_{c,D}) \omega_2$.

Since $c_{LTS} \in \mathtt{Coop}_{c_{LTS}}$ w.r.t. $\sim_{c,D}$ and Theorem 7.1 (i.e. $\rho_a(c_{LTS}) \in \mathtt{Coop}_{\rho_a(c_{LTS})}$ w.r.t. $\rho_a(\sim_{c,D})$), we know there exists a trace t_2 such that $\omega_2 \models \rho_a(c_{LTS}) \xrightarrow{t_2}$ and $t_1 \rho_a(\sim_{c,D}) t_2$. For all messages in t_1 and t_2, either they are communications between c and d, or they do not involve d. By construction of $\sim_{c,D}$ (Definition 7.7) and the renaming function ρ_a, we know that all messages which represent communication not involving d: $m \rho_a(\sim_{d,D}) \Box$. All messages which involve c as well as d in their communication are due to calls or terminations of services which are either provided by c and required by d or vice versa. Since c and d can only be bound via the same interfaces, the services have the same specification in the PCM model. Therefore, for messages m_1, m_2 involving c and d it holds $m_1 \rho_a(\sim_{c,D}) m_2 \Leftrightarrow m_1 \rho_a(\sim_{d,D}) m_2$. Therefore it has to hold that $t_1 \rho_a(\sim_{d,D}) t_2$ and therefore also $t_1 \sim t_2$. This contradicts the assumption that no such trace exists and therefore $\rho_a(c_{LTS}) \in \mathtt{Coop}_{\rho_a(c_{LTS})}$ w.r.t. $\sim$.

The proof for $\rho_a(d_{LTS}) \in \mathtt{Coop}_{\rho_a(d_{LTS})}$ w.r.t. $\sim$ is analogue. ◁

Theorem 7.2 shows that we can intersect the equivalence relations we gain from the PCM model for components and gain a common equivalence relation for their assembly contexts by intersecting the equivalence relations of each assembly context. Since both components are equivalent w.r.t. the intersection, we can apply the Composition Non-Interference Theorem (Theorem 3.2) and know the composite component is non-interferent.

It is left to show that the composition of all components embedded in a *Composite Component* are also non-interferent w.r.t. the specification of the *Composite Component*. Trivially, all messages which are communicated by the *Composite Component* via the provided and required interfaces have the same specification as the messages of the internal components, since they are declared by the same interfaces with the same specifications in the model. The composition however communicates additionally the internal messages as outputs of the composition (messages on synchronized channels are considered outputs of the composition). By construction of the equivalence relation for the *Composite Component* (Definition 7.7), these outputs are specified to be invisible. According to Definition 4.4 the equivalence relation gained from the specification of the composite component is a weakening of the combined equivalence relations gained from the specifications of the assembly contexts. Therefore the composition is non-interferent w.r.t. the specification of the *Composite Component* according to the Weakening Theorem (Theorem 4.3).

Theorem 7.3. *Let c be a Composite Component with assembly contexts $a_1, ..., a_n$ instantiating components $c_1, ..., c_n$. Let further D be a dataset and each component c_i be non-interferent w.r.t. its specification referring to D. Then c is non-interferent w.r.t. its specification referring to D.*

Making information flow specifications a property of service declarations in interfaces instead of properties of roles was a design decision made above. As a result, components, assembly contexts, and composite components share a common non-interference specification, if they provide or require the same interface via a role. This design decision directly provides us with a non-interference guarantee for composed assembly contexts and composite components, if each basic component in a system is non-interferent w.r.t. its own specification.

However, this design decision makes components less re-usable, since, if an interface is used in a different context, the required non-interference specification can differ in the new context. This problem can be overcome by making non-interference specifications a property of roles rather than interfaces. Then, for composed assembly contexts it has to be shown that required and provided interfaces related via assembly connectors or delegation connectors are compatible w.r.t. their specification in order to make the composition well-defined. Since we do allow in general declassification in our specification language, this compatibility check is an undecidable problem, which we consider impractical for a graphical specification language like the PCM on an architecture level.

7.4 Related Work

A recent systematic overview of specification approaches for model-driven system design for security properties can can be found in Nguyen et al. [2015]. One finding of the review is that many approaches concentrate on access control properties, which is orthogonal to the work presented in this chapter. We limit the discussion here to approaches which support information flow specifications.

SECTET by Alam et al. [2004] is a specification language for security requirements in workflow management systems. The main focus of this approach is on the specification of access control mechanisms. Hafner and Breu [2009] extend the specification language with *basic security policies*, which allow the specification of confidentiality, integrity and non-repudiation for documents. The specification is meant to be implemented using public key encryption mechanisms, and thus specifies less of an information flow property than a requirement for encrypting documents.

UMLSec by Jürjens [2005] extends UML with a multitude of security related specification mechanisms ranging from fair exchange, over access control, to information flow security. Information flow specifications are given for specifications of dynamic behavior in the form of state machines using UML stereotypes. The stereotype explicitly states the low information (or high information for integrity properties). Semantics of information flow properties in UMLSec are given as attacker knowledge, which makes it hard to compare the non-interference notion with the notion underlying the work presented in this chapter. Information flow properties in UMLSec are local to activities, do not provide a specification of non-interference of components and are to the best of our knowledge not compositional.

IFlow by Stenzel et al. [2014] extends UML by a UML profile for the specification of confidentiality of data. Messages in sequence diagrams can be annotated with security domains, and UML activity diagrams are used to specify which security domains may not influence each other. IFlow supports temporal declassification of information in the sense of a variable being declassified after a particular action. Analysis on whether a behavioral model of the system satisfies its non-interference specification is done by creating code from sequence diagrams and applying information flow analysis tools on the generated code (Katkalov et al. [2013]). The main differences to our approach are that for one, IFlow does not support what-declassification, including different security properties for different parameters of one message, and, second, due to the underlying non-interference notion does not support secrecy of message existence. Further, by using sequence diagrams for the specification of the classification of information, a behavioral specification of a system is required, while for our approach a static representation of the system is sufficient.

secBIP by Ben Said et al. [2014] is a model-driven approach for explicit information flow specification and analysis of components. The approach distinguishes between *event-flow* and *data-flow* security, where data-flow security is similar to our non-interference notion for services in Part I and event-flow security is similar to non-interference for interactive systems. A specification maps data variables, ports and interactions to security levels and a policy defines allowed flows between security levels. One difference between their work and ours is that they do not allow detailed specification of secrecy of the content of messages, but only their existence. Also, they consider event-flow security (similar to message equivalence in our framework) and data-flow security (similar to state equivalence in our framework) as separate, unrelated security properties. Our extension of Palladio is only concerned with what they call event-flow security. We consider their data-security a supporting property for compositionality of services, and irrelevant for black-box specifications for components.

7.5 Conclusion

In this chapter we instantiated our framework as a graphical specification language for non-interference properties of components specified in the Palladio Component Model. Non-interference specifications are provided as annotations of services in a PCM model. Further, non-interference specifications are parameterized with datasets, i.e. abstract descriptions of information managed by the modeled system. We provided semantics for non-interference specifications based on LTS as used in our framework and we showed that non-interferent components can be combined to non-interferent composite components.

Parametrized specifications allow input and output information to be precisely specified as high or low for different purposes, e.g. *billing data* and *delivery data*, or based on a domain-oriented security lattice, like *public*, *internal*, and *top secret*. Especially the notion of datasets allows to specify the sensitivity of information close to the domain in which a system is used. Thus our specifications can serve as a communication tool between stakeholders and the system engineer during requirement elicitation. Compositionality of non-interference for PCM components allows to limit information flow analysis to basic components during quality assurance. Also, the specifications of basic components directly provide a requirement for the programmer of a component during the implementation phase.

Our specification language was applied in two case studies. Greiner and Herda [2017a] applied our specification language to specify information flow requirements for CoCoME (Rausch et al. [2008]), a case study describing a cashier system of a retail store. Different entities involved directly or indirectly in a purchase process or store management are considered as attackers. For each entity one dataset is defined and the overall information flow policy for the system is specified from a domain point-of-view, i.e. for each entity it is specified, which input information the entity is meant to gain knowledge about, and which outputs the entity is directly able to observe. The system specification then is refined by additional specifications for interfaces binding internal components.

In work by Kramer et al. [2017] a multi-user cloud storage system is modeled, including security mechanism for physical accessibility of hardware and encryption requirements for communication links. Multiple entities interacting with the cloud storage system are modeled as potential attackers, including users of the cloud, administration staff, unregistered guest users, and others. For each potential attacker a dataset is declared and an extension of the non-interference specifications as presented in this chapter are used to label input and output information as an element of these datasets. An analysis shows that the modeled system is secure w.r.t. the attackers, taking information flow properties into account, combined with physical access specifications and tampering powers of each attacker.

8 Deductive Verification of Dependency Clusters in JavaEE

In this chapter, we preset a program analysis technique for Dependency Cluster (as introduced in Chapter 4) for components implemented in the Java Enterprise Edition. Our approach is based on the deductive verification tool for Java programs KeY.

Many different techniques can be found in the literature for program analysis for non-interference properties. A discussion of some of the techniques can be found in Section 8.6. We are confident that several of these techniques can be adapted to support our framework, however, each with limitations unique to the approach. Automatic approaches typically lack the precision necessary to analyze programs w.r.t. elaborate declassification. Interactive and autoactive techniques, on the other hand, require manual interaction, often making verification a time-consuming task.

One novel and important feature of our framework is that declassification can be described very precisely on a semantic level. Therefore, we want to present an analysis technique which supports this precision as far as possible. We extend in this chapter the KeY-approach, a technique for reasoning about Java programs based on the dynamic logic JavaDL. The analysis itself is implemented in the KeY tool, an interactive theorem prover which can read annotated Java source code, and translate specifications into JavaDL formulas. By applying rules of a sequent calculus, KeY can auto-actively, i.e. potentially with user interaction, verify the validity of the formulas.

In order to apply our framework to the KeY approach, we have to make extensions on several levels: we extend the logic itself, introduce new rules to the calculus, extend the specification language and create new proof obligations for Dependency Cluster verification.

We first introduce JavaEE, a framework for implementing distributed systems in the Java programming language. JavaEE is widely used in practice, often in web service architectures for implementing the business logic of an application. We then introduce in Section 8.2 JavaDL, a dynamic logic for reasoning about Java programs, and JML, a specification language for Java programs. Both, JavaDL and JML, currently support sequential Java programs, but not constructs necessary for remote method invocations, the JavaEE equivalent of service calls. In Section 8.3 we extend JavaDL by constructs required for reasoning about JavaEE programs and introduce *service contracts*, a variation of method contracts which abstractly describe the effects of remote method calls. In the section thereafter, we define an extension of JML for Dependency Cluster specification for JavaEE services, and apply our non-interference framework from Part I to gain proof obligations for distributed programs implemented in JavaEE. As a result, this allows us to verify for JavaEE components that they are non-interferent w.r.t. an object-oriented Dependency Cluster specification. In Section 8.5 we apply our extension of KeY by verifying non-interference for a web shop system. Finally, we present related work and conclude the chapter.

The results presented in this chapter are an extension of work by the author previously published in Greiner et al. [2017b] and Greiner et al. [2017a]. Parts of the results presented here are based on two Bachelor Theses, which were supervised by the author of this thesis (see Diekhoff [2017] and Krämer [2017]).

8.1 JavaEE

The Java Enterprise Edition (JavaEE) is a framework that extends the Java programming language to modular, distributed, and highly scaleable applications, aimed for business applications. JavaEE is specified in the Java Specification Request (JSR) 342 (JavaEE [2013]). Our presentation in this section is based on this document, while we concentrate on the parts of JavaEE most relevant for our purposes. Hence, the presentation in this section is not a complete introduction for JavaEE, but should only be seen as an introduction of some concepts which we require in the remainder of this chapter.

JavaEE separates low-level system requirements like transaction handling, authentication, communication, and concurrency from the actual business logic. The business logic is implemented as so-called applications, consisting of a set of Enterprise Java Beans (EJBs), to which we refer to in the following as *beans*. Beans are executed by containers, which are responsible to ensure the low-level guarantees on which the application relies on.

8.1.1 Enterprise Java Beans

Beans are components of a JavaEE application and the services provided by beans are defined in *remote interfaces*, i.e. Java interfaces with the annotation `@Remote`.

Example 8.1.
```
import javax.ejb.Remote;

@Remote
public interface CartIF {
  public int buy(int prod, int price, int amount);
  public int pay(int ccnr);
}
```

The interface `CartIF` defines the methods `buy` and `pay`. Since `CartIF` is annotated with `@Remote`, the methods are declared to be remote methods.

A bean is implemented as a Java class which implements a remote interface, and is additionally annotated, with `@Stateless`, `@Stateful`, or `@Singleton`, declaring the bean a *stateless bean*, a *stateful bean*, or a *singleton bean*. *Stateless beans* must not manage an internal state in the sense that whenever a remote method provided by a stateless bean is called, the behavior of the method execution is independent from an internal state. *Stateful beans* may manage an internal state and can thus be used to keep information over several remote method calls. *Singleton beans* also may manage an internal state and are instantiated exactly once per application. Singleton beans can be used to exchange information between different beans, since all beans access the same singleton bean.

Message-driven beans are a third kind of bean which do not provide return values with their remote methods. Remote methods provided by message-driven beans can be called by another bean asynchronously, i.e. the calling bean resumes its execution, while the message-driven bean executes the called remote method.

We do not consider message driven beans in this chapter, however we would like to note that we assume that our framework can be easily extended to support this kind of asynchronous execution. Since we do not provide formal proofs for this asynchronous behavior in our framework, we exclude message driven beans from the further discussion.

A bean can hold a reference to another bean by implementing a field of the respective remote interface type annotated with `@EJB`. At creation time of the bean the container in which the bean is run, is responsible to instantiate the field with a non-null object implementing the respective remote interface, the proxy object. In order to call remote methods the bean calls the respective method of the proxy object.

Example 8.2. The following listing defines the bean `Cart`. The annotation `@Stateful` declares the bean to be a stateful session bean and the annotation `@EJB` declares the field `sale` to contain a reference to a bean implementing the interface `Cart2SalesIF`. The bean manages several fields representing the product in the cart, its price, and the amount of the product in the cart. Further, it manages the fields `countbuy`, `countpay`, and `countcheck` to record how the bean is used by a customer. The bean implements the functionality of the cart component of the running example we used throughout Part I of this thesis.

```
@Stateful
public class Cart implements CartIF{
  int product;
  int prodprice;
  int prodamount;
  int countbuy;
  int countpay;
  int countcheck;

  @EJB
  Cart2SalesIF sale;

  public int buy(int prod, int price, int amount) {... }
  public Triple checkCart(int x) {...}
  public int clearCart(int x) {...}
  public int pay(int ccnr) {...}
  public Triple getAllNums(int x) {...}
}
```

JavaEE defines some restrictions to implementations of applications. Applications must not directly access persistent storage or manage sockets, but have to use the API provided by the container. Applications must not create or manage threads on their own, only the container may manage concurrency. Also, applications must not access static fields, unless they are declared final.

8.1.2 Container

Container are responsible for executing beans, managing their life cycle, concurrent remote method calls, and forwarding remote method calls to the respective objects actually implementing the called bean.

The life cycle of a stateless bean is undefined. The container may decide if only one instance of the class is created at application start-up and all remote method calls are redirected to this one bean, if for each call a new bean is created, or anything in between. Typically, the container decides on this strategy based on what is best for performance and memory usage. Stateful beans belong to one client, e.g. one other bean holding a reference to an associated proxy object, and all remote method calls to this proxy object are forwarded by the container to this one bean. Therefore, the life cycle of a stateful bean is defined by the life cycle of the client. Singleton beans are created at start-up of the application and terminate with the application.

When a client makes a remote method call, the container uses the Java API to serialize the parameters of the call and, after termination, the return value. If the application is distributed, the serialized data is transmitted to another container, where the respective called bean lives. Then the data is deserialized and the respective method of the bean is called with the deserialized parameters, or, in case of the return value, the deserialized return value is provided the calling bean. Since the container ensures that parameters are always passed by value and by disallowing beans to have write-access to static fields, each bean manages its own state and two beans can never have access to one reference.

By default, the container ensures that concurrent calls of remote methods implemented by the same bean are sequential, i.e. from the bean's point of view, a remote method is only called after any previous call to a remote method has terminated. Since beans are disallowed to create threads, within one bean all remote method executions are single-threaded and no concurrency can be caused.

We would like to note that JavaEE allows two ways for the programmer to override the default concurrency management and allow concurrency within one singleton bean by using special annotations. For one, the programmer can declare by annotation read-only remote methods, which must not change the state of the bean. Then read-only remote methods may be executed concurrently, while remote methods marked to write the state may only be executed without other threads executing the bean. And second, programmers may also define their own concurrency management, however this has to be done very carefully. We assume in the following that there are no concurrent remote method executions allowed in a bean, i.e. the default configuration is chosen.

The container provides several guarantees: Remote methods are executed sequentially, not concurrently. Parameters and return values are serialized and deserialized, and beans must not manage threads. This combination makes beans very similar to DSCs as defined in our framework.

8.2 JavaDL and JML

In this section, we introduce syntax and semantics of *Java Dynamic Logic* (JavaDL), and explain some elements of the logic which are not common in first order predicate logic. Further, we introduce the Java Modeling Language (JML) as a specification language for Java programs. We limit the discussion of JavaDL and JML to the extent needed in this chapter; for a full account please refer to Weiß [2011] and Ahrendt et al. [2016]. The presentation in this section is based on the foundations chapter by Scheben [2014].

8.2.1 JavaDL Syntax and Semantics

JavaDL is a first order dynamic logic (Harel et al. [2000]) tailored for the Java programming language. Additionally to first order predicate logic elements, JavaDL defines the diamond operator $\langle p \rangle$ where p is a sequential Java program. The logic also contains updates $\{u\}$, which can be seen as substitutions, directly built into the logic.

The syntax of JavaDL is based on signatures. A signature is a tuple $\Sigma = (\mathcal{F}, \mathcal{F}^{\text{Unique}}\mathcal{P}, \mathcal{V}, (\mathcal{T}, \preccurlyeq), \alpha, \mathit{Prg})$ where

- $\mathcal{F}$ is a set of function symbols,
- $\mathcal{F}^{\text{Unique}} \subseteq \mathcal{F}$ is a set of unique function symbols,
- $\mathcal{P}$ is a set of predicate symbols,
- $\mathcal{V}$ is a set of variable symbols,
- $\mathcal{T}$ is a set of types and $\preccurlyeq \subseteq \mathcal{T} \times \mathcal{T}$ is the sub type relation,
- α is a static typing function for variables, program variables, functions, and predicates,
- and Prg is a fragment of a sequential Java program

The set of function symbols consists of *program variables* $\mathcal{PV}$ and interpreted function symbols. The set $\mathcal{F}^{\text{Unique}}$ are function symbols which are marked unique, which means for 0-ary function symbols that for pairwise different symbols they are interpreted differently.

The signature contains common boolean operators like equality, implication, equivalence and others. In line with the notation common in JavaDL, we use dotted symbols for these operators ($\doteq$ for equality, $\dot{\rightarrow}$ for implication, $\dot{\leftrightarrow}$ for equivalence, ...). In the remainder of this chapter, we introduce several functions and predicates which we assume to have a corresponding representation in the signature.

The grammar for JavaDL terms and formulas is straight forward, we only consider here the grammar of the more special operators modality and update. If *Frm* is a formula, and p is a fragment of a sequential Java program, then $\langle p\rangle Frm$ is a formula. If v is a program variable of type A and t is a Term of type A, then $v := t$ is an update; if u_1, u_2 are updates, then $u_1 \parallel u_2$ (*parallel update*), $u_1; u_2$ (*sequential update*), and $\{u_1\}u_2$ are updates. If t is a term, f is a formula, and u is an update, then $\{u\}t$ is a term and $\{u\}f$ is a formula.

The type hierarchy resembles the type hierarchy of Java. $\mathcal{T}$ contains *Any*, which is the supertype for primitive types like integers, and boolean. *Any* is also the supertype for reference types like *Object*. $\mathcal{T}$ additionally contains for a given Java program types of classes contained in the program. Parallel to *Any*, the set also contains the type *Heap* for heaps and *Field* for fields of objects. The type *Seq* is the type for the abstract sequence datatype, which represents finite lists in the logic. We will extend $\mathcal{T}$ in the remainder of this chapter with new types for events and remote methods and discuss them where this becomes relevant.

A JavaDL formula is interpreted in a *Kripke structure*, which is a tuple $(\mathrm{Dom}, \mathrm{I}, \mathrm{S}, \delta, P)$ where

- Dom is a set of values, called the *domain*,
- I is a function assigning meaning to predicate symbols in $\mathcal{P}$ and function symbols in $\mathcal{F} \setminus \mathcal{PV}$. The function is called *interpretation function.*
- S is a set of states consisting of functions assigning values to program variables in $\mathcal{PV}$,
- $\delta : \mathrm{Dom} \mapsto \mathcal{T}$ is a function assigning types to the elements in the domains, called the *dynamic typing function*,
- P associates to each program fragment a transition relation from prestates to poststates.

Free variables are interpreted by a variable assignment β, as common in first order logic. Formulas not containing updates or modalities are evaluated as in first order logic. We write $t^{D,s,\beta}$ for a term t evaluated in Kripke structure D, state s and variable assignment β. The notion $D, s, \beta \models \varphi$ states that formula φ evaluates to true in (D, s, β). The tuple (D, s) is called a *model* of φ, if $D, s, \beta \models \varphi$ for all variable assignments β.

The semantics for the diamond modality and updates is defined as follows:

- $D, s, \beta \models \langle p\rangle\varphi$ holds, if and only if there exists a state s_2 such that $(s, s_2) \in \rho$ for $\rho \in P$, the transition relation for program p, and $D, s_2, \beta \models \varphi$. Intuitively, this means the program fragment p, started in s terminates in s_2 and φ holds in the state after execution of p.
- $(\{u\}t)^{D,s,\beta} = t^{(D,s^u,\beta)}$ with $s^u = val_{D,s',\beta}(u)(s)$ where

  - $val_{D,s',\beta}(x := t)(s)$ is the state s_x^t defined as
    $s_x^t(y) \begin{cases} t^{D,s',\beta} & \text{if } y = x \\ s(y) & \text{otherwise} \end{cases}$
  - $val_{D,s',\beta}(u_1 \parallel u_2)(s) = val_{D,s',\beta}(u_2)(s'')$ with $s'' = val_{D,s',\beta}(u_1)(s)$
  - $val_{D,s',\beta}(\{u_1\}u_2)(s) = val_{D,s'',\beta}(u_2)(s)$ with $s'' = val_{D,s',\beta}(u_1)(s)$ and
  - $val_{D,s',\beta}(u_1; u_2)(s) = val_{D,s',\beta}(\{u_1\} \parallel \{u_1\}u_2)(s)$

- $D, s, \beta \models \{u\}\varphi$ holds if and only if $D, s^u, \beta \models \varphi$ holds, where s^u is defined as above as $s^u = val_{D,s,\beta}(u)(s)$

8.2.2 Fields, Heaps and Object Creation

Domain elements of type *Field* represent the fields in a Java program. The tuple (o, f) with object o and field f is called a *heap location.* A heap is a mapping from heap locations to values in the domain, where the program variable `heap` of type *Heap* represents the current heap of a Java program. The function $select_A(h, o, f)$ returns the value of the heap location (o, f) with type A on heap h, i.e. $select_A$ models a field access. Heaps in JavaDL follow the idea of the theory of arrays (Kassios [2006]). We omit here functions for storing values on the heap.

Object creation in JavaDL is modeled with the field *created*, which is *TRUE* if an object is created and *FALSE* otherwise. The value of the field *created* can not be modified directly using the store function, but instead the creation of an object is modeled by the function $create(h, o)$, yielding a heap which is equal to h except for $select_{bool}(create(h, o), o, f)$ returning *TRUE*. When a new object is created in a Java program, JavaDL assumes that a previously non-created object is selected for creation deterministicly. However, it is underspecified, which object is selected.

The function $anon(h, a, h')$, called anonymization function, is frequently used in JavaDL to characterizes underspecified changes to heap. The location set a is a set of locations denoting the locations of heap h which are at most changed. Additionally new objects may be created in h' which were not created in h. For an object o and a field f the semantics of anon is formally defined as

$$anon(h, a, h')(o, f) = \begin{cases} h'(o, f) & if(o, f) \in a and f \neq created \\ & or(o, f) \in unusedLocs(h) \\ h(o, f) & otherwise \end{cases}$$

The function *unusedLocs* describes the set of locations of non-created objects in a heap:

$$unusedLocs(h) = \{(o, f) \mid o \neq \texttt{null} \land h(o, created) \neq TRUE\}$$

A well-formed predicate $\mathit{wellformed}(h)$ evaluates to true, if for a heap h several assumptions are satisfied. Intuitively, a heap is well-formed, if it can be reached by a Java program. A heap is well-formed, if all objects stored on the heap are either created or `null`, and only finitely many objects are created.

8.2.3 Sequences

Finite sequences are modeled in JavaDL by the abstract data type *Seq* and may contain elements of different types, as long as the element is subtype of *Any*. We use $\mathit{seq}(a_1, ..., a_n)$ as a constructor of a sequence containing the elements a_1 to a_n with $a_i \in \mathrm{Dom}$. $\mathit{seqConcat}\,(s_1, s_2)$ is the concatenation of the sequences s_1 and s_2. *seqEmpty* refers to the empty sequence, $\mathit{seqGet}\,(s, i)$ returns the i-th element of s. The function $\mathit{length}(s)$ returns the length of sequence s.

8.2.4 Calculus

To reason about formulas, KeY defines a sequent calculus. Sequents consist of an *antecedent* Γ and a *succedent* Δ, each a finite set of formulas. Sequences are typically written as $\Gamma \Longrightarrow \Delta$. A sequent is semantically equivalent to the formula $\bigwedge \Gamma \rightarrow \bigvee \Delta$.

The calculus consists of a set of *schematic rules* which operate on sequents. Rules have the form

$$\frac{\Gamma_1 \Longrightarrow \Delta_1 \ldots \Gamma_n \Longrightarrow \Delta_n}{\Gamma \Longrightarrow \Delta}$$

where $\Gamma \Longrightarrow \Delta$ is the *conclusion* and $\Gamma_1 \Longrightarrow \Delta_1 \ldots \Gamma_n \Longrightarrow \Delta_n$ are the *premisses* of the rule. The sequents in schematic rules may contain *schema variables*, placeholders which are substituted by terms and formulas when a rule is applied on a concrete sequent. A rule is sound, if the universal validity of the premises implies the universal validity of the conclusion. In order to apply a schematic rule on a sequent, the sequent has to match the rule's conclusion and is replaced by the premisses, substituting the matched schema variables.

When a modality occurs in a formula, the modality typically has the form $\langle \pi p; \omega \rangle$, where π is a prefix of the program containing non-active statements, like opening brackets, labels, beginnings of try-catch blocks and others. p is the active statement, i.e. the next statement of the program to be executed, and ω is the rest of the program. Typically, rules either replace the active statement by another statement, e.g. in order to gain a single assignment form of the program, or the rule removes the active statement from the program and adds the effect of the program statement to the formula, e.g. by adding an update to the formula. Applying rules on a formula which contains a program until the formula does not contain a program anymore is called symbolic execution.

A *proof* for the universal validity of a formula f in the sequent calculus is represented by a *closed proof tree.* In a proof tree each node is annotated with a sequent, and the root note is annotated with the formula f. Each child of a node N is annotated with an instantiation of a premise of a schematic rule, whose conclusion is applicable to the sequent N is annotated by. A branch is closed if its leaf node is annotated with an axiom (a rule without a premise). A proof tree is closed if all branches of the tree are closed.

8.2.5 Java Modeling Language

The Java Modeling language (JML) (Leavens et al. [2008]) is a specification language for Java programs. We refer here to JML*, an extended version of JML (Weiß [2011]). Program specifications are written as formulas using side-effect-free Java expressions plus some additional constructs, like quantifiers and operations on abstract data types. Specifications are written directly in the program code as comments expressing pre- and postconditions of methods, class invariants, loop invariants, or others.

Example 8.3. The following example shows a simple Java class with a JML specification.

```
public class JMLExample {
  //@ public invariant o1 != o2;
  Object o1; Object o2;

  /*@ public normal_behavior
    @ requires p1 != p2;
    @ ensures \result == p1;
    */
  public Object doSth(Object p1, Object p2) {
    o1 = p1; o2 = p2;
    return o1;
  }
}
```

In Line 2 the class invariant is specified stating that the fields `o1` and `o2` must not point to the same object.

In line 5 a method contract is specified. The keyword `normal_behavior` states that the method must not throw an exception, and the keyword `requires` in Line 6 introduces the precondition of the method. The precondition states that the parameters must not point to the same object. A caller of the method has to ensure that in the state when the method is called, the invariant holds as well as the precondition. The method satisfies its method contract if in the poststate the class invariant and the postcondition hold. The postcondition is the formula following the `ensures` keyword in the JML contract (Line 7).

JML specifications can be translated into JavaDL formulas which then can be shown to be universally valid using KeY. Method contracts provide an abstract description of a program in logical form. These specifications can be translated into schematic rules, which then can be applied during verification of a formula containing a Java program. For example when verifying that a program satisfies its specification and the active statement is a method call, the proof tree can be expanded by two branches. In the first branch, it has to be shown that the precondition of the method holds and in the second branch the method's postcondition can be assumed to hold.

We assume here that there exists a translation of a JML specification into respective JavaDL formulas without discussing this translation in detail. For a full account of JML and its connection to JavaDL, the interested reader may refer to Weiß [2011] and Ahrendt et al. [2016].

8.2.6 The KeY Tool

The KeY tool [1] implements the calculus rules, rule application and the translation of Java programs and JML specifications into JavaDL formulas, rules and proof obligations. Using this tool, the correctness of Java programs w.r.t. their specification can be verified. Either KeY applies a strategy for rule selection and application automatically, or the user can step in and manually apply rules to support the proof process.

KeY ensures that the proof tree can only be manipulated by rule applications in order to ensure a sound proof. Further, proof management ensures that all required proofs are performed, i.e. if in one proof the rule gained from a method contract is applied, it also has to be shown that the respective method satisfies its method contract.

8.3 Extending JavaDL

In JavaDL a Java program fragment manipulates a state, where the state is a mapping from program variables (for example for parameters, local variables, and return values) to values in the domain. A distinct program variable is mapped to the heap. When considering distributed systems, i.e. JavaEE beans as described in Section 8.1, we make a small change to this point of view. We always consider a Java program from the point of view of one particular bean. The bean manages its own state containing an exclusive heap and all beans the program interacts with cannot manipulate this heap.

[1]The official version of the KeY tool is available online at the project website `www.key-project.org`

Additionally, the state contains a program variable which records the bean's history, i.e. the trace of events sent or received by the bean. Again, this history is local to the bean and interactions of other beans do not influence the bean's history. In the following, we extend JavaDL by the relevant elements.

8.3.1 Extending JavaDL Syntax

We extend JavaDL by events, a program variable storing the sequence of events communicated by a bean, and representations for remote methods. We assume the signature ($\mathcal{T}$, $\preccurlyeq$, $\mathcal{V}$, $\mathcal{PV}$, $\mathcal{F}$, $\mathcal{F}^{\text{Unique}}$, $\mathcal{P}$, α, Prg) of Section 8.3 additionally contains the following symbols.

- $\mathit{Event}, \mathit{Calltype}, \mathit{Method} \in \mathcal{T}$
- $\mathit{Event} \preccurlyeq \mathit{Any}$
- $\mathit{servcall} : \mathit{Calltype}, \mathit{servterm} : \mathit{Calltype} \in \mathcal{F}^{\text{Unique}}$
- For each method declared remote and visible in Prg there is a 0-ary function symbol m in $\mathcal{F}^{\text{Unique}}$ and $\alpha(m) = \mathit{Method}$ called *method identifier*
- $\mathit{event} : (\mathit{Object}, \mathit{Object}, \mathit{Method}, \mathit{Calltype}, \mathit{Seq}, \mathit{Heap}) \to \mathit{Event} \in \mathcal{F}$
- $\mathit{evCalltype} : \mathit{Event} \to \mathit{Calltype} \in \mathcal{F}$
- $\mathit{evCaller} : \mathit{Event} \to \mathit{Object} \in \mathcal{F}$
- $\mathit{evClient} : \mathit{Event} \to \mathit{Object} \in \mathcal{F}$
- $\mathit{evMethod} : \mathit{Event} \to \mathit{Method} \in \mathcal{F}$
- $\mathit{evParams} : \mathit{Event} \to \mathit{Seq} \in \mathcal{F}$
- $\mathit{evHeap} : \mathit{Event} \to \mathit{Heap} \in \mathcal{F}$
- $\mathtt{hist}, \mathtt{hist}^{\mathtt{pre}} \in \mathcal{PV}$ and $\alpha(\mathtt{hist}) = \mathit{Seq}, \alpha(\mathtt{hist}^{\mathtt{pre}}) = \mathit{Seq}$
- $\mathtt{hist}_{\mathtt{local}}, \mathtt{hist}^{\mathtt{pre}}_{\mathtt{local}} \in \mathcal{PV}$ and $\alpha(\mathtt{hist}_{\mathtt{local}}) = \alpha(\mathtt{hist}^{\mathtt{pre}}_{\mathtt{local}}) = \mathit{Seq}$
- $\mathit{wellformedHist} : \mathit{Seq}, \mathit{wellformedHist}_l : \mathit{Seq} \in \mathcal{P}$
- $\mathit{callingComp} : \mathit{Object} \in \mathcal{PV}$

We model messages as elements in the domain of type *Event*, a subtype of *Any*. Domain elements of type *Method* are representatives for Java methods declared remote. The constructor function *event* yields an event containing the object identity of the bean calling a remote method, the identity of the called bean, the identifier of the called remote method, and a marker whether the message represents the call of the remote method or its termination. Further, the event contains the values communicated by the messages, i.e. a sequence of the parameters of a call or the communicated return value for a termination message. Finally, the event contains a heap in which a field access on the communicated parameters and return value can be resolved.

Note that the heap is not actually communicated in a message. We will see later that during communication, beans exchange messages containing the parameters of remote method calls, as well as the transitive closure of the fields of the parameters. In order to represent the communicated values of messages, we make the heap in which field values can be evaluated an element of the event.

We assume the logic to contain selector functions for retrieving from an event the calling bean (*evCaller*), the client bean (*evClient*), the method identifier (*evMethod*), whether it is a call or termination event (*evCalltype*), the sequence of parameters/return value (*evParams*), and the heap for resolving field access to the parameters and return value (*evHeap*).

Additional program variables $\mathtt{hist}, \mathtt{hist}^{\mathtt{pre}}, \mathtt{hist}_{\mathtt{local}}, \mathtt{hist}^{\mathtt{pre}}_{\mathtt{local}}$ represent the history of a bean. The program variable `hist` contains all events sent or received by a bean over its lifetime, while $\mathtt{hist}_{\mathtt{local}}$ represents the history communicated by the execution of a single service without the calling event starting the execution of the service and the respective termination event.

While after the execution of a remote method, `hist` does contain all events also contained in $\mathtt{hist}_{\mathtt{local}}$, introducing $\mathtt{hist}_{\mathtt{local}}$ simplifies some proof obligations as described in the remainder of this chapter. The predicates *wellformedHist* and $\mathit{wellformedHist}_l$ express a wellformedness property for histories and local histories, similar to the wellformedness predicate for heaps. The program variable *callingComp* represents the remote bean initiating a service call.

8.3.2 Serialization and Deserialization

Containers perform remote method calls by copying parameters and return values, and possibly communicating them remotely as byte-representations of the parameters. The Java API EJB 3.1 Expert Group [2009] provides a functionality called *serialization* that translates Java object structures into a byte representation, e.g. for persistently storing information or transmitting information in distributed systems. The reverse functionality, *deserialization*, translates the byte representation back into an Java object structure that can be used in a Java program. JavaEE uses serialization and deserialization for the transmission of parameters and return values in remote method calls. In the following we provide a JavaDL formalization of serialization and deserialization.

Effects of Serialization on the Heap

During serialization, for all objects in the object structure a unique representation is created and stored together with the type of the object. Additionally, for each object and each field the value of this field is stored. If the type of the field is *Object*, the respective object representation is stored (or `null`); if the field is of primitive type, the primitive value is stored.

During deserialization, for each object in the byte representation a new object of the previously stored type is created. For each object the values of the fields are set to the value as stored in the byte representation. If the field type is a reference type, its value is set to the newly created object. If its type is primitive, the field's value is set to the respective primitive value. We ignore here details like transient fields, we assume they are not used in a program.

Example 8.4. In the following example, we illustrate some properties of serialization and deserialization of remote method calls before describing these functionalities formally. Although we have not yet formally introduced service contracts, we use them in this example for illustration purposes.

The method `doSth` implemented by class `SerialExample` calls the remote method `callMethTo` and provides as a parameter a transfer object. This remote method sets the field `i` of the transfer object to 1 and returns the object. The method `doSth` satisfies its contract.

```
public class SerialExample {
  /*@ invariant to.o1 == to.o2 && to.o1 != null &&
    @           to.o2 != null;*/
  private TransObj to;
  private int x;
  @EJB
  private RemoteIF r;

  /*@ public normal_behavior
    @ requires true;
    @ ensures this.to.o1 != \old(this.to.o1) &&
    @    this.to.o2 != \old(this.to.o2) &&
    @    x == \old(x) &&
    @    \fresh(this.to) && \result == 1; */
  public int doSth() {
    this.to = r.callMethTo(to);
    return to.i;
  }
}

@Remote
public Interface RemoteIF {
 /*@ public normal_behavior
   @ requires true;
   @ ensures \result == t && \result.o1 == \old(t.o1) &&
   @ \result.o2 == \old(t.o2) && \result.i==1 &&
   @ \result.o1 != null && \result.o2 != null; */
 public TransObj callMethTo(TransObj t);
}

public class TransObj {
  Object o1; Object o2; int i; }
```

The example illustrates several properties of serialization and deserialization. For example, the contract of `callMethTo` guarantees in its postcondition that it returns the same objects `o1` and `o2`, as it received with the parameter `t`. The postcondition of `doSth` however guarantees that after its execution the values of these fields have changed. The postcondition of `doSth` also guarantees that the new value of the field `to` is a freshly created object.

Further, the postcondition of `doSth` guarantees that the value of field `x` does not change. The contract of `callMethTo`, however, does not provide any explicit guarantee on how it changes the heap apart from the fields of the transfer object.

The sequential serialization and deserialization of an object structure ensures an isomorphic relation between the original object structure and the new, deserialized one. Let's assume, a Java program calls a remote method with the values $\{o_1, ..., o_m\}$ as parameters in heap h_1, which leads to a call to the respective remote method of the client bean with heap h_2. During deserialization, the heap h_2 is changed into a heap h_2' by creating the object structure of the values in the heap using freshly created objects.

We refer to the heap h_2' gained by serialization of $\{o_1, ..., o_m\}$ in heap h_1 and deserialization in heap h_2 by $\mathit{deserial}(h_2, h_1, \{o_1, ..., o_m\})$. Serialization and deserialization ensure several properties for h_2' with respect to h_1 and h_2 that we can use for reasoning in our logic.

For one, deserialization does not change already existing objects or their fields on the heap h_2, but only creates new ones.

Definition 8.1 (Deserialization Anonymization)**.** Given heaps h_1, h_2, and values $o_1, \ldots, o_m$. For the heap $h_2' = \mathit{deserial}(h_2, h_1, \{o_1, ..., o_m\})$ it holds $\mathit{deserial}(h_2, h_1, \{o_1, ..., o_m\}) \doteq \mathit{anon}(h_2, \emptyset, h)$ for some unspecified heap h.

Second, serialization and deserialization of $\{o_1, ..., o_m\}$ ensures a partial isomorphism $\gamma : \mathit{Any} \mapsto \mathit{Any}$ which maps the original values in the original data structure to those in the deserialized data structure, i.e. $\gamma(o_i) = o_i'$ where o_i' is the value created by deserialization of o_i. The mapping γ ensures that the data structure in the heap h_1 is preserved for the heap h_2'. The following definition states the properties of γ:

Definition 8.2 (Serialization Isomorphism)**.** Given heaps h_1, h_2, and values $o_1, \ldots, o_m$. For the heap $h_2' = \mathit{deserial}(h_2, h_1, \{o_1, ..., o_m\})$ there exists a partial isomorphism γ from values in h_1 to values in h_2' such that

- $x \doteq y \Leftrightarrow \gamma(x) \doteq \gamma(y)$ for all $x, y : \mathit{Any}$

- $\gamma(\texttt{null}) \doteq \texttt{null}$
- $o \cong \gamma(o)$ for all $o \in \{o_1, \ldots, o_m\}$

where $o \cong \gamma(o)$ is defined as

$$\begin{array}{ll}
\gamma(o) \doteq o & \text{if } \delta(o) \in \mathit{Primitive} \\
\gamma(\mathit{length}(o)) \doteq \mathit{length}(\gamma(o)) & \text{if } o \text{ is of array type} \\
\left.\begin{array}{l}
\delta(o) \doteq \delta(\gamma(o)) \wedge \\
\forall f : \mathit{Field} \cdot \\
\quad \gamma(\mathit{select}_E(h_1, o, f)) = \mathit{select}_E(h_2', \gamma(o), f) \wedge \\
\quad \mathit{select}_E(h_1, o, f) \cong \mathit{select}_E(h_2', \gamma(o), f)
\end{array}\right\} & \begin{array}{l}\text{if } \delta(o) \preccurlyeq \mathit{Object} \wedge \\ \quad o \not\doteq \texttt{null}\end{array}
\end{array}$$

The first two conditions formalize that γ is indeed an isomorphism. The third condition in combination with the definition of $\cong$ states that γ is the identity function for primitive values, that isomorphic array objects have equal length and for isomorphic objects, all fields store respectively isomorphic values. Note that we treat array accesses here as field access with the index of the array access as the field.

Third, the entire object structure of serialized objects are freshly created, especially this means that every field of an object that is created during deserialization holds a primitive value, a reference to `null`, or a reference to a freshly created object. We use the predicate $\mathit{transfresh}(o, h_2, h_2')$ to state for a object o that it is not created in h_2, and that transitively the same holds for all references stored in fields of o evaluated in h_2' .

Definition 8.3 (Serialization Freshness)**.** Given heaps h_1, h_2, and values $o_1, \ldots, o_m$. For the heap $\mathit{deserial}(h_2, h_1, \{o_1, \ldots, o_m\})$ it holds

$$\forall o \in \{o_1, ..., o_m\} \cdot \mathit{transfresh}(\gamma(o), h_2, \mathit{deserial}(h_2, h_1, \{o_1, ..., o_m\}))$$

with

$$\begin{array}{l}
\mathit{transfresh}(o, h_2, h_2') :\Leftrightarrow \delta(o) \preccurlyeq \mathit{Object} \wedge o \not\doteq \texttt{null} \implies \\
\qquad\qquad\qquad \mathit{select}_{\mathit{bool}}(h_2, o, \mathit{created}) \not\doteq \mathit{TRUE} \\
\qquad\qquad\qquad\quad \wedge \forall f : \mathit{Field} \cdot \mathit{transfresh}(\mathit{select}_{\mathit{Any}}(h_2', o, f), h_2, h_2')
\end{array}$$

The three properties of a sequential execution of serialization and deserialization as formalized in Definition 8.1, 8.2, and 8.3 follow directly from the semantics of Java serialization and deserialization.

Optimizations for Serialization

The properties for serialization and deserialization as formalized above have to be represented in the sequent of a proof. Especially when a remote method has several parameters, and symbolically executing a remote method call would lead to very big sequents with many nested quantifier. As a result, performing a proof would become very time consuming. We therefore provide an optimization for remote method calls in the following.

The selection of which object identity is chosen during object creation is up to Java and can not be observed by a Java program. We can assume that for two beans that whenever a new object is created, object identities are always chosen in a way such that in the heaps of the two beans no object with the same identity chosen. We therefore can assume that the two two heaps are completely disjoint in the sense that whenever one object is created in a heap h_1, the same object is not created in the other heap h_2, and vice versa, i.e. $\mathit{select}_{\mathit{bool}}(h_1, o, \mathit{created}) \dot{\rightarrow} \dot{\neg}(\mathit{select}_{\mathit{bool}}(h_2, o, \mathit{created}))$ and $\mathit{select}_{\mathit{bool}}(h_2, o, \mathit{created}) \dot{\rightarrow} \dot{\neg}(\mathit{select}_{\mathit{bool}}(h_1, o, \mathit{created}))$. In this case, it is sound to assume w.l.g. that the objects created during deserialization in the heap h_2 are exactly those objects used in h_1. We then can assume γ to be the identity function, which results in simpler proofs during program verification, which we will see later in this chapter.

We introduce a new function $\oplus : (\mathit{Heap}, \mathit{Heap}, \mathcal{P}(\mathit{Object})) \mapsto \mathit{Heap}$ to join two disjoint heaps. In the following definition, *p-expression* characterizes heap access expressions starting with a parameter and *self-expression* characterizes heap access expressions starting with `self`. We define the semantics for field access in a joined heap as follows:

Definition 8.4 (Joined Heaps)**.** For heaps h_1 and h_2 and the finite set of objects $\{p_1, ..., p_m\}$ the semantics of the joined heap $h_j = \oplus(h_1, h_2, \{p_1, ..., p_m\})$ is

$$\mathit{select}_E(h_j, o, f) = \begin{cases} \mathit{select}_E(h_1, o, f) & \text{if } o \text{ is } \textit{p-expression} \text{ w.r.t. } h_j \\ \mathit{select}_E(h_2, o, f) & \text{if } o \text{ is } \textit{self-expression} \text{ w.r.t. } h_j \\ \text{undefined} & \text{otherwise} \end{cases}$$

where o is *p-expression* w.r.t. h_j iff

- $o \in \{p_1, ..., p_m\}$ or
- $o = \mathit{select}_{E'}(h_1, o', f)$ and o' is *p-expression* w.r.t. h_j or
- $o = \mathit{select}_{E'}(h_j, o', f)$ and o' is *p-expression* w.r.t. h_j

and o is *self-expression* w.r.t. h_j iff

- $o = o_{\mathit{self}}$, where o_{self} refers to the object identity of the bean with heap h_2 or

- $o = \mathit{select}_{E'}(h_2, o', f)$ and o' is *self-expression* w.r.t. h_j or
- $o = \mathit{select}_{E'}(h_j, o', f)$ and o' is *self-expression* w.r.t. h_j

In Definition 8.4 heap access-expressions are evaluated in one of the two disjoint heaps, depending on whether they represent a field access starting with the `this`, i.e. the called bean, or with a parameter, i.e. the objects created during deserialization. Expressions starting with `self` are evaluated in the heap of a bean before serialization, while expressions staring with a parameter are evaluated on the heap of the sending bean. Note that in JavaEE remote method calls must not communicate references to beans as parameters. Therefore, Definition 8.4 is well-defined.

Under the assumption that γ is the identity function, evaluation of *self-expressions* and *p-expressions* yield the same values for a joined heap as for a direct encoding of the heap gained by deserialization as defined in Definition 8.1, 8.2, and 8.3.

Lemma 8.1. *Given two disjoint heaps h_1 and h_2, a set of values $\{p_1, ..., p_m\}$, a term $t = \mathit{select}_A(\mathbf{heap}, o, f)$, program variables $o_1, \dots, o_m$, and a state s such that $s(o_i) \doteq p_i$. If t is a p-expression or self-expression w.r.t. $\{p_1, ..., p_m\}$, then the formula*

$$\{\mathbf{heap} := \oplus(h_1, h_2, \{o_1, ..., o_m\})\}t \doteq \{\mathbf{heap} := \mathit{deserial}(h_2, h_1, \{o_1, ..., o_m\})\}t$$

is universally valid in s.

We do not provide a full proof for the lemma, but limit the presentation to a proof-sketch for the case when the term is a heap access. Note that we assume the extension to other expressions, especially to function symbols, predicates and quantifier to be relatively straight forward, although laborious and without explicit benefit.

Proof sketch for Lemma 8.1. We proof Lemma 8.1 by induction over the length of the formula t.

Induction start: Case a: $t = \mathit{select}_A(\texttt{heap}, \texttt{self}, f)$. After applying the update, we gain $t = \mathit{select}_A(\mathit{deserial}(h_2, h_1, \{p_1, ..., p_m\}), \texttt{self}, f)$. Since the heap gained by deserialization is an anonymization of h_2 and `self` was already created in h_2, we gain $t = \mathit{select}_A(h_2, \texttt{self}, f)$, which is equal to the unrolling of the definition for $\{\texttt{heap} := \oplus(h_1, h_2, \{p_1, ..., p_m\})\}t$.

Case b: $t = \mathit{select}_A(\texttt{heap}, p_i, f)$. After applying the update, we gain $t = \mathit{select}_A(\mathit{deserial}(h_2, h_1, \{p_1, ..., p_m\}), p_i, f)$. We know by definition of deserialization above that $p_i \cong \gamma(p'_i)$ for some parameter variable o_i, and since γ is the identity function, we know that $p_i = p'_i$. Also from conditions for the *deserial* operator, we know
$\gamma(\mathit{select}_A(h_1, p_i, f)) \doteq \mathit{select}_A(\mathit{deserial}(h_2, h_1, \{p_1, ..., p_m\}), p_i, f)$, which is the same term we gain for $\{\texttt{heap} := \oplus(h_1, h_2, \{p_1, ..., p_m\})\}t$.

Additionally, Definition 8.2 provides us with
$\mathit{select}_A(h_1, p_i, f) \cong \mathit{select}_A(\mathit{deserial}(h_2, h_1, \{o_1, ..., o_m\}), p_i, f)$.

Induction step Case a: $t = \mathit{select}_A(\mathtt{heap}, o, f)$ and o is a *self-expression* w.r.t. $\oplus(h_1, h_2, \{p_1, ..., p_m\})$. With the same argument above, i.e. that o must have been created in h_2 and *deserial* is an anonymization operator, no field of o must have been changed during deserialization. Therefore $\mathit{select}_A(\mathit{deserial}(h_2, h_1, \{p_1, ..., p_m\}), o, f) = \mathit{select}_A(h_2, o, f)$. Again, this is the same term we gain by unrolling the definition of the $\oplus$ operator.

Case b: $t = \mathit{select}_A(\mathtt{heap}, o, f)$ and
o is a *p-expression* w.r.t. $\oplus(h_1, h_2, \{p_1, ..., p_m\})$. Since o is a *p-expression*, we know by induction that
$\mathit{select}_A(h_1, o, f) \cong \mathit{select}_A(\mathit{deserial}(h_2, h_1, \{p_1, ..., p_m\}), o, f)$ has to hold (Note that γ is the identity). By the definition of $\cong$ we gain
$\gamma(\mathit{select}_A(h_1, o, f)) \doteq \mathit{select}_A(\mathit{deserial}(h_2, h_1, \{p_1, ..., p_m\}), \gamma(o), f)$ and thus
$\gamma(\mathit{select}_A(h_1, o, f)) \doteq \mathit{select}_A(\mathit{deserial}(h_2, h_1, \{p_1, ..., p_m\}), , f)$ and
$\gamma(\mathit{select}_A(h_1, t, f')) \cong \mathit{select}_A(\mathit{deserial}(h_2, h_1, \{p_1, ..., p_m\}), t, f')$.
Again, $\gamma(\mathit{select}_A(h_1, o, f))$ is what we gain by unrolling the definition for $\oplus$. $\lhd$

8.3.3 Service Contract

We have to ensure that the distinct program variables `hist` and $\mathtt{hist}_{\mathtt{local}}$ during reasoning with JavaDL reflect the actual trace of events communicated by a bean. JavaDL deals with Java programs by symbolic execution, i.e. by applying rules reflecting the semantics of the Java language formulas are modified such that the effect of a Java statement is reflected by changes to the formula. We have to ensure that during symbolic execution of remote method calls the history program variables are updated properly. JavaDL knows two ways to deal with the symbolic execution of methods in general.

The first technique is inlining, where a method call is replaced during symbolic execution by the method body which then again can be symbolically executed. Since in JavaEE remote method calls are typically performed on methods declared in interfaces, the implementation is not available. Also, the method is executed in another state, which would additionally complicate the rule for replacing a remote method call by its implementation while reflecting JavaEE semantics.

The second technique is the application of method contracts, i.e. the method call is replaced by an abstraction, typically generating two branches in the proof tree. In the first branch, it has to be shown that in the state in which the method is called, the precondition of the called method is satisfied. In the second branch it can be assumed that the method terminates in a state where its postcondition holds and the remaining program can be symbolically executed in this state. Additionally, it has to be shown in a separate proof that the method satisfies its postcondition after termination if it is started in a state when its precondition holds.

We introduce in this subsection *service contracts*, a modification of method contracts which reflect the semantics of JavaEE for remote method calls. During verification, service contracts can be used for symbolic execution of remote method calls. In the following, we develop the proof obligation which has to be valid in order to ensure that a service satisfies its contract. Finally, we introduce the JavaDL rule for symbolic execution of remote method calls using service contracts.

In JML, the functional specification for a method is given as a method contract. A method contract *mct* according to Weiß [2011] is a tuple

$$(m, \mathtt{self}, (a_1, \ldots a_m), \mathtt{res}, \mathtt{heap}^{\mathtt{pre}}, \mathtt{exc}, \mathit{pre}, \mathit{post}, \mathit{mod})$$

A method contract expresses that, assuming method m is executed with parameters $(a_1, \ldots a_m)$ in a state in which the precondition *pre* holds, the method terminates in a state such that the postcondition *post* holds. Additionally, the execution at most changes the heap on the elements described in a location set *mod*. The method contract also describes the variables used in *pre* and *post* to refer to the return value (`res`), the heap before method execution ($\mathtt{heap}^{\mathtt{pre}}$), and the exception which may be thrown (`exc`). For a full description of method contracts, we refer to Weiß [2011].

For service contracts, we restrict the method m to those methods which are declared remote, since only remote methods represent services. We extend method contracts by a program variable referring to the history before execution ($\mathtt{hist}^{\mathtt{pre}}$), and a callable set *callable* describing the clients and the remote methods which may at most be called during execution of a remote method.

Example 8.5. The following service contract expresses that the remote method `pay` must at most call the remote method `registerSale` provided by the bean to which `sale` points.

```
/*@ public normal_behavior
  @ requires true;
  @ ensures true;
  @ callable sale.registerSale; */
 public int pay(int ccnr) {
    countpay++;
    sale.registerSale(product, prodprice,
                             prodamount, ccnr);
    return 0;
  }
```

The formal definition for service contracts is as follows:

Definition 8.5 (Service Contract)**.** A service contract *sct* is a tuple

$$(m, \texttt{self}, (a_1, \dots a_m), \texttt{res}, \texttt{heap}^{\texttt{pre}}, \texttt{hist}^{\texttt{pre}}, \texttt{exc},\\ \qquad\qquad pre_{noinv}, inv, post, mod, callable)$$

where m is a Java method declared remote in type $C \in \mathcal{T}$ with argument types $A_1, \dots A_m \in \mathcal{T}$ and return type $A \in \mathcal{T}$; where $\texttt{self} : D \in \mathcal{PV}$ for some $D \preccurlyeq C$, where $a_1 : A_1, \dots a_m : A_m \in \mathcal{PV}$; where $\texttt{res} : A \in \mathcal{PV}$; where $\texttt{heap}^{\texttt{pre}} : \mathit{Heap} \in \mathcal{PV}$; where $\texttt{hist}^{\texttt{pre}} : \mathit{Seq} \in \mathcal{PV}$; where $\texttt{exc} : \mathit{Exception} \in \mathcal{PV}$; where $pre_{noinv} \in \mathit{Fma}_\Sigma$ is the precondition of the method contract without the class invariant; $inv \in \mathit{Fma}_\Sigma$ is the class invariant of the class in which m is declared; $post \in \mathit{Fma}_\Sigma$ is the postcondition of m, including the class invariant; where mod is a list $(e_1, \dots, e_k)$ with $e_1, \dots e_k \in \mathit{LocSet}$ and $callable$ is a list $(exp_1.m_1, \dots, exp_n.m_n)$ with $exp_1, exp_n \in \mathit{LocSet}$ and $m_1 : \mathit{Method}, \dots m_n : \mathit{Method}$.

Service Contract Validity

In order to soundly apply service contracts during symbolic execution of a program, we have to ensure that the called remote method actually satisfies its service contract.

Before we provide the formal definition of the proof obligation, we introduce the intuitive meaning of some variables used in the definition and motivate parts of the proof obligation.

The heap of the called bean before the call of the remote method is represented by the program variable `heap`. We assume that all remote methods preserve the class invariant *inv*, which means that we can assume the invariant to hold in `heap`.

Deserialization of the remote method parameters causes the heap of the called bean to change, and we denote the heap after deserialization by the variable $heap_{pserial}$. Due to the properties of deserialization, as discussed above, we know that the heap after deserialization is equal to the heap before deserialization, except that new objects may have been created. We therefore can assume that $heap_{pserial} \doteq anon(\texttt{heap}, \emptyset, h)$. The heap h represents some underspecified heap.

Again due to the properties of deserialization, we know that the parameters, as well as fields of parameters reference freshly created objects, if they are of some reference type. We introduced the predicate *transfresh* above to express this property. So, we can assume for all parameters a_i: $transfresh(\texttt{heap}, a_i, heap_{pserial})$

Further, the calling component has to ensure that the precondition of the remote method holds. Opposed to the original method contract, the calling component should not be required to provide guarantees about the internal state of the bean, therefore, we only consider here the precondition without the class invariant. So, we can assume $\{\texttt{heap} := \mathit{heap}_{\mathit{pserial}}\}\mathit{pre}_{\mathit{noinv}}$.

Two program variables are used to represent the event history of the called bean. The program variable `hist` records the overall history of the bean, while $\texttt{hist}_{\texttt{local}}$ records the history caused by a single remote method execution. Before the call, we can assume $\texttt{hist}_{\texttt{local}}$ to be the empty sequence, i.e. $\texttt{hist}_{\texttt{local}} = \mathit{seqEmpty}$.

In order to add the events caused by the call of the remote method and its termination, we require a representation of the communication partner, i.e. the calling bean. We represent the calling bean by some underspecified variable *callingComp*, and all we assume about this object is that it must not be `null` and is different than the called bean, i.e. $\mathit{callinComp} \neq \texttt{self}$.

Before we can symbolically execute the body of the remote service, we have to set the state such that it represents the state after the calling event was received. We do this in an update, which contains the respective update assignments. The update $\texttt{heap} := \mathit{heap}_{\mathit{pserial}}$ sets the heap before execution to the heap after deserialization of the parameters. The update $\texttt{hist} := \mathit{seqConcat}(\texttt{hist}, \mathit{callevent})$ adds the call event to the history. The postcondition of the remote method may reference the history and heap before the execution, thus we have to remember them with the updates $\texttt{heap}^{\texttt{pre}} := \mathit{heap}_{\mathit{pserial}}$ and $\texttt{hist}^{\texttt{pre}} := \texttt{hist}$.

After symbolic execution of the body of the remote method, the termination event is sent to the calling component, and thus added to the history of the remote method. In the proof obligation, this is represented by the update $\texttt{hist} := \mathit{seqConcat}(\texttt{hist}, \mathit{termevent})$ after the modality containing the body of the remote method.

It has to be shown, that after execution of the remote method, the postcondition holds (including the class invariant). Apart from other conditions, which are equal to those in the original proof obligation of method contracts, it additionally has to be shown that the `callable` clause of the contract is satisfied. This is represented by the formula *calls*. We also require remote methods to terminate normally, i.e. the service must not throw an exception. It has to be shown that after execution $\texttt{exc} = \texttt{null}$ holds.

Additionally, we assume several well-formed properties for heaps and histories used in the formula.

The following definition states the complete formal proof obligation for a service contract.

Definition 8.6 (Proof Obligation for Service Contracts)**.** Let *sct* be a service contract, such that $\mathit{sct} =$

$$(m, \texttt{self}, (a_1, \ldots a_m), \texttt{res}, \texttt{heap}^{\texttt{pre}}, \texttt{hist}^{\texttt{pre}}, \texttt{exc}, \\ \mathit{pre}_{\mathit{noinv}}, \mathit{inv}, \mathit{post}, \mathit{mod}, \mathit{callable})$$

with `self` : D and given type $E \in \mathcal{T}$ with $E \preccurlyeq D$, $mod = \{l_1, \ldots, l_n\}$ with $l_i \in LocSet$, $callable = (b_1.m_1, ..., b_k.m_k)$ with $b_i \in LocSet$ and $\delta(m_i) = Method$ and m_i is declared remote, the proof obligation formula $CorrectServiceContract(sct, E)$ is defined as

$$
\begin{aligned}
& \mathit{wellformed}(\mathtt{heap}) \land \mathit{wellformed}(\mathit{heap_{pserial}}) \land \mathit{wellformed}(h) && (8.1)\\
& \land\ \mathit{wellformedHist}(\mathtt{hist}) \land \mathtt{hist_{local}} = \mathit{seqEmpty} && (8.2)\\
& \land\ \mathtt{self} \mathrel{\dot{\neq}} \mathtt{null} \land \mathtt{bean} \doteq \mathtt{self} \land \mathit{exactInstance}_E(\mathtt{self}) && (8.3)\\
& \land\ \mathit{select_{bool}}(\mathtt{heap}, \mathtt{self}, \mathit{created}) \doteq \mathit{TRUE} \land \mathit{inv} && (8.4)\\
& \land\ \mathit{wellformed}(h) \land \mathit{heap_{pserial}} \doteq \mathit{anon}(\mathtt{heap}, \dot{\emptyset}, h) && (8.5)\\
& \land\ \mathit{callingComp} \mathrel{\dot{\neq}} \mathtt{null} \land \mathit{callingComp} \mathrel{\dot{\neq}} \mathtt{self} && (8.6)\\
& \land\ \mathit{select_{bool}}(\mathtt{heap}, \mathit{callingComp}, \mathit{created}) \doteq \mathit{TRUE} && (8.7)\\
& \land\ \mathit{transfresh}(a_1, \mathtt{heap}, \mathit{heap_{pserial}}) \land \ldots && (8.8)\\
& \land\ \mathit{transfresh}(a_m, \mathtt{heap}, \mathit{heap_{pserial}}) && (8.9)\\
& \land\ \{\mathtt{heap} := \mathit{heap_{pserial}}\}(\mathit{pre_{noinv}} \land \mathit{reachableIn}) && (8.10)\\
& \dot{\rightarrow}\quad \{\mathtt{heap} := \mathit{heap_{pserial}} \parallel \mathtt{heap^{pre}} := \mathit{heap_{pserial}} && (8.11)\\
& \qquad \parallel \mathtt{hist} := \mathit{seqConcat}\,(\mathtt{hist}, \mathit{callevent}) \parallel \mathtt{hist^{pre}} := \mathtt{hist} && (8.12)\\
& \qquad \parallel \mathtt{hist_{local}} := \mathit{seqEmpty}\} && (8.13)\\
& \qquad \langle \mathtt{exc\ =\ null;} && (8.14)\\
& \qquad \mathtt{try\ \{res\ =\ self.m(}a_1, ..., a_m\mathtt{);\ \}} && (8.15)\\
& \qquad \mathtt{catch(Exception\ e)\ \{\ exc\ =\ e;\}}\rangle && (8.16)\\
& \qquad \{\mathtt{hist} := \mathit{seqConcat}\,(\mathtt{hist}, \mathit{termevent})\} && (8.17)\\
& \qquad (\mathit{post} \land \mathit{frame} \land \mathit{calls} \land \mathit{wellformedHist}(\mathtt{hist}) \land \mathtt{exc} \doteq \mathtt{null}) && (8.18)
\end{aligned}
$$

where:

- $\mathit{frame} \in \mathit{Fma}_\Sigma$ is the formula

$$
\begin{aligned}
&\forall \mathit{Object}\ o; \forall \mathit{Field}\ f;\\
&\qquad ((o, f) \mathrel{\dot{\in}} \{\mathtt{heap} := \mathtt{heap^{pre}}\}\mathit{mod} \mathbin{\dot{\cup}} \mathit{unusedLocs}(\mathtt{heap^{pre}})\\
&\qquad \lor\ \mathit{select_{Any}}(\mathtt{heap}, o, f) \doteq \mathit{select_{Any}}(\mathtt{heap^{pre}}, o, f))
\end{aligned}
$$

- $\mathit{calls} \in \mathit{Fma}_\Sigma$ is the formula

$$
\begin{aligned}
&\bigwedge_{i \in \{0, \ldots, \mathit{length}(\mathtt{hist_{local}})\}} \ \bigvee_{b_j.m_j \in \mathit{callable}}\\
&\qquad (\mathit{evMethod}(\mathit{seqGet}(\mathtt{hist_{local}}, i)) \doteq m_j\\
&\qquad \land \{\mathtt{heap} := \mathtt{heap^{pre}}\}(\mathit{evClient}(\mathit{seqGet}(\mathtt{hist_{local}}, i)) \doteq b_j))
\end{aligned}
$$

- *reachableIn* $\in Fma_\Sigma$ is the formula

$$\begin{aligned}&\bigwedge_{i\in\{1,\dots,n\},\delta(l_i)\preccurlyeq Object} (l_i \doteq \mathtt{null} \lor l_i.\,created \doteq TRUE)\\ \land &\bigwedge_{i\in\{1,\dots,n\},\delta(l_i)=LocSet} (disjoint(l_i, unusedLocs(\mathtt{heap})))\end{aligned}$$

- *callevent* $\in Term_\Sigma$ is the term
  $event(callingComp, \mathtt{self}, m_{id}, call, seq(a_1, \dots, a_m), heap_{pserial})$
- *termevent* $\in Term_\Sigma$ is the term
  $event(callingComp, \mathtt{self}, m_{id}, term, seq(\mathtt{res}), \mathtt{heap})$
- $m_{id} \in Method$ is the method identifier for remote method `m`

We say a remote method satisfies its service contract *sct*, if the formula in Definition 8.6 is valid.

Service Contract Application

We can use service contracts as an abstraction of the effect of a remote method call curing symbolic execution. In the following, we explain the service contract rule before formally defining it.

Applying the service contract rule in a proof leads to a split of the proof into two branches. In one branch it has to be shown that the precondition of the called service is satisfied. In the second branch, we can assume the postcondition of the service contract to hold for the remaining proof.

Precondition Branch: The precondition of the service contract has to hold in the heap of the called bean. We therefore use the fresh heap variable $heap_o$ to represent the heap of the called bean. Similarly, the fresh history variable $hist_o$ represents the history of the called bean.

Due to serialization and deserialization of the remote method call and its parameters, the heap of the called bean changes. We refer to with this changed heap with the fresh variable $heap_{opre}$. In this heap, the parameters are transitively fresh created, as explained earlier in this section, such that we can assume $transfresh(a_i, heap_{opre}, heap_o)$ for all parameters a_i. Further, the heap after deserialization of the remote method call is the joined heap according to Definition 8.4, such that we can assume $heap_{opre} \doteq \oplus(\mathtt{heap}, heap_o, \{a_1, \dots, a_m\})$, where $a_1, \dots, a_m$ are the parameters of the remote method call.

Since all beans are ensured to execute remote methods sequentially and all remote methods preserve the invariant, we know that before deserialization, i.e. in $heap_o$ and history $hist_o$, the class invariant *inv* of the called bean holds. After deserialization, i.e. in $heap_{opre}$ and history $hist_{opre}$, it has to be shown that the precondition without the invariant pre_{noinv} of the called remote method holds.

Postcondition Branch: In the second branch, where we resume the original proof, we can assume the called remote method's postcondition, have to update the history, and resume the symbolic execution. The postcondition, however holds from the point of view of the called bean, i.e. in the heap of the called bean, while symbolic execution resumes in the heap caused by deserialization of the return value.

We therefore again introduce fresh variables representing the heap of the called bean before ($heap_o$) and after deserialization of the parameters ($heap_{opre}$) and after termination of the called remote method ($heap_{opost}$). The heaps have the same properties as in the first branch, i.e. $heap_{opre} \doteq \oplus(\mathtt{heap}, heap_o, \{a_1, \ldots, a_m\})$.

Similarly we can express the histories. The history of the remote method after execution of the remote service is extended by the calling event and termination event from the perspective of the called bean. It is also extended by some local history ($hist_{olocal}$) which may contain events referring to calls to remote methods by the called bean. So, we get for the history of the called bean after execution of the remote method $hist_{opost} \doteq seqConcat(hist_o, callevent_o, hist_{olocal}, termevent_o)$.
The events $callevent_o$ and $termevent_o$ represent the calling and terminating events caused by the symbolically executed remote method call from the perspective of the called bean.

From the perspective of the bean whose implementation is currently symbolically executed, we refer to the heap after deserialization of the return value by $heap_{post}$. Compared to the heap before the remote method call ($\mathtt{heap}$), at most new objects may be created during deserialization, so we can assume $heap_{post} \doteq anon(\mathtt{heap}, \emptyset, h_2)$. Additionally, we know that the return value (r) is transitively fresh created, i.e. we can assume $transfresh(r, heap_{post}, \mathtt{heap})$.

We also know that serialization and deserialization of the return value preserves the data structure of the return value, as defined in Definition 8.2. We therefore introduce a fresh function γ, representing the mapping from the return value from the perspective of the called bean (r_o) to the return value from the perspective of the calling bean, so we can assume that $r \doteq \gamma(r_o)$. The predicate $\cong$ encodes that two values have an isomorphic structure and we assume $r \cong r_o$.

Finally, we can assume that the postcondition ($post$) of the remote method holds in the poststate (established by the update u_{opost} in the definition below) of the remote bean, and we can establish the poststate from the perspective of the calling bean (u_{post} in the definition). Note that the update of the histories of the calling bean, i.e. $\mathtt{hist}$ and $\mathtt{hist}_{\mathtt{local}}$ is performed directly in the update u_{post}.

The following definition shows the full formal definition of the service contract rule.

Definition 8.7 (Service Contract Rule).

$$\frac{\begin{array}{l}
\Gamma \Longrightarrow \{u\}\{w\}(\mathit{wellformed}(\mathit{heap}_o)(\wedge\ \mathit{wellformed}(\mathit{heap}_{opre}) \\
\qquad\qquad \wedge\ \mathit{transfresh}(a_1, \mathit{heap}_{opre}, \mathit{heap}_o) \wedge \dots \\
\qquad\qquad \wedge\ \mathit{transfresh}(a_m, \mathit{heap}_{opre}, \mathit{heap}_o) \\
\qquad\qquad \wedge\ \mathit{wellformedHist}(\mathit{hist}_o) \\
\qquad\qquad \wedge\ \mathit{heap}_{opre} \doteq \oplus(\texttt{heap}, \mathit{heap}_o, \{a_1, \dots, a_m\}) \\
\qquad\qquad \wedge\ \mathit{callevent}_o \doteq \mathit{event}(\mathit{callingComp}, \texttt{self}, m_{id}, \mathit{call}, \\
\qquad\qquad\qquad\qquad \mathit{seq}(a_1, \dots a_m), \mathit{heap}_{opre}) \\
\qquad\qquad \wedge\ \{u_o\}(\mathit{inv})) \\
\qquad\qquad\quad \mathrel{\dot{\rightarrow}} \{u_o\}\{u_{opre}\}(\mathit{pre}_{noinv} \wedge \mathit{reachableIn} \\
\qquad\qquad\qquad \wedge\ \texttt{self} \not\doteq \texttt{null} \wedge \texttt{self}.\,\mathit{created} \doteq \mathit{TRUE})), \Delta \\
\Gamma \Longrightarrow \{u\}\{v\}(\mathit{wellformed}(\mathit{heap}_{opost}) \wedge \mathit{wellformed}(\mathit{heap}_{opre}) \\
\qquad \wedge\ \mathit{wellformed}(h_1) \wedge \mathit{wellformed}(h_2) \\
\qquad \wedge\ \mathit{wellformedHist}(\mathit{hist}_o) \wedge \mathit{wellformedHist}(\mathit{hist}_{opost}) \\
\qquad \wedge\ \mathit{heap}_{opre} \doteq \oplus(\texttt{heap}, \mathit{heap}_o, \{a_1, \dots, a_m\}) \\
\qquad \wedge\ \mathit{hist}_{opost} \doteq \mathit{seqConcat}(\mathit{hist}_o, \mathit{callevent}_o, \mathit{hist}_{olocal}, \mathit{termevent}_o) \\
\qquad \wedge\ \mathit{heap}_{post} \doteq \mathit{anon}(\texttt{heap}, \emptyset, h_2) \\
\qquad \wedge\ \mathit{callevent} \doteq \mathit{event}(\texttt{self}, o, m_{id}, \mathit{call}, \mathit{seq}(a_1, ..., a_m), \texttt{heap}) \\
\qquad \wedge\ \mathit{termevent} \doteq \mathit{event}(\texttt{self}, o, m_{id}, \mathit{call}, \mathit{seq}(r), \mathit{heap}_{post}) \\
\qquad \wedge\ \mathit{callevent}_o \doteq \mathit{event}(\texttt{self}, o, m_{id}, \mathit{call}, \mathit{seq}(a_1, ..., a_m), \mathit{heap}_{opre}) \\
\qquad \wedge\ \mathit{termevent}_o \doteq \mathit{event}(\texttt{self}, o, m_{id}, \mathit{call}, \mathit{seq}(r_o), \mathit{heap}_{opost}) \\
\qquad \wedge\ \mathit{transfresh}(r, \mathit{heap}_{post}, \texttt{heap}) \\
\qquad \wedge\ r \doteq \gamma(r_o) \wedge r \cong r_o \\
\qquad \wedge\ \{u_{opost}\}(\mathit{post}) \\
\qquad\quad \mathrel{\dot{\rightarrow}} \{u_{post}\}(\mathit{reachableOut} \mathrel{\dot{\rightarrow}} [\![\pi\omega]\!]\varphi)), \Delta
\end{array}}{\Gamma \Longrightarrow \{u\}[\![\pi\texttt{r = o.m}(a'_1, \dots, a'_m)\texttt{;}\omega]\!]\varphi, \Delta}$$

where:

- $w = (\texttt{self} := o \parallel \texttt{bean} := o \parallel \mathit{callingComp} := \texttt{bean} \parallel a_1 := a'_1 \parallel \dots \parallel a_m := a'_m)$
- $v = (a_1 := a'_1 \parallel \dots \parallel a_m := a'_m$
- $u_o = (\texttt{heap} := \mathit{heap}_o \parallel \texttt{hist} := \mathit{hist}_o \parallel \texttt{hist}_{\texttt{local}} := \mathit{seqEmpty})$
- $u_{opre} = (\texttt{heap} := \mathit{heap}_{opre} \parallel \texttt{hist} := \mathit{seqConcat}\,(\mathit{hist}_o, \mathit{callevent}_o) \parallel \texttt{hist}_{\texttt{local}} := \mathit{seqEmpty})$
- pre_{noinv} is the precondition of the service contract, without the invariant
- $u_{opost} = (\texttt{heap} := \mathit{heap}_{opost} \parallel \texttt{hist}_{\texttt{local}} := \mathit{hist}_{olocal} \parallel \texttt{hist} := \mathit{hist}_{opost} \parallel \texttt{heap}^{\texttt{pre}} := \mathit{heap}_{opre} \parallel \texttt{hist}^{\texttt{pre}}_{\texttt{local}} := \mathit{seqEmpty} \parallel \texttt{hist}^{\texttt{pre}} := \mathit{hist}_{opre} \parallel \texttt{res} := r_o)$

- $u_{post} = (\texttt{heap} := heap_{post} \parallel$ $\texttt{hist}_{\texttt{local}} := seqConcat(\texttt{hist}_{\texttt{local}}, callevent, termevent) \parallel$ $\texttt{hist} := seqConcat(\texttt{hist}, callevent, termevent) \parallel r := \gamma(r_o) \parallel \texttt{res} := r)$
- $\cong \in \mathcal{P}$ is a fresh predicate symbol with the semantics defined as in Definition 8.2 with respect to $heap_{opost}$ and $heap_{post}$
- $reachableIn \in Fma_{\Sigma}$ for the assignable set $mod = \{l_1, \ldots, l_n\}$ for the service with $l_i \in LocSet$ is the formula

$$\bigwedge_{i \in \{1,\ldots,n\}, \delta(l_i) \preccurlyeq Object} (l_i \doteq \texttt{null} \lor l_i.\, created \doteq TRUE)$$
$$\land \bigwedge_{i \in \{1,\ldots,n\}, \delta(l_i) = LocSet} (disjoint(l_i, unusedLocs(\texttt{heap})))$$

- $reachableOut \in Fma_{\Sigma}$ is the formula

$$(\texttt{res} \doteq \texttt{null} \lor \texttt{res}.\, created \doteq TRUE) \land \texttt{exc} \doteq \texttt{null}$$

  if $\delta(\texttt{res}) \preccurlyeq Object$ and $(\texttt{exc} \doteq \texttt{null})$ otherwise.
- $heap_o, heap_{opre}, heap_{opost}, h_1, h_2, heap_{post} : Heap \in \mathcal{F}$ are fresh function symbols
- $hist_o, hist_{olocal}, hist_{opost}, hist_{post} : Seq \in \mathcal{F}$ are fresh function symbols
- $callevent, callevent_o, termevent, termevent_o : Event \in \mathcal{F}$ are fresh function symbol
- m_{id} is the method identifier for the remote method `m`.

The rule is applicable when the first active statement of the program is a remote method call. It splits the proof into two branches as explained in detail above.

The main changes to the original method contract rule are, for one, we switch perspective, i.e. the precondition and postcondition of the method contract are evaluated from the point of view of the client bean, while the symbolic execution of the remaining program is performed from the point of view of the calling bean. We also directly encode the effect of calling remote methods into the rule, i.e. adding respective events to the histories and leaving the heap of the calling bean unchanged except creating new objects.

We only treat the case when remote methods do not throw exceptions, which is ensured in the proof obligation for service contracts. This limitation is not justified by the JavaEE specification, since exceptional behavior is explicitly allowed, however, we leave lifting this limitation for future work.

8.4 Specification and Verification of Dependency Clusters in Beans

In this section, we extend JML with primitives that allow the specification of Dependency Clusters for beans. This way, we instantiate the framework from Part I for JavaEE applications. We first introduce the syntax of the specification language for Dependency Cluster in JML and then develop step by step how the equivalence relations based on a specification written in this language are defined in JavaDL. We provide the JavaDL proof obligation resulting from a JML Dependency Cluster specification and the framework in Part I. Finally we provide the syntax for combined Dependency Cluster and the proof obligation to verify a component-global specification using a service-local Dependency Cluster.

8.4.1 Dependency Cluster Syntax in JML

In Chapter 4 we briefly introduces a specification language for Dependency Clusters for services as three lists expressing the visibility of messages, the low content of communicated values and the low part of the state. While this specification method was sufficient for presentation purpose in the examples, it is insufficient for Java programs, since it leads to imprecise specifications for object-oriented languages, as we will see in the reminder.

Figure 8.1 shows the syntax of Dependency Cluster specifications in JML. If ⟨*clusterSpec*⟩ is part of a remote method specification, it specifies a Dependency Cluster. If it is part of a class specification, it specifies the equivalence relations for which a bean is specified to be non-interferent.

A JML Dependency Cluster specification starts with the keyword `cluster`, followed by a label. The label can be used to refer to a JML Dependency Cluster in other specifications.

Then the cluster specifies five lists. The first list, following the keyword `lowIn` specifies for input events the low information. If the list contains an entry `this.m.\call(x, y)`, then the values of the parameters `x` and `y` of a call to the remote method `m` is low. Analogous, the list following `lowOut` specifies the low information contained in output events.

The keyword `lowstate` starts a list of expressions over the heap of the bean. The expressions describe the low information on the heap. The keyword `visible` starts the list specifying the visibility of events. If the list contains an entry `this.m.\term(\result > 0)`, then termination event of the remote method `m` is visible, it the return value is positive.

Finally the keyword `new_objects` starts a list of object valued expression over the heap. The list describes all low objects which are newly created on the state during the execution of the remote method. For bean-level

⟨*clusterSpec*⟩	::=’`cluster` ’ ⟨*label*⟩
	’`\lowIn`’ ⟨*lowSpecList*⟩
	’`\lowOut`’ ⟨*lowSpecList*⟩
	’`\visible`’ ⟨*visibilitySpecList*⟩
	’`\lowState`’ ⟨*expressionList*⟩
	[’`\new_objects`’ ⟨*objExprList*⟩’];’
⟨*lowSpecList*⟩	::=’`\nothing`
	| ⟨*lowSpec*⟩ (’,’ ⟨*lowSpec*⟩)*
⟨*lowSpec*⟩	::=⟨*comp*⟩’.’⟨*serv*⟩’.’⟨*cType*⟩’(’⟨*paramExprList*⟩’)’
⟨*paramExprList*⟩	::=⟨*paramExpr*⟩ (,⟨*paramExpr*⟩)*’
⟨*paramExpr*⟩	::=JML expression over parameters
⟨*expressionList*⟩	::=⟨*stateExpression*⟩ (, ⟨*stateExpression*⟩)*
⟨*stateExpression*⟩	::=JML expression without parameters
⟨*visibilitySpecList*⟩	::=⟨*visibilitySpec*⟩ (,⟨*visibilitySpec*⟩)*
⟨*visibilitySpec*⟩	::=⟨*comp*⟩’.’⟨*serv*⟩’.’⟨*cType*⟩’(’⟨*paramBoolList*⟩’)’
⟨*paramBoolList*⟩	::=⟨*paramBoolExpr*⟩ (,⟨*paramBoolExpr*⟩)*
⟨*paramBoolExpr*⟩	::=JML expression over parameters of boolean type
⟨*objExprList*⟩	::=⟨*objExpr*⟩ (, ⟨*objExpr*⟩)*
⟨*objExpr*⟩	::=JML expression without parameters of Type Object
⟨*cType*⟩	::=’`\call`’ |’`\term`’
⟨*comp*⟩	::=JML expression starting with ’`this`’ of an object type declared remote
⟨*serv*⟩	::=Method identifier for a method declared remote.

Figure 8.1: Grammar for JML Dependency Cluster

Dependency Cluster specifications, this list can be omitted. The `new_objects` list is a technicality which simplifies dependency cluster verification. Details will be presented in the reminder.

Example 8.6. As an example, we re-phrase the Dependency Clusters for the service `pay` from Example 4.3 in JML for a respective JavaEE implementation.

```
/*@ public normal_behavior
  @ cluster cluster2
  @ \lowIn this.pay.\call(ccnr),
  @        sale.registerSale.\term(0)
  @ \lowOut this.\term.pay(0),
  @         sale.registerSale.\call(ccnr)
  @ \lowState \nothing
  @ \visible this.pay.\call(true),
  @          this.pay.\term(true),
    @          sale.registerSale.\call(true),
    @          sale.registerSale.\term(true)
    @ \new_objects \nothing;
    @
    @ cluster cluster3
    @ \lowIn this.pay.\call(0),
    @        sale.registerSale.\term(0)
    @ \lowOut this.pay.\term(0),
    @         sale.registerSale.\call(prodId)
    @ \lowState product
    @ \visible this.pay.\call(true),
    @          this.pay.\term(true),
    @          sale.registerSale.\call(true),
    @          sale.registerSale.\term(true)
    @ \new_objects \nothing;
    */
 public int pay(int ccnr) {
   this.countpay++;
   sale.registerSale(this.product, this.prodprice,
                               this.prodamount, ccnr);
   return 0;
 }
```

The first Dependency Cluster, labeled cluster2, states that the parameter ccnr contains low information and that the existence termination message of sale.registerSale is low. The list lowOut states that the implementation of the remote method guarantees that the existence of the termination message only depends on low information (not the return value, however), as does the existence of a call to sale.registerSale. The Dependency Cluster does not consider any information on the state to be low, indicated by the keyword nothing. The visible list states that call and termination of the remote methods pay and sale.registerSale are visible unconditionally.

Analogous the Dependency Cluster labeled cluster3 expresses that the value of parameter prodId of sale.registerSale depends on the value of the field product. Again, all messages involved in the contract are specified visible.

8.4.2 Dependency Cluster Semantics in JavaDL

The elements in a JML Dependency Cluster specification define an equivalence relation over states and an equivalence relation over heaps, representing $\sim$ and $\approx$ from Chapter 3. In the following, we describe in detail the definition of the predicates modeling these equivalence relations in JavaDL.

We use the following abstract JML Dependency Cluster specification for illustration purposes. We frequently refer to the variables names used in this specification for definitions.

```
/*@ cluster <label>
  @ \lowIn $c_1^{in}.s_1^{in}.t_1^{in}(v_1^{in}), \dots c_m^{in}.s_m^{in}.t_m^{in}(v_m^{in})$
  @ \lowOut $c_1^{out}.s_1^{out}.t_1^{out}(v_1^{out}), \dots c_l^{out}.s_l^{out}.t_l^{out}(v_l^{out})$
  @ \visible $c_1^{vis}.s_1^{vis}.t_1^{vis}(v_1^{vis}), \dots, c_j^{vis}.s_j^{vis}.t_j^{vis}(v_j^{vis})$
  @ \lowState $R$
  @ \new_objects $N$;
  @*/
```

Definition 8.8 (JavaDL Dependency Cluster)**.** A JavaDL Dependency Cluster *dc* is a tuple (*label*, *ins*, *outs*, *visibles*, *state*, *new*) where

- *label* is a label unique among all JavaDL Dependency Cluster specifications in a Java class,
- *ins* is a set of tuples $\{(c_1^{in}, s_1^{in}, v_1^{in}), \dots, (c_m^{in}, s_m^{in}, v_m^{in})\}$,
- *outs* is a set of tuples $\{(c_1^{out}, s_1^{out}, v_1^{out}), \dots, (c_l^{out}, s_l^{out}, v_l^{out})\}$,
- *visibles* is a set of tuples $\{(c_1^{vis}, s_1^{vis}, v_1^{vis}), \dots, (c_j^{vis}, s_j^{vis}, v_j^{vis})\}$,
- c_i^{in}, c_i^{out}, c_i^{vis} are terms of type *Object*,
- s_i^{in}, s_i^{out}, and s_i^{vis} are method identifier of type *Method*,
- t_i^{in}, t_i^{out}, and $t_i^{vis} \in \{call, term\}$,
- v_i^{in} and v_i^{out} are observation expressions over parameters and return values of remote methods,
- v_i^{vis} are JavaDL terms of type *bool* over parameters and return values of remote methods,
- *state* is an observation expression over the bean's heap (see R in the example above)
- *new* is an observation expression of objects over the bean's heap (see N in the example above)

A JavaDL Dependency Cluster defines two equivalence relations: one for event equivalence (analogous to $\sim$ in Chapter 3) and one for heap equivalence (analogous to $\approx$ in Chapter 3). Instead of introducing an explicit invisible representative (see $\square$ in Chapter 3), we use an additional predicate expressing whether an event is visible or not. In the following, we define the predicates *invEvent*, *equivEvent*, and combine them to gain the equivalence relation for histories *equivHist* for a Dependency Cluster. We further define two equivalence relations for heaps, $agree_{pre}$ and $agree_{post}$, for heaps in the prestate and in the poststate of a remote method and explain, why we use two different equivalence relations here.

For the following definitions, we assume a JavaDL Dependency Cluster to be given and refer to the names of the elements of the tuples with the names as used in Definition 8.8.

Event Visibility A call event for a remote method m of bean b is specified to be visible, if the list `visible` contains an entry $b.m.call(vis)$ for the respective remote method, and vis evaluates to true given the communicated values in the event. Analogous, visibility of termination events is specified.

Formally, the predicate *invEvent* is defined as follows.

Definition 8.9.

$$invEvent(e) :\Leftrightarrow$$
$$\bigwedge_{x=1}^{j} (o \doteq c_x^{vis} \wedge s \doteq s_x^{vis} \wedge t \doteq t_x^{vis}) \dot{\rightarrow} \{u_x^{vis}\} v_x^{vis} \doteq FALSE$$

where

- $s = evMethod(e)$, $t = evCalltype(e)$, $h = evHeap(e)$, $o = evClient(e)$
- $p_x^i = \begin{cases} \texttt{a} & \text{if } t_x^{vis} = call \text{ and } \texttt{a} \text{ is the program variable used} \\ & \text{as the i-th parameter of } s_x^{vis} \\ \texttt{res} & \text{if } t_x^{vis} = term \end{cases}$
- $param_x^i = seqGet\,(evParams(e), i)$
- $u_x^{vis} = p_x^1 := param_x^1, \parallel \ldots \parallel p_x^p := param_x^p \parallel \texttt{heap} := h$

Each of the tuples (indexed 1 to j) in *visibles* specifies a boolean expression (v_x^{vis}) which states that an event with the respective receiving object c_x^{vis}, method identifier s_x^{vis} and communication direction t_x^{vis} is visible, if v_x^{vis} evaluates to *TRUE* in the heap stored in the event.

Event Equivalence To define event equivalence, we define several predicates for channel equality, equivalence of types of communicated low information and primitive values, and for equivalence of objects-valued low information. We combine the predicates in order to gain a definition of equivalence of events (analogue to state equivalence $\sim$ in Chapter 3).

In Chapter 3 we formulated the side condition for equivalence relations over messages, that if two messages are visible and equivalent, then they have to be communicated over the same channel. In Section 2.4 we also stated that a channel must at most be shared between two components, one requiring the respective service, the other one providing it. We further stated, that this does not state a major problem, since a service can be made available for more than one component by simply renaming the initial and terminating channel and copying the body of the service implementation. Actually copying the code of the Java implementation of a bean, however would be very impractical.

Thus, we model the channel as a combination of the calling bean, the client bean, the method identifier and whether the event is a call or termination event. This way, we do not require an explicit renaming of channels, but different beans calling the same remote method provided by one bean use different channels by definition. The predicate *equalMetadata* expresses whether two event are communicated over the same channel.

Definition 8.10 (Channel Equality)**.**

$$\begin{aligned} &\mathit{equalMetadata}(e_A, e_B) :\Leftrightarrow \\ &\qquad \mathit{evCaller}(e_A) \doteq \mathit{evCaller}(e_B) \wedge \mathit{evClient}(e_A) \doteq \mathit{evClient}(e_A) \\ &\quad \wedge \quad \mathit{evCalltype}(e_A) \doteq \mathit{evCalltype}(e_B) \wedge \mathit{evMethod}(e_A) \doteq \mathit{evMethod}(e_B) \end{aligned}$$

Apart from the same channels, equivalent events have to communicate equivalent information. In specification $b.m.call(e_1, e_2)$, the list of low information contained in an event for the call of remote method m of bean b is given by the expressions e_1 and e_2. The easiest and straight forward way to define information equivalence would be to require the observation expressions to evaluate to the same sequence of values, as done in the approach in Chapter 4. For the example, this would mean, we require e_1 and e_2 to evaluate to the same values, given the values communicated by the respective events. However this would mean that whenever an object is communicated as a low information, the expression evaluating to this object would have to evaluate to the exactly same object in both events.

Example 8.7. See for example the following method with a partial specification.

```
/*@ cluster someCluster
  @ ...
  @ \lowOut (otherbean.m.call(a), ...)
  @ \visible (otherbean.m.call(true) ,...)
  @ ...
  @*/
  public void doSth() {
    otherbean.m(new Object());
  }
```

When `a` is the name of the only parameter of `m`, evaluating the observation expression containing `a` for two different runs of `doSth()` would mean that the two call events of `otherbean.m` would only be equivalent, if in both runs the same object would be created. Since the order of object creation in JavaDL is underspecified, this can not be guaranteed. The events therefore would not be equivalent in general, and the remote method would not satisfy its specification. Labeling `doSth` as insecure, however, would be an over-approximation.

We require a less strict condition for equivalence of parameters which takes into consideration that object identities are opaque. The approach we present in the following is inspired by object-sensitive information flow for sequential Java programs as discussed by Beckert et al..

As a first condition, we require the observation expressions for both events to evaluate to sequences of the same length and that values on the same position in the two sequence are of the same dynamic type. If the entry is neither of type *Object* nor *Seq*, i.e. a primitive value is communicated, the entries actually have to be equal. If the entry is of type *Seq*, we recursively require the respective sequences to contain elements of the same type themselves. Remember that sequences are an abstract datatype, i.e. they only exist in specifications. They are not Java datatypes, which are actually communicated. The predicate $agree_T$ formalizes type equivalence of two sequences.

Definition 8.11.

$$
\begin{aligned}
&agree_T(S_A, S_B) :\Leftrightarrow\\
&length(S_A) \doteq length(S_B) \land\\
&\forall 0 \leq i < length(S_A); (\delta(seqGet(S_A, i) \doteq \delta(seqGet(S_B, i)))\\
&\qquad \land ((\dot{\neg}(\delta(seqGet(S_A, i)) \preccurlyeq Object) \land \dot{\neg}(\delta(seqGet(S_A, i)) \preccurlyeq Seq))\\
&\qquad\qquad\qquad\qquad\qquad\qquad \dot{\rightarrow} seqGet(S_A, i) \doteq seqGet(S_B, i))\\
&\qquad \land (\delta(seqGet(S_A, i)) \preccurlyeq Seq \dot{\rightarrow} agree_T(seqGet(S_A, i), seqGet(S_B, i))))
\end{aligned}
$$

The predicate $agree_T$ ignores values of entries in the sequence of type *Object*.

Beckert et al. argue that if the object identities are opaque, it is sufficient to ensure that there exists an isomorphism between object identities in two observation expressions. To be precise, Beckert et al. also argue that this is only true for newly created objects, because otherwise, it might be possible for an attacker to check if he had already seen the object in a previously observable state. However, due to serialization and deserialization of remote method calls, the entire communicated object structure is newly created, be it with new identifier during serialization or object creation during deserialization.

The predicate $\mathit{agree}_{\mathit{Obj}}$ evaluates to true for two pairs of sequences T_A, S_A and TB, S_B, if for every entry in T_A and every entry in S_A, the lists contain the same object, iff the same entries in T_B and S_B also refer to the same object. If $\mathit{agree}_{\mathit{Obj}}(T_A, T_A, T_B, T_B)$ evaluates to true, then there exists an isomorphism for T_A and T_B

The predicate $\mathit{agree}_{\mathit{Obj}}$ evaluates to true for two observation expressions T_A and T_B if for each object-valued element in T_A that it is equal to an entry in S_A that the corresponding entry in T_B is equal to the same entry of S_B. If $\mathit{agree}_{\mathit{Obj}}(T_A, T_A, T_B, T_B)$ is satisfied, then there exists a partial isomorphism for object identities in T_A and T_B.

We formally define $\mathit{agree}_{\mathit{Obj}}$:

Definition 8.12.

$$
\begin{aligned}
&\mathit{agree}_{\mathit{Obj}}(S_A, T_A, S_B, T_B) :\Leftrightarrow\\
&\forall 0 \leq i < \mathit{length}(T_A); (\\
&(\mathit{seqGet}\,(T_A, i) \preccurlyeq \mathit{Object}\\
&\qquad \dot{\rightarrow}((\mathit{seqGet}\,(T_A, i) \doteq \mathtt{null} \land \mathit{seqGet}\,(T_B, i) \doteq \mathtt{null})\\
&\qquad\quad \lor \mathit{agree}^2_{\mathit{Obj}}(S_A, \mathit{seqGet}\,(T_A, i), S_B, \mathit{seqGet}\,(T_B, i))))\\
&\land (\mathit{seqGet}\,(T_A, i) \preccurlyeq \mathit{Seq} \,\dot{\rightarrow}\, \mathit{agree}_{\mathit{Obj}}(S_A, \mathit{seqGet}\,(T_A, i), S_B, \mathit{seqGet}\,(T_B, i))))
\end{aligned}
$$

where

$$
\begin{aligned}
&\mathit{agree}^2_{\mathit{Obj}}(S_A, O_A, S_B, O_B) :\Leftrightarrow\\
&\forall 0 \leq i < \mathit{length}(S_A); ((\mathit{seqGet}\,(S_A, i) \preccurlyeq \mathit{Object}\\
&\qquad \dot{\rightarrow}(\mathit{seqGet}\,(S_A, i) \doteq O_A \,\dot{\leftrightarrow}\, \mathit{seqGet}\,(S_B, i) \doteq O_B))\\
&\quad \land (\mathit{seqGet}\,(S_A, i) \preccurlyeq \mathit{Seq}\\
&\qquad \dot{\rightarrow}(\mathit{agree}^2_{\mathit{Obj}}(\mathit{seqGet}\,(S_A, i), O_A, \mathit{seqGet}\,(S_B, i), O_B))))
\end{aligned}
$$

The definition of $\mathit{agree}_{\mathit{Obj}}$ walks through all entries of T_A and T_B. If the respective entry is of type object, it uses the predicate $\mathit{agree}^2_{\mathit{Obj}}$, which checks if all entries in a list S_A are equal to an object O_A, iff the same entry in S_B is equal to an object O_B. If an entry of T_A is a sequence, it recursively calls itself.

We can now combine agree_T and $\mathit{agree}_{\mathit{Obj}}$ to define information equivalence for events with predicate *equivInfo*. Given a JML dependency Cluster specification, *equivInfo* first selects the relevant entry in the `lowIn` and `lowOut` lists by checking if the meta data of an event matches the respective information of an entry in the specification. Then the predicate evaluates to true, if agree_T and $\mathit{agree}_{\mathit{Obj}}$ evaluate to true, if the lists over parameters in the specification evaluate to true, given the communicated values in the event.

We formally define information equivalence as follows:

Definition 8.13 (Equivalent Information)**.**

$$
\begin{aligned}
&\mathit{equivInfo}(e_A, e_B) :\Leftrightarrow \\
&\quad \bigwedge_{x=1}^{m} (or_A \doteq c_x^{in} \land m_A \doteq s_x^{in} \land t_A \doteq t_x^{in}) \\
&\qquad \dot{\rightarrow}\ \mathit{agree}_T(V_{A,x}^{in}, V_{B,x}^{in}) \land \mathit{agree}_{Obj}(V_{A,x}^{in}, V_{A,x}^{in}, V_{B,x}^{in}, V_{B,x}^{in}) \\
&\quad \land \bigwedge_{x=1}^{l} (or_A \doteq c_x^{out} \land m_A \doteq s_x^{out} \land t_A \doteq t_x^{out}) \\
&\qquad \dot{\rightarrow}\ \mathit{agree}_T(V_{A,x}^{out}, V_{B,x}^{out}) \land \mathit{agree}_{Obj}(V_{A,x}^{out}, V_{A,x}^{out}, V_{B,x}^{out}, V_{B,x}^{out})
\end{aligned}
$$

where

- $or_A = \mathit{evClient}(e_A) \land m_A = \mathit{evMethod}(e_A) \land t_A = \mathit{evCalltype}(e_A)$
- $V_{A,x}^{in} = \{p_x^{in,1} := \mathit{param}_A^1 \parallel \ldots \parallel p_x^{in,k} := \mathit{param}_A^k \parallel \texttt{heap} := \mathit{evHeap}(e_A)\}(v_x^{in})$
- $V_{B,x}^{in} = \{p_x^{in,1} := \mathit{param}_B^1 \parallel \ldots \parallel p_x^{in,k} := \mathit{param}_B^k \parallel \texttt{heap} := \mathit{evHeap}(e_B)\}(v_x^{in})$
- $V_{A,x}^{out} = \{p_x^{out,1} := \mathit{param}_A^1 \parallel \ldots \parallel p_x^{out,k} := \mathit{param}_A^k \parallel \texttt{heap} := \mathit{evHeap}(e_A)\}(v_x^{out})$
- $V_{B,x}^{out} = \{p_x^{out,1} := \mathit{param}_B^1 \parallel \ldots \parallel p_x^{out,k} := \mathit{param}_B^k \parallel \texttt{heap} := \mathit{evHeap}(e_B)\}(v_x^{out})$
- $p_x^{in,i} = \begin{cases} a & \text{if } t_x^{in} = \mathit{call} \text{ and } a \text{ is the program variable used} \\ & \text{as the i-th parameter of } s_x^{in} \\ \texttt{res} & \text{if } t_x^{in} = \mathit{term} \end{cases}$
- $p_x^{out,i} = \begin{cases} a & \text{if } t_x^{out} = \mathit{call} \text{ and } a \text{ is the program variable used} \\ & \text{as the i-th parameter of } s_x^{out} \\ \texttt{res} & \text{if } t_x^{out} = \mathit{term} \end{cases}$
- $\mathit{param}_C^i = \mathit{seqGet}\left(\mathit{evParams}(e_C), i\right)$ for $C \in \{A, B\}$

The predicate *equivInfo* extracts the relevant observation expression from the specification depending on the meta data (i.e. channel) in the event e_A and evaluates the observation expression with the parameters and heap provided by the events. Note that we only evaluate the meta data of event e_A in Definition 8.13. Since in the following predicate *equivEvent* the events are also checked for equal metadata, i.e. equal channels, only checking the meta data of e_A is sufficient.

The predicate *equivEvent* defines the equivalence relation for events (see $\sim$ in Chapter 3) by combining the predicates *invEvent*, *equalMetadata*, and *equivInfo*.

Definition 8.14 (Event Equivalence)**.**

$$\mathit{equivEvent}(e_A, e_B) :\Leftrightarrow \begin{cases} (\mathit{invEvent}(e_A) \wedge \mathit{invEvent}(e_B)) \vee \\ (\dot{\neg}\, \mathit{invEvent}(e_A) \wedge \dot{\neg}\, \mathit{invEvent}(e_B) \\ \qquad \wedge\, \mathit{equalMetadata}(e_A, e_B) \wedge \mathit{equivInfo}(e_A, e_B) \end{cases}$$

History Equivalence Equality for events gives a rise for equivalence of histories, similar to the definition in Chapter 3. Two histories are equivalent, if the contained events are equivalent, while invisible events are ignored.

The formal definition is straight forward.

Definition 8.15 (History Equivalence)**.**

$$\begin{aligned}&\mathit{equivHist}(h_A, h_B)\\ &:\Leftrightarrow \begin{cases} \mathit{TRUE} & \text{if } h_A \doteq \mathit{seqEmpty} \wedge h_B \doteq \mathit{seqEmpty} \\ \mathit{equivHist}(h'_A, h_B) & \text{if } h_A = \mathit{seqConcat}\,(e_A, h'_A) \wedge \mathit{invEvent}(e_A) \\ \mathit{equivHist}(h_A, h'_B) & \text{if } h_B = \mathit{seqConcat}\,(e_B, h'_B) \wedge \mathit{invEvent}(e_B) \\ \mathit{equivHist}(h'_A, h'_B) & \text{if } h_A = \mathit{seqConcat}\,(e_A, h'_A) \wedge \mathit{seqConcat}\,(e_B, h'_B) \\ & \wedge\, \mathit{equivEvent}(e_A, e_B) \end{cases}\end{aligned}$$

Heap Equivalence To define the equivalence relation over states (i.e. $\approx$ in Chapter 3), we directly employ the approach as discussed by Beckert et al.: an object sensitive notion for equivalence of heaps based on two observation expressions. The first observation expression is provided in a JavaDL Dependency Cluster as the observation expression R following the `lowState` keyword. The second list N, following the keyword `new_objects`, the low objects newly created during execution of a service are listed.

Directly comparing the evaluation of the observation expression R for equality, as done in Chapter 4, would, again, lead to over-approximation as already discussed for information equivalence in events.

Example 8.8. In the example below, the content of the field `f` is specified to contain low information. After execution of the remote method, the field points to a new object, independent from any high information. When comparing two runs, however, it can not be guaranteed that in both runs, the same object was newly created. Therefore, requiring equality of the expression `this.f` in both poststates would consider `doSthElse` not to satisfy its contract.

```
/*@ cluster someCluster
  @ ...
  @ \lowState (this.f ,...)
  @ ...
  @*/
  public void doSthElse() {
    this.f = new Object();
  }
```

We apply the solution by Beckert et al. for object-sensitive information flow with minor changes. Beckert et al. differentiate between equivalence in the prestate and the poststate of a program execution. In both prestates the observation expression is assumed to evaluate to equal values. The formula $\mathit{agree}_{\mathit{pre}}$ defines for a given observation expression R the equivalence relation $\approx$ for prestates.

Definition 8.16 (Prestate Equivalence)**.**

$$\mathit{agree}_{\mathit{pre}}(h_a, h_b) :\Leftrightarrow \{\mathtt{heap} := h_a\}R \doteq \{\mathtt{heap} := h_b\}R$$

In order to gain a precise notion of equivalence in the case of object creation, a different notion of equivalence is defined for the poststate. As for event equivalence defined above, an isomorphism of newly created objects in R is required. To make verification easier, the newly created objects are explicitly given in the JavaDL Dependency Cluster as observation expression N.

The predicate $\mathit{agree}_{\mathit{post}}$ evaluates to true for an observation expression and two pre heaps h_a, h_b and two post heaps $h'_a, h'b'$, if all not-newly created objects in h'_a and h'_b contained in R are equal, and there exits an isomorphism for the newly created objects in R evaluated in the two post heaps.

The formal definition for $\mathit{agree}_{\mathit{post}}$ is as follows:

Definition 8.17 (Poststate Equivalence)**.**

$$\begin{aligned}
&\mathit{agree}_{\mathit{post}}(h_a, h'_a, h_b, h'_b) :\Leftrightarrow\\
&\quad \mathit{newIso}(h_a, h'_a, h'_b, h_b)\\
&\quad \wedge (\{\mathtt{heap} := h'_a\}N \doteq \{\mathtt{heap} := h'_b\}N \dot{\rightarrow} \{\mathtt{heap} := h'_a\}R \doteq \{\mathtt{heap} := h'_b\}R)
\end{aligned}$$

where

$$\begin{aligned}
&\mathit{newIso}(h_a, h'_a, h_b, h'_b) :\Leftrightarrow\\
&\quad \mathit{allNew}(\{\mathtt{heap} := h'_a\}N, h_a) \wedge \mathit{allNew}(\{\mathtt{heap} := h'_b\}N, h_b)\\
&\quad \wedge \mathit{agree}_T(\{\mathtt{heap} := h'_a\}N, \{\mathtt{heap} := h'_b\}N)\\
&\quad \wedge \mathit{agree}_{\mathit{Obj}}(\{\mathtt{heap} := h'_a\}N, \{\mathtt{heap} := h'_a\}N, \{\mathtt{heap} := h'_b\}N,\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \{\mathtt{heap} := h'_b\}N)
\end{aligned}$$

and

$$\begin{aligned}
&\mathit{allnew}(S, h) :\Leftrightarrow\\
&\quad \forall 0 \leq i < \mathit{length}(S); (\\
&\qquad (\delta(\mathit{seqGet}\,(S,i)) \preccurlyeq \mathit{Object} \wedge (\mathit{seqGet}\,(S,i) \dot{\neq} \mathtt{null})\\
&\qquad\qquad \dot{\rightarrow} \mathit{select}_{\mathit{Object}}(h, \mathit{seqGet}\,(S,i), \mathit{created}) \doteq \mathit{FALSE})\\
&\qquad \wedge (\delta(\mathit{seqGet}\,(S,i)) \preccurlyeq \mathit{Seq} \dot{\rightarrow} \mathit{allnew}(\mathit{seqGet}\,(S,i), h)))
\end{aligned}$$

The heaps h_a and h_b represent the heaps before program execution and h'_a and h'_b represent the heaps after execution. Note that in contrast to *equivInfo* only the the newly created objects, i.e. observation expression N, are checked for an existing isomorphism (predicate *newIso*). The reason is that in contrast to events, potentially only some of the objects in R are newly created during execution of a remote method.

The *allnew* predicate ensures that all objects described by the observation expression N actually are newly created during execution.

Beckert et al. present a compositionality proof which states that if for two heaps in the poststate and their respective heaps in the prestate $agree_{post}$ holds, it is sound to assume $agree_{pre}(h_a, h_b)$ when the heaps in the poststate become heaps in the prestate for the next execution of a remote method. The idea behind this assumption is that if an isomorphism for the newly created objects exist, it can be assumed that in both runs of the remote method, the same object identities were selected for creation.

8.4.3 Cooperative Environments

In Chapter 3 we made the assumption that the environment in which a component is executed is cooperative, i.e. for every service called with two equivalent call messages, the environment returns two equivalent termination events. The assumption of a cooperative environment is a restriction on the history of a bean.

We directly provide a definition for cooperative environments with the predicate *coopHistEquiv* for two histories:

Definition 8.18 (JavaDL Cooperative Environments)**.**

$$
\begin{aligned}
&\mathit{coopHistEquiv}(h_A, h_B)\\
&:\Leftrightarrow \begin{cases}
\mathit{TRUE} & \text{if } h_A \doteq \mathit{seqEmpty} \wedge h_B \doteq \mathit{seqEmpty}\\
\begin{array}{l}\mathit{equivEvent}(e'_A, e'_B)\\ \wedge\, \mathit{coopHistEquiv}(h'_a, h'_B)\end{array} & \text{if } \begin{cases} h_A \doteq \mathit{seqConcat}(e_A, e'_A, h'_A)\\ \wedge h_B \doteq \mathit{seqConcat}(e_B, e'_B, h'_B)\\ \wedge\, \mathit{equivEvent}(e_A, e_B)\\ \wedge\, \mathit{evCalltype}(e_A) \doteq \mathit{call}\\ \wedge\, \mathit{evCalltype}(e_B) \doteq \mathit{call}\\ \wedge\, \mathit{evCalltype}(e'_A) \doteq \mathit{term}\\ \wedge\, \mathit{evCalltype}(e'_B) \doteq \mathit{term} \end{cases}
\end{cases}
\end{aligned}
$$

The predicate *coopHistEquiv* states for two histories that they are cooperatively equivalent, iff, given two equivalent call events followed by a termination event each, the termination events again are equivalent.

Cooperative environments also ensure that an (in-)visible call event is always followed by an (in-)visible termination event respectively. The predicate *coopHist* formalizes this behavior as a property for a history in JavaDL.

Definition 8.19 (Visibility Preserving Histories)**.**

$$
\begin{aligned}
&\mathit{coopHist}(h)\\
&:\Leftrightarrow \begin{cases} \mathit{TRUE} & \text{if } h \doteq \mathit{seqEmpty} \\ \mathit{invEvent}(e) \leftrightarrow \mathit{invEvent}(e') \\ \wedge\, \mathit{coopHist}(h') & \text{if } \begin{cases} h \doteq \mathit{seqConcat}(e, e', h') \\ \wedge\, \mathit{evCalltype}(e) \doteq \mathit{call} \\ \wedge\, \mathit{evCalltype}(e') \doteq \mathit{term} \end{cases} \end{cases}
\end{aligned}
$$

8.4.4 Verifying Dependency Cluster

We have now defined the JavaDL formalization for event equivalence ($\sim$), state equivalence ($\approx$) and cooperative environments. This allows us to provide a JavaDL formalization of the non-interference criterion as in Definition 3.12 as a JavaDL proof obligation.

The formula is a direct encoding of Definition 3.12 in JavaDL. We compare two executions of a remote method. We assume that both executions represent a call to the same bean ($\mathtt{self}_a \doteq \mathtt{self}_B$) and are performed by the same caller ($\mathit{callingComp}_a \doteq \mathit{callingComp}_B$). We further assume that the fields b_i of the bean, which reference other remote beans, actually reference the same remote beans $\mathit{select}_{\mathit{Object}}(\mathtt{heap}_A, \mathtt{self}_A, b_i) \doteq \mathit{select}_{\mathit{Object}}(\mathtt{heap}_B, \mathtt{self}_B, b_i)$.

We assume the both executions are started in equivalent prestates ($\mathit{agree}_{\mathit{pre}}(\mathtt{heap}_A, \mathtt{heap}_B)$), and called with equivalent call events:

$$\mathit{equivEvent}(\mathit{callevent}_A, \mathit{callevent}_B)$$

Further, we assume cooperative environments:

$$
\begin{aligned}
&\mathit{coopHist}(\mathtt{hist}_{\mathtt{local}A}), \mathit{coopHist}(\mathtt{hist}_{\mathtt{local}B}), \text{ and}\\
&\mathit{coopHistEquiv}(\mathtt{hist}_{\mathtt{local}A}, \mathtt{hist}_{\mathtt{local}B}).
\end{aligned}
$$

We then have to check after the two executions, that they terminate in equivalent poststates and that the events communicated during execution are equivalent.

$$
\begin{aligned}
&\mathit{agree}_{\mathit{post}}(\mathtt{heap}_A, \mathit{heapAtPost}_A, \mathtt{heap}_B, \mathit{heapAtPost}_B) \text{ and}\\
&\mathit{equivHist}(\mathtt{hist}_{\mathtt{local}A}, \mathtt{hist}_{\mathtt{local}B})
\end{aligned}
$$

We have to check that both executions terminate with equivalent termination events ($\mathit{equivEvent}(\mathit{termevent}_A, \mathit{termevent}_B)$).

Finally, we also have to verify that the remote method is visibility-preserving. It is sufficient to check this for one execution. So, we additionally require the remote method to terminate with a visible event, if the it was called with a visible event:

$$\dot\neg(\mathit{invEvent}(\mathit{callevent}_A)) \mathrel{\dot\rightarrow} \dot\neg(\mathit{invEvent}(\mathit{termevent}_A))$$

If the service was called with an invisible event, it has to terminate with an invisible event, the events sent during execution have to be invisible, and the prestate has to be equivalent to the poststate:

$$\begin{aligned}&\mathit{invEvent}(\mathit{callevent}_A)\\&\quad\dot{\rightarrow}(\mathit{invEvent}(\mathit{termevent}_A) \wedge \mathit{equivHist}(\mathtt{hist}_{\mathtt{local}A}, \mathit{seqEmpty})\\&\qquad\wedge\ \mathit{agree}_{\mathit{pre}}(\mathtt{heap}_A, \mathit{heapAtPost}_A))\end{aligned}$$

The complete definition of the proof obligation is presented in the following definition. It additionally contains some technicalities (especially Lines 8.19 to 8.25), which we do not discussed here:

Definition 8.20 (Remote Method Non-interference)**.** A method m declared remote is non-interferent w.r.t. to a JavaDL Dependency Cluster dc, defining the predicates *invEvent*, *equivEvent*, $\mathit{agree}_{\mathit{pre}}$, and $\mathit{agree}_{\mathit{post}}$ if the following formula is valid.

$$\mathit{wellformed}(\mathtt{heap}_A) \land \mathit{wellformedHist}(\mathtt{hist}_A) \tag{8.19}$$

$$\land\, \mathit{wellformed}(\mathtt{heap}_B) \land \mathit{wellformedHist}(\mathtt{hist}_B) \tag{8.20}$$

$$\land\, \mathtt{self}_A \mathrel{\dot{\neq}} \mathtt{null} \land \mathtt{self}_B \mathrel{\dot{\neq}} \mathtt{null} \tag{8.21}$$

$$\land\, \mathit{callingComp}_A \mathrel{\dot{\neq}} \mathtt{null} \land \mathit{callingComp}_B \mathrel{\dot{\neq}} \mathtt{null} \tag{8.22}$$

$$\land\, \mathit{callingComp}_A \mathrel{\dot{\neq}} \mathtt{bean}_A \land \mathit{callingComp}_B \mathrel{\dot{\neq}} \mathtt{bean}_B \tag{8.23}$$

$$\land\, \mathtt{self}_A \doteq \mathtt{bean}_A \land \mathtt{self}_B \doteq \mathtt{bean}_B \tag{8.24}$$

$$\land\, \mathtt{bean}_A \doteq \mathtt{self}_A \land \mathtt{bean}_B \doteq \mathtt{self}_B \land \mathit{call}_A \land \mathit{call}_B \tag{8.25}$$

$$\land\, \{\mathtt{heap} := \mathtt{heap}_A \parallel \mathtt{self} := \mathtt{self}_A \parallel \mathtt{hist} := \mathtt{hist}_A \tag{8.26}$$

$$\parallel \mathit{callingComp} := \mathit{callingComp}_A \parallel \mathtt{hist}_{\mathtt{local}} := \mathit{seqEmpty} \tag{8.27}$$

$$\parallel a'_1 := a_{1A} \parallel \ldots \parallel a'_m := a_{mA}\} \tag{8.28}$$

$$\{\mathtt{hist} := \mathit{seqConcat}\,(\mathtt{hist}, \mathit{callevent}_A)\} \tag{8.29}$$

$$\langle \mathtt{r\ =\ this.m(}a'_1, \ldots, a_m\mathtt{)} \rangle \tag{8.30}$$

$$(\mathit{termevent}_A \doteq \mathit{event}(\mathit{callingComp}, \mathtt{self}, m_{id}, \mathit{term}, \mathit{seq}(\mathtt{res}), \mathtt{heap}) \tag{8.31}$$

$$\land\, \mathit{histAtPost}_A \doteq \mathit{seqConcat}\,(\mathtt{hist}, \mathit{termevent}_A) \tag{8.32}$$

$$\land\, \mathit{resultAtPost}_A \doteq \mathtt{res} \land \mathit{heapAtPost}_A \doteq \mathtt{heap} \tag{8.33}$$

$$\land\, \mathtt{hist}_{\mathtt{local}A} \doteq \mathtt{hist}_{\mathtt{local}}) \tag{8.34}$$

$$\land\, \mathit{execution}_B \tag{8.35}$$

$$\dot{\rightarrow} \tag{8.36}$$

$$((\mathtt{self}_A \doteq \mathtt{self}_B \land \mathit{callingComp}_A \doteq \mathit{callingComp}_B \tag{8.37}$$

$$\land\, \mathit{select}_{\mathit{Object}}(\mathtt{heap}_A, \mathtt{self}_A, b_1) \doteq \mathit{select}_{\mathit{Object}}(\mathtt{heap}_B, \mathtt{self}_B, b_1) \tag{8.38}$$

$$\land \ldots \tag{8.39}$$

$$\land\, \mathit{select}_{\mathit{Object}}(\mathtt{heap}_A, \mathtt{self}_A, b_k) \doteq \mathit{select}_{\mathit{Object}}(\mathtt{heap}_B, \mathtt{self}_B, b_k) \tag{8.40}$$

$$\land\, \mathit{agree}_{\mathit{pre}}(\mathtt{heap}_A, \mathtt{heap}_B) \land \mathit{equivEvent}(\mathit{callevent}_A, \mathit{callevent}_B) \tag{8.41}$$

$$\land\, \mathit{invEvent}(\mathit{callevent}_A) \mathrel{\dot{\leftrightarrow}} \mathit{invEvent}(\mathit{callevent}_B) \tag{8.42}$$

$$\land\, \mathit{coopHist}(\mathtt{hist}_{\mathtt{local}A}) \land \mathit{coopHist}(\mathtt{hist}_{\mathtt{local}B}) \tag{8.43}$$

$$\land\, \mathit{coopHistEquiv}(\mathtt{hist}_{\mathtt{local}A}, \mathtt{hist}_{\mathtt{local}B})) \tag{8.44}$$

$$\dot{\rightarrow} \tag{8.45}$$

$$(\mathit{agree}_{\mathit{post}}(\mathtt{heap}_A, \mathit{heapAtPost}_A, \mathtt{heap}_B, \mathit{heapAtPost}_B) \tag{8.46}$$

$$\land\, \mathit{equivHist}(\mathtt{hist}_{\mathtt{local}A}, \mathtt{hist}_{\mathtt{local}B}) \tag{8.47}$$

$$\land\, \mathit{equivEvent}(\mathit{termevent}_A, \mathit{termevent}_B) \tag{8.48}$$

$$\land\, (\dot{\neg}(\mathit{invEvent}(\mathit{callevent}_A)) \mathrel{\dot{\rightarrow}} \dot{\neg}(\mathit{invEvent}(\mathit{termevent}_A))) \tag{8.49}$$

$$\land\, (\mathit{invEvent}(\mathit{callevent}_A) \tag{8.50}$$

$$\dot{\rightarrow}(\mathit{invEvent}(\mathit{termevent}_A) \land \mathit{equivHist}(\mathtt{hist}_{\mathtt{local}A}, \mathit{seqEmpty}) \tag{8.51}$$

$$\land\, \mathit{agree}_{\mathit{pre}}(\mathtt{heap}_A, \mathit{heapAtPost}_A))))) \tag{8.52}$$

where

- for $C \in \{A, B\}$:
  $call_C = (callevent_C \doteq event(callingComp_C, \mathtt{self}_C, m_{id}, call, seqConcat(a_{1C}, ..., a_{mC}), \mathtt{heap}_C))$
- m_{id} is the identifier for method m
- $execution_B$ is the same formula as the formula from 8.26 to 8.34, indexed with B instead of A
- $b_1, \ldots, b_k$ are the fields of the bean which themselves are declared to hold beans.
- *equivHist*, *invEvent*, *equivEvent*, $agree_{pre}$, and $agree_{post}$ are the predicate as defined above according to JavaDL Dependency Cluster *dc*.

8.4.5 Combined Dependency Cluster

Dependency Cluster are meant to be building blocks for complex non-interference specifications. In order to build complex non-interference specifications, we introduced combined Dependency Cluster in Chapter 4. We now provide a JML specification scheme to define combined Dependency Cluster in JML. The following listing shows the syntax of a specification combining non-interference specifications.

```
/*@ cluster label \combines labels; */
```

where *labels* is a list of labels used for Dependency Cluster specifications.

Example 8.9. In the following listing, we declare the JML Dependency Cluster `clusterBuy1` for the remote method `buy` of our running example. The JML Dependency Cluster expresses the dependency of the value of the field `product` on the value of the parameter `prod` of the remote method.

We also define the bean-global Dependency Cluster `prodLowCluster` as a combination of `clusterBuy1` cluster and `cluster3` as declared in Example 8.6.

```
public class Cart implements CartIF {
  ...
  /*@ cluster prodLowCluster
    @    \combines clusterBuy1, cluster3; */
  ...
    /*@ public normal_behavior
    @ cluster clusterBuy1
    @ \lowIn this.buy.\call(prod)
    @ \lowOut this.buy.\term(0)
    @ \lowState this.product
    @ \visible this.buy.\call(true),
    @          this.buy.\term(true)
    @ \new_objects \nothing;
    @*/
  public int buy(int prod, int price, int amount) {...}
}
```

Similar to Dependency Cluster specifications, combined Dependency Cluster specifications in JML can be applied on class level and on remote method level. If it is applied on class level, any label can be used in the list, which identifies either a bean-level or a remote method-level JML Dependency Cluster in the same class. If the specification is applied on method level, only those labels are allowed which refer to a specification for the same remote method.

The semantics of a combined non-interference specification is given as the intersection of the equivalence relations, which are defined by the predicates *invEvent*, *equivEvent*, $agree_{pre}$, and $agree_{post}$. We indicate in the following predicates that specified by different JavaDL Dependency Cluster by adding the label to the predicate name.

Definition 8.21 (Combined Non-interference Specification)**.** The predicates $invEvent^{combined}$, $equivEvent^{combined}$, and $agree_{pre}^{combined}$ for a Combined Dependency Cluster with label *combined* and combining JML Dependency Clusters with labels $l_1, \ldots, l_n$ are defined as

$$\begin{aligned}
invEvent^{combined}(e) &:\Leftrightarrow \bigwedge_{x=1}^{n} (invEvent^{l_x}(e)) \\
equivEvent^{combined}(e_A, e_B) &:\Leftrightarrow \bigwedge_{x=1}^{n} (equivEvent^{l_x}(e_A, e_B)) \\
agree_{pre}^{combined}(h_A, h_B) &:\Leftrightarrow \bigwedge_{x=1}^{n} (agree_{pre}^{l_x}(h_A, h_B)) \\
agree_{post}^{combined}(h_A, h_B) &:\Leftrightarrow \bigwedge_{x=1}^{n} (agree_{post}^{l_x}(h_A, h_B))
\end{aligned}$$

Combined Dependency Clusters do not need to be verified, since as shown in Theorem 4.1, a remote method is non-interferent w.r.t. a combined non-interference specification if it is non-interferent w.r.t. all non-interference specifications that are combined.

8.4.6 Bean-Level Verification

To show that a bean is non-interferent w.r.t. a bean-level JML Dependency Cluster specification, we have to verify according to Definition 3.13 and Theorem 3.3 that all remote methods are non-interferent w.r.t. the specification. In Theorem 4.2 we showed that service-level Dependency Clusters can be used to show that a service is non-interferent w.r.t. a component-wide non-interference specification without reasoning about a program. We want to make use of this property for remote method non-interference.

To do so, we provide a specification mechanism which allows for each remote method to directly provide the method-local Dependency Cluster, which implies the bean-wide specification. This specification represents the connection between component-level non-interference and service-level non-interference.

The following method-level specification states that one or more remote method-level Dependency Clusters satisfy a component-global Dependency Cluster.

/*@ cluster g \satisfied_by l; */

where g is a label used for a component-level JavaDL Dependency Cluster and l is a label of a method-level JavaDL Dependency Cluster.

Example 8.10. We can specify for each remote method provided by the bean `Cart` which local clusters are used in order to verify that the bean-level Dependency Cluster `prodLowCluster` from Example 8.9 is also a Dependency Cluster for the remote method. In the following listing, we use the cluster `clusterBuy1` for the service `buy` and `cluster3` for `pay`.

```
public class Cart implements CartIF {
  ...
 /*@ cluster prodLowCluster \satisfied_by clusterBuy1;*/
 public int buy(int prod, int price, int amount) {...}

 /*@ cluster prodLowCluster \satisfied_by cluster3; */
 public int pay(int ccnr) {
}
```

Theorem 4.2 directly provides us with the proof obligation which we proved to verify that a method-level Dependency Cluster implies a component-level non-interference specification. We formalize the condition as a direct translation of the original condition in JavaDL:

Theorem 8.1. *A method m declared remote, which is non-interferent w.r.t. a JavaDL Dependency Cluster dc_l with label l is also non-interferent w.r.t. a*

JavaDL Dependency Cluster or Combined Dependency Cluster dc_g with label g, if the following formula is valid.

$$\begin{aligned}
&\texttt{self} \mathrel{\dot{\neq}} \texttt{null} \land \mathit{callingComp} \mathrel{\dot{\neq}} \texttt{null} && (8.53)\\
&\land \mathit{evCaller}(e_A) \mathrel{\dot{\neq}} \texttt{null} \land \mathit{evCaller}(e_B) \mathrel{\dot{\neq}} \texttt{null} && (8.54)\\
&\land \mathit{evClient}(e_A) \mathrel{\dot{\neq}} \texttt{null} \land \mathit{evClient}(e_B) \mathrel{\dot{\neq}} \texttt{null} && (8.55)\\
&\land \mathit{wellformed}(\mathit{evHeap}(e_A)) \land \mathit{wellformed}(\mathit{evHeap}(e_B)) && (8.56)\\
&\land \mathit{wellformed}(h_A) \land \mathit{wellformed}(h_B) && (8.57)\\
&\land \mathit{wellformed}(h_{2A}) \land \mathit{wellformed}(h_{2B}) && (8.58)\\
&\land h_A^{post} \doteq \mathit{anon}(h_{2A}, \mathit{mods}, h_A) \land h_B^{post} \doteq \mathit{anon}(h_{2B}, \mathit{mods}, h_B) && (8.59)\\
&\dot{\rightarrow} && (8.60)\\
&\qquad \mathit{equivEvent}^g(e_A, e_B) \mathrel{\dot{\rightarrow}} \mathit{equivEvent}^l(e_A, e_B) && (8.61)\\
&\quad \land \mathit{agree}^g_{post}(h_A, h_B) \mathrel{\dot{\rightarrow}} \mathit{agree}^l_{post}(h_A, h_B) && (8.62)\\
&\quad \land ((\mathit{equivEvent}^l(e_A, e_B) \land \mathit{isCallable}(e_A) \land \mathit{isCallable}(e_B)) && (8.63)\\
&\qquad \mathrel{\dot{\rightarrow}} \mathit{equivEvent}^g(e_A, e_B)) && (8.64)\\
&\quad \land (\mathit{agree}^g_{post}(h_A, h_B) \land \mathit{agree}^l_{post}(h_A^{post}, h_B^{post})) && (8.65)\\
&\qquad \mathrel{\dot{\rightarrow}} \mathit{agree}^g_{post}(h_A^{post}, h_B^{post}) && (8.66)
\end{aligned}$$

where

- *m_{id} is the representative for the method m*
- *isCallable is the formula:*

$$\begin{aligned}
&\mathit{isCallable}(e) :\Leftrightarrow\\
&\qquad (\mathit{evMethod}(e) \doteq m_{id} \land \mathit{evCaller}(e) \doteq \mathit{callingComp}\\
&\qquad\qquad \land \mathit{evClient}(e) \doteq \texttt{self})\\
&\qquad \lor (\mathit{evMethod}(e) \doteq m_1 \land \mathit{evCaller}(e) \doteq \texttt{self}\\
&\qquad\qquad \land \mathit{evClient}(e) \doteq o_1)\\
&\qquad \lor \ldots\\
&\qquad \lor (\mathit{evMethod}(e) \doteq m_n \land \mathit{evCaller}(e) \doteq \texttt{self}\\
&\qquad\qquad \land \mathit{evClient}(e) \doteq o_n)
\end{aligned}$$

- *$\mathit{callable} = (\mathit{exp}_1.m_1, \ldots, \mathit{exp}_n.m_n)$*
- *$o_i = \{\texttt{heap} := \mathit{evHeap}(e)\}(\mathit{exp}_i)$*
- *mods is the assignable set of m*

We omit the proof for the theorem since it is a direct JavaDL encoding of Theorem 4.2.

The first part of the formula (8.53 to 8.59) ensures that the events are wellformed in the sense that the calling bean and the client bean are non-null,and the heaps are wellformed. The post-heaps are an anonymization of the pre-heaps, considering the assignable set of the remote method the specification belongs to. The second part (from 8.61) is a direct JavaDL representation of the formula in Theorem 4.2. The helper formula *isCallable* formalizes whether an event, according to the callable set of the method, can actually appear in the history generated by the execution of the method.

Note that the formula in Theorem 8.1 does not contain a program, as does the formula in Theorem 4.2. Verifying the formula thus is first order predicate logic reasoning with substitution due to the updates in the formula.

8.5 Case Study

We implemented the rules and JML extensions in the KeY tool with slight simplifications, and applied it to a component-based web shop system. The tool, the web shop implementation, and the specifications can be found at the accompanying website[2]. In this section, we describe the system, its functionality and the Java beans implementing the functionality. We present the non-interference specification for the system and report on the verification process.

The case study is based on a previously published analysis of the case study by Greiner et al. [2017b] and Greiner et al. [2017a]. We focus here on the JML Dependency Cluster verification using our extension of KeY, while re-using some results from these publications.

8.5.1 Web Shop System Description

The web shop system provides interfaces for interacting with customers, employees at the delivery department, and at the billing department. The web shop also requires interfaces from a financial institute and the product database storing product information and prices. Figure 8.2 shows the interfaces and the remote methods declared by them.

A customer uses the remote method `addToCart`, declared in the interface `CartIF` to add products to his shopping cart. In order to check which products he already put into his cart, he can call the remote method `getCartContent`. The interface `AccountIF` provides remote services for managing the customer's account information. `orderElementsInCart` performs the payment process with the financial institute and marks the products in the cart as ordered. With the remote methods `setName`, `setAdress`, `setCCNr`, and `setCVC`, the customer can provide his name, delivery address, credit card number, and his card validation code for the credit card.

[2]`https://formal.iti.kit.edu/~greiner/niframework/`

CartIF		
Service	Parameter	Return Value
`getCartContent`		`OrderElement[]`
`addToCart`	`int prodId`, `int amount`	`boolean`

AccountIF		
`orderElementsInCart`		`boolean`
`setName`	`char[] name`	
`setAdress`	`char[] adr`	
`setCCNr`	`int ccnr`	
`setCVC`	`int cvc`	

BillingIF		
`getBillsToSend`	-	`Bill[]`

DeliveryIF		
`getDeliverySheets`	-	`DeliverySheet[]`

BankIF		
`makePayment`	`char[] name`, `int ccnr`, `int cvc`, `int amount`	`boolean`

ProductDBIF		
`getProductPrice`	`int prodId`	`int`

Figure 8.2: Interfaces provided and required by the Webshop system

An employee at the billing department calls `getBillsToSend` provided via the interface `BillingIF` to get the information required to issue bills to the customer. An employee at the delivery department calls the remote method `getDeliverySheets` provided by the interface `DeliveryIF` to gain the information necessary to package ordered products and ship them to the customer's address.

The web shop system is made up of five components, implemented as Java beans: The component `Cart` implements functionalities of the shopping cart; `Account` implements the customer's account; `OrderDB` implements the database which keeps track of all orders and their status; `Billing` prepares the relevant data from the `OrderDB` bean for employees working in the billing department; and `Delivery` does the same for the delivery department.

The web shop application consists in total of the five aforementioned beans, and the six external interfaces mentioned in Figure 8.2. The components are internally connected via four additional interfaces, declaring 12 additional remote services.

8.5.2 Non-Interference Specification

Domain-Motivated Specification The overall domain-motivated security specification states who (customer, delivery department, billing department) may gain knowledge about which input information. For example, the billing department may gain knowledge about the last four digits of the credit card used for payment, since this information should be printed on the bill. The delivery department must not gain knowledge about this information at all. Further, the billing department may gain knowledge about the price of the products payed by the customer, while the delivery department does not need this information.

In the specification, we assume that the customer has access to the interfaces `AccountIF` and `CartIF`, the delivery department has access to the interface `DeliverIF`, and the billing department to `BillingIF`. Each of theses stakeholders can see all output events provided via the respective interfaces. The full domain-motivated specification of who may know what information can be found in Appendix A.2.2.

Dependency Cluster Specification We provide JML Dependency Cluster specifications in two ways: For one we manually specify JML Dependency Clusters using the JML specification mechanism discussed earlier in this chapter. And second, we use an extension of the JOANA tool[3] to extract Dependency Clusters automatically from the implementation of the remote services. The manually specified Dependency Clusters are necessary, since the automatic tool sacrifices precision for automation and therefore does not extract all Dependency Cluster necessary to verify the concrete information flow specification.

We manually specified a total of 22 Dependency Clusters using our JML extension.

Additionally, we automatically extracted Dependency Cluster for the remote methods using an extension of the tool JOANA by Martin Mohr (see Greiner et al. [2017b] and Greiner et al. [2017a] for details on the tool). Given the Byte code of a Java program, JOANA calculates a program dependency graph (PDGs) for the program. The nodes in a PDG represent statements and expressions of the program under analysis, including method calls and parameters. The edges represent dependencies between the nodes. Data dependencies represent for a statement that it depends on the value of a previous expression, and control dependencies represent for a statement that its execution depends on a previous statement. The backward slice for a node, in particular a return value or field, contains all expressions and statements on which the value of the node depends. By calculating backward

[3]`https://pp.ipd.kit.edu/projects/joana/`

slices for return values and fields, we can calculate a dependency cluster for the remote method under analysis.

We gained a total of 502 automatically extracted Dependency Cluster. In order to verify non-interference for our web shop example, we selected 245 of these Dependency Cluster and provided corresponding JML Dependency Cluster specifications for each of them.

We finally provide for each bean in the web shop a specification for two combined Dependency Cluster, one representing the non-interference property for the delivery department and one for the billing department. For each remote method we provide for the two bean-global Dependency Clusters a `satisfied by` clause. All specifications can be found in the online available material.

8.5.3 Verification

We applied our KeY implementation on each of the manually specified JML Dependency Cluster as well as the satisfies-clause for each service. Both of these verification goals posed their own challenges.

Program verification for simple JML Dependency Clusters for simple programs runs automatically most of the time. Also, verifying very precise dependency cluster, e.g. a Dependency Cluster expressing declassification of the last four digits of a credit card number, did not pose an additional challenge.

Manual interaction is necessary during the proof, however, if dependency clusters are larger, i.e. event- or state equivalence depends on many different expressions. The main problem in this case is especially the need for proving object isomorphism, which leads to infeasible many quantifier instantiations a user has to provide. One way to overcome this problem is by introducing a proof obligation for equivalence of outgoing messages, which assumes newly generated objects in both runs of a service to be equal, and showing equality of objects instead of an existence of an isomorphism. While this solution should heavily reduce manual interactions, it is an over-approximation of the proof obligation shown here.

Additional manual interactions are typical for KeY, e.g. when heaps with many manipulations have to be simplified or when nested quantifier have to be instantiated. This problem is not unique to our approach but a general challenge in program verification with KeY.

The grade of automation for verification of satisfies-clauses depends mainly on two properties of the proof obligations. For one, if a service potentially calls many other services, i.e. many event equivalences have to be verified, many case splits are necessary to support the prover.

A second criterion is the complexity of the specification of the low state. For one, if many nested quantifier are contained in the specification, again many manual interactions are necessary to show global low equivalence of the poststates. And second, if many object-valued expressions are contained in the specification of the low part of the state, showing object isomorphy is practically infeasible. We therefore simplified the proof obligation to require $agree_{pre}$ for the poststates instead of $agree_{post}$.

In conclusion, we did require support from an automatic tool to verify simple dependency cluster for our components in order to make the verification feasible. Dependency Cluster allowed us to combine the tools on a very fine-gained specification level. We also re-used many dependency cluster, including manually specified ones, for verification of non-interference w.r.t. the delivery department and the billing department. This freed us from verifying both properties from scratch, and verifying the second property was only a small overhead in the verification.

We also found that improvements for the verification are necessary for bigger, more realistic programs. We expect optimizations similar to those by Scheben and Schmitt [2014] for sequential Java programs to be possible and useful for our proof obligations as well. We also think that additional support for method contracts and block contracts can make precise dependency cluster verification a more feasible task.

We finally want to mention that we selected the 245 Dependency Cluster out of the 502 automatically generated ones manually. This task is error prone and time consuming. However, we assume that it is possible to automate this task by providing an overall domain-driven specification, automatically selecting a set of the given Dependency Cluster, and check the weakening property of the selected Dependency Cluster against the domain-motivated Specification. This check could, for example, be efficiently executed using SMT solver based tools.

8.6 Related Work

We distinguish in the discussion of related work between related work for deductive verification of object-oriented interactive programs, deductive verification of non-interference in object-oriented sequential programs, and automatic program analysis approaches for non-interference.

8.6.1 Deductive Verification of Interactive Programs

ABS (Johnsen et al. [2012]) is an executable modeling language for distributed interactive systems, which supports synchronous as well as asynchronous communication. Components in ABS communicate via message passing, similar to our DSCs and the modeling language is object oriented. Specifications for component-based systems modeled in ABS are provided as invariants over global histories of the system, typically expressing some kind of functional well-behavior of the system.

ABSDL (Din et al. [2015a]) is a dynamic logic for the ABS modeling language, and the tool KeY-ABS, a deductive verification tool based on KeY for modular reasoning about ABS models, is presented. Similar to our work, the authors provide a calculus for ABSDL, which also supports events with a similar structure as our extension of JavaDL, as well as local histories for components. Contrary to our work, the calculus for ABS supports different forms of synchronization, as well as futures in the specification language, and allows reasoning about global histories of systems consisting of components (Din et al. [2015b]). While they focus on the verification of global properties of a composed system, we concentrate on reasoning about properties local to remote methods, which we require in order to verify non-interference properties w.r.t. Dependency Clusters.

History-based reasoning for asynchronous method calls in interactive systems was first done for the programming language Creol in Dovland et al. [2005]. The approach was adapted to dynamic logic by Ahrendt and Dylla [2012] and later on simplified by Din et al. [2012] and extended to support futures by Din and Owe [2014] and Din and Owe [2015]

8.6.2 Deductive Verification of Object-oriented Non-interference

We limit the discussion here to work on deductive verification of non-interference properties for sequential object-oriented programs.

Amtoft et al. [2006] present a Hoare-like logic (Hoare [1969]) specialized for the deductive verification of non-interference in a small object-oriented programming language. The approach is termination insensitive and does not allow what-declassification in the sense that only variables and fields can be specified to be low, not parts of them. Amtoft and Banerjee [2007] extend this approach with what they call conditional information flow, where the classification of input or output of a program may depend on a condition over the state of the program. This work was implemented as a software contract approach for SPARK Ada and supports a simple while language (Amtoft et al. [2008]). Amtoft et al. [2010] added array support for this approach.

Bubel et al. [2008] present an approach where dependencies between program locations (i.e. variables and fields) are tracked using an approach based on abstract interpretation and deductive verification.

Self-composition is a technique where non-interference is directly encoded in a formula by executing a program twice and expressing equivalence of pre- and poststates in a formula. Concrete applications of self-composition for non-interference formalization was presented by Darvas et al. for dynamic logic, by Barthe et al. for temporal logic, and by Joshi and Leino [2000] for the weakest precondition calculus.

Darvas et al. provided also the basis for a series of work on non-interference analysis based on dynamic logic for sequential Java programs. Scheben and Schmitt [2012] provide a formalization of low equivalence of terms over a state is presented in the form of views that allows declassification of nearly arbitrary terms. We extended this approach for the specification of Dependency Clusters and our formalization of Dependency Clusters for remote methods can be seen as an extension of their non-interference property with events and histories. Beckert et al. present a specification method for JML and Java programs, on which our JML extension for Dependency Cluster is based. Scheben and Schmitt [2014] present optimizations for reasoning about non-interference properties in Java programs, which reduces significantly the effort for program analysis. Their results in general can be applied to the analysis of services and Dependency Clusters, which we leave for future work.

8.6.3 Automatic Program Analysis for Non-interference

A large body of work on automatic approaches for program analysis for non-interference properties is on type systems. A good overview of type systems for non-interference analysis is provided by Hedin and Sabelfeld [2012]. Typically, type systems are designed for rather simple programming languages and not easily extendable for practically used programming languages. Additionally, type systems typically sacrifice precision in order to be applied automatically, which prevents them from allowing expressive semantic declassification of information.

An exception here is the PER model presented by Sabelfeld and Sands [2001], where partial equivalence relations are used to describe which partial information of some input variable is declared low. The work also presents a type system which allows analysis for a simple while language with non-deterministic commands. A line of work on dependent types (e.g. Zheng and Myers [2007],Swamy et al. [2010], Swamy et al. [2011], Nanevski et al. [2011], and Lourenço and Caires [2015]) allows a dynamic classification of information, which may also depend on constraints, e.g. , conditions over a state. As a consequence, type checking generates proof obligations that have to be checked for validity, which is in general an undecidable task.

Another exception is JIF (Myers et al. [2006]), a security-typed extension of Java. JIF supports a realistic programming language, including exceptions, and object-orientation.

Hammer and Snelting [2009] propose a program analysis technique based on program dependency graphs implemented in the tool JOANA. Inputs and outputs of a program are labeled high or low. The authors show that if the program dependency graph does not contain a path from a high input to low output, the program is non-interferent w.r.t. its specification. Based on this technique, the tool JOANA can check non-interference properties for realistic

Java programs. An modification of JOANA was used to extract Dependency Cluster from remote methods by Greiner et al. [2017b] and Greiner et al. [2017a].

8.7 Conclusion

In this chapter, we applied our framework from Part I to concrete JavaEE implementations of components, so-called beans. We extended JavaDL by events and histories and a rule for symbolic execution of remote method calls. The rule directly formalizes the effects of serialization and deserialization of parameters and return values on the heap, as well as the resulting proof obligation for remote methods. We further extended JML with specification primitives for Dependency Cluster and bean-wide non-interference specifications. Using both of these extensions we provide a proof obligation for Dependency Cluster of remote methods.

Additional specification primitives for combining Dependency Clusters and bean-level non-interference specifications allow a modular specification of non-interference properties for beans, as described for DSCs in general in Chapter 4. The condition whether a remote method is non-interferent w.r.t. a bean-wide non-interference specification is given as a JavaDL formalization of Theorem 4.2 in first order predicate logic.

Our specification language is intuitive while practically still providing the expressiveness of equivalence relations. Further, Dependency Cluster formulated in our JML extension can be used as building blocks for elaborate non-interference specifications and combinations of Dependency Cluster is made explicit on a syntactical level.

We provide an implementation of the techniques described this chapter as an extension of the KeY theorem prover. This way, we show that it is feasible to build precise program analysis technique based on our framework for a real, object-oriented programming language.

We presented here an approach for program analysis for individual beans and left open how bean-wide specifications are motivated. While typically in the literature specifications of this form are given by some oracle (or domain expert), in our case it would make sense if these specifications are related to some overall systems-engineering approach. Yurchenko et al. [2017] shows that code-generation techniques can be extended to enrich code skeletons gained from model-based specifications (see Chapter 7) with non-interference specifications. These specifications are then domain-motivated and ensure a system-wide security property with respect to a specific attacker model.

While deductive verification, as used in this chapter, allows verification of very precise specifications, it often does not scale very well. Martin Mohr show (Greiner et al. [2017b] and Greiner et al. [2017a]) that Dependency Cluster can be extracted from programs efficiently using an automatic static program analysis tool, in that case based on the tool JOANA. While automatically generated Dependency Cluster do not cover flows which express semantic declassification, they amount for the vast majority of relevant dependencies caused by the implementation of a remote method. Only very specific dependencies have to be specified and verified manually with our interactive tool. We make use of this Dependency Cluster extraction in our case study.

We would like to stress that the modularity of Dependency Clusters allows a combination of tools on service level and limited to partial information flow specifications. While feasibility for this combination was shown with JOANA, this does not mean that there is a limit to this particular combination. We expect that many different forms of program analysis techniques, including dynamic techniques, may be useful in order to achieve scaleablitiy. The more techniques are available for analysis, the more options exist in order to combine the strength of different approaches while avoiding weaknesses of the respective approach.

Part III

Beyond the Framework

9 Trace-based Non-interference

9.1 Introduction

In this chapter, we discuss how our framework from Part I can be extended to support non-interference specifications where the sensitivity of a message depends on the history of previously communicated messages. In the original definitions, whether or not a message or parts of the information communicated in a message is low may only depend on the values of the parameters of the message. This limitation prevents our framework from being used in a context where secrecy of information depends for example on whether a user was previously added as a friend, a property common in social networks.

In this chapter we discuss a privacy preserving video surveillance system as an example. One requirement of the video surveillance system is to ensure that an operator watching the system's monitors does not gain information about people under surveillance, which the operator should not know. Whether or not some input to the system contains this sensitive information depends on the history of inputs to the system, i.e. the non-interference specification the video surveillance system should obey to is trace-dependent. Along this example, we revisit strategies and non-interference from Chapter 3 and extend them to support trace-based non-interference. We further develop specifications for the services of the system and show for an implementation that the system obeys to the privacy requirement of our example.

In the next section we introduce the privacy preserving video surveillance system and the intuitive idea of the information flow property which the system should guarantee. In Section 9.3 we provide a trace-dependent equivalence relation which formally defines which input information is low from point of view of the operator. We thus formalize the privacy requirement motivated by the domain of the video surveillance system. In Section 9.4 we extend several definitions from Chapter 3 such that we gain a notion of trace-dependent non-interference for DSCs, as far as needed for the example system. We additionally provide a trace-dependent non-interference notion for services, which implies trace-based non-interference for a DSC. We further we provide trace-independent conditions for services, which imply trace-based non-interference for services. In Section 9.5 we present the implementation of the core component of the video surveillance system and provide a formalization of the trace-independent conditions in JML, and report on the formal verification of the component w.r.t. these conditions with KeY. After this we discuss related work and conclude the chapter.

The work presented in this section is an extension of the work published in Greiner et al. [2013] and Birnstill et al. [2015].

9.2 An Example: Privacy Preserving Video Surveillance

In this section, we introduce a smart video surveillance system with enhanced privacy protection mechanisms for people under surveillance. We present here a simplified system which is inspired by an implementation, which is discussed by Birnstill and Pretschner [2013]. The system was designed in order to make video surveillance possible in sensitive environments, as for example hospitals. The purpose of the surveillance system is detecting dangerous situations, e.g. patients lying on the floor helplessly or theft being committed on the property of patients in the hospital.

A hospital is a very sensitive place, where different regulations have to be obeyed to. Of course it is not allowed to install video surveillance systems in patient's rooms or restrooms, since this would constitute a strong invasion of privacy of people. Even if there is no surveillance at these sensitive places, worker protection regulations also would not allow to put the entire hospital under surveillance, because this would allow the employer to perform strict surveillance of employees.

The smart surveillance system we use as an example in this chapter therefore implements several privacy protection mechanisms. The first protection mechanism is that the system does not directly provide surveillance images on a screen. Instead, only abstract information about the areas under surveillance is provided to the operator of the system. Figure 9.1 shows an abstract map, as it is displayed to the operator.

The smart video surveillance system uses object recognition to extract where people are located in the hospital. Then, these people are shown on the screen abstractly as white figures. Additionally, the system can detect dangerous situations, like people lying on the floor. In this case, the operator receives an alert from the system and can request live images to examine if an actual emergency situation is shown and whether she has to organize help. These requests are logged such that misuse of live images can be detected.

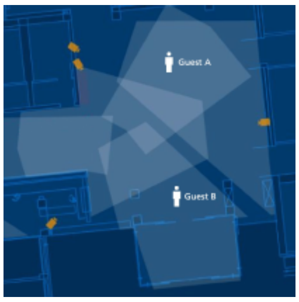

Figure 9.1: Screen shot of abstract map

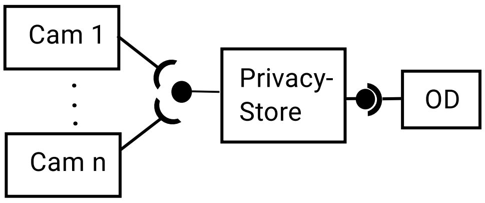

Figure 9.2: System diagram for the smart surveillance system

The abstraction of concrete information makes it harder to directly identify for each abstractly shown figure which person in reality it represents. However, employees of the hospital, e.g. nurses and doctors, have characteristic patterns of movement due to their tasks. By observing one abstract figure over a period of time would allow to identify the employee which it represents.

The smart surveillance system therefore implements a second protection mechanism which states that employees should not be displayed in the abstract map at all. When an employee enters the hospital, she can identify herself as a member of the *protected group* (as an employee) by holding a smartphone into a camera displaying a QR code. Then the system should not show this person at all on the abstract map.

In order to achieve this functionality, the smart video surveillance system has to track this person. Only if the system knows, which person has previously identified herself a an employee, the system can decide not to show her on the screen. To perform the tracking in a reliable way, the system has to store sensitive information, i.e. the information which should not be displayed, about the person. The situation where sensitive information has to be stored in order to protect it is called the *tracking paradox* (Greiner et al. [2013]). We can formalize the property that stored sensitive information must not be available to the operator's desk as an information flow property.

In Figure 9.2 the general architecture of the smart surveillance system shown. Figure 9.3 shows the services provided by the privacy store.

```
public class PrivacyStore {
  public void updateObservation(int[] observation) {...}
  public void registerCoWorker(int[] observation) {...}
  public int[] getGuest(int pos) {...}
}
```

Figure 9.3: Interface of a privacy store

The system consists of three kinds of components. The first kind of components are the *cameras*. The cameras record pictures of a certain area and perform object recognition tasks on the images and extract features, e.g. the location of a person, his height, and hair color. Cameras can also detect whether a person identifies himself with a QR code displayed on a smart phone. The cameras pass the features, but not the original image, on to the privacy store. If the camera identified a person, it calls the service `updateObservation`, if the camera additionally identified the person to register with a QR code, it calls the service `registerCoWorker`.

The *operator's desk* (OD in Figure 9.2) is the component which shows the abstract map to the operator. The component frequently calls `getGuest` provided by the privacy store and receives as a return value an observation, i.e. the features of one person. The operator's desk then displays the gained information on the abstract map.

The *privacy store* is the core component which stores the observations and performs the tracking of persons. For each observation the privacy store receives from a camera, it has to decide whether or not the observation shows some person already in the system, or if a new person entered the area under surveillance. The privacy store also has to ensure that no observation of a registered person is forwarded to the operator's desk. So all privacy preserving functionality is encapsulated in the privacy store and we limit our presentation in the following to this component.

We make the following idealizing assumptions about the system. For one, we assume that feature extraction performed by the cameras is perfect. It always provides the correct features, never misses a person and never identifies a non-exiting person. And second, we also assume that the tracking algorithm is perfect in the sense that every observation provided by the camera to the privacy store is matched to the correct person already known to the system.

9.3 Specification

In order to analyze whether the privacy store component only provides non-sensitive information to the operator's desk, we require a specification of sensitive and non-sensitive information. We define sensitive and non-sensitive information based on inputs and outputs of the privacy store, i.e. based in the event trace of the component.

Potentially sensitive information is provided to the privacy store via the inputs to the services `updateObservation` and `registerCoWorker`. Simply put, an observation is non-sensitive, i.e. low, if it does not show an employee. An observation is also non-sensitive, if a member of the protected group registers himself as such. In this case, the operator is able to see on his screen that one abstract figure disappears from the abstract map. Finally, all information provided to the operator's desk as a return value of the service `getGuest` has to be low.

In this section, we provide a formal definition of sensitive and non-sensitive information. We reuse the formal notations introduced in Part I.

9.3.1 Tracking Formalization

The surveillance system distinguishes two groups of people under surveillance. One group consists of the employees, which may be tracked, however whose personal information must not be shown to the operator. Their personal information is high. We call these people *registered*, because they identified themselves to the system via registration at a camera. The other group of people are those under surveillance who are shown at the operator's desk for security reasons. We call this group of people *guests*. Their private information is low.

Intuitively a person is registered, if she registered herself at the system. From the point of view of the privacy store component, this information is provided as a call to the service `registerCoworker`, while the provided parameter contains the person's features. A person can move in the building while frequently observations of the person are provided to the privacy store by the cameras as calls to the service `updateObservation`. The privacy store has to ensure that new observations are correctly matched with previously observed people. Especially, if a registered person is shown in the provided feature vector, this information must not be conflated with information about a non-registered person.

In order to formalize the tracking functionality, we introduce the two predicates. *samePerson* evaluates to true if for a given trace two observations in the trace show the same person. The tracking predicate ξ evaluates to true for two observations, if the same person is shown in two observations.

We leave the tracking predicate $\xi : \mathbb{D} \times \mathbb{D}$ undefined for the moment. We formally define $samePerson : (int \times int \times \mathbb{T})$ as follows:

Definition 9.1 (*samePerson*)**.** Let α, β, $\gamma \in \{\mathit{Ini}(\mathit{updateObservation})$, $\mathit{Ini}(\mathit{registerCoWorker})\}$.

$$
\begin{aligned}
&\mathit{samePerson}(i,j,t) \Leftrightarrow && (9.1)\\
&\quad i = j \lor && (9.2)\\
&\quad (i < j < |t| \land && (9.3)\\
&\quad (\exists i < k < j \cdot \mathit{samePerson}(i,k,t) \land \mathit{samePerson}(k,j,t) \lor && (9.4)\\
&\quad (\exists t = t_1 \frown \alpha.v \frown t_2 \frown \beta.w \frown t_3 \cdot && (9.5)\\
&\qquad |t_1| = i - 1 \land |t_2| = j - i - 1 \land \xi(v,w) \land && (9.6)\\
&\quad \neg\exists i < k < j \cdot t = t_1 \frown \alpha.v \frown t_2' \frown \gamma.x \frown t_3' \land && (9.7)\\
&\qquad |t_2'| = i - k - 1 \land \mathit{sameObs}(i,k,t)))) && (9.8)
\end{aligned}
$$

samePerson takes as parameters two indices and a trace. The definition contains several case distinctions: For one, *samePerson* evaluates to true, if the two indices are equal, i.e. i and j point to the same input event in the trace (Line 9.2).

The second case (Line 9.4) covers transitivity of *samePerson*. If in a trace $t_1 \frown \alpha.v \frown t_2 \frown \beta.w \frown t_3 \frown \gamma.x \frown t_4$, the observation in the events $\alpha.v$ and $\beta.w$ show the same person and the events $\beta.w$ and $\gamma.x$ show the same person, then of course $\alpha.v$ and $\gamma.x$ also show the same person.

The third case (Line 9.6) reduces the semantics of *samePerson* to the tracking predicate ξ. Two events contain an observation showing the same person, if the tracking predicate ξ evaluates to true for the two observations, unless there is another observation at position k between i and j, such that i and k show the same person (Line 9.8). With this exception, we ensure that ξ only identifies the same person in two observations, if it maps to the last observation of that person, and not to a person which has moved to another place in the meantime.

The definition of *samePerson* provides us with a domain-driven formalization of the tracking functionality, expressed only using trace properties. For the remainder of this chapter it is convenient to have a more intuitive notion of the people tracked by the system. The function $\mathit{lastObs} : \mathbb{T} \mapsto \mathcal{P}(\mathbb{N})$ returns for a given trace t the indices showing the last observation of all persons in the trace:

$$\mathit{lastObs}(t) := \{i \mid 1 \le i \le |t| \land \neg\exists i < j \land j \le |t| \cdot \mathit{samePerson}(i,j,t)\} \quad (9.9)$$

As stated above, we leave the definition of ξ open for the moment, however we do assume that tracking works correctly. This means, we assume that the tracking predicate is sufficiently precise such that the resulting *samePerson* predicate only evaluates to true, if the two indexed observations actually show the same person in the physical world. We do not want to formalize the real world here, but we can formalize a part of the assumption: Tracking does not confuse two different people.

Definition 9.2 (Perfect Tracking)**.** The tracking predicate ξ is perfect, if

$$\begin{aligned}&\forall t, v, i, j \cdot \\ &\quad (i, j \in \mathit{lastObs}(t) \ \wedge\ t[i] = \alpha.w \ \wedge\ t[j] = \beta.x \ \wedge\ \xi(w, v) \ \wedge\ \xi(x, v)) \\ &\qquad \implies\ i = j\end{aligned}$$

Definition 9.2 expresses that a new observation v can never be matched by ξ to the last observation of two different persons.

Similar to *lastObs*, the functions *registered* and *guests* provide us with the last observations of registered and non-registered persons under surveillance.

$$\begin{aligned}\mathit{registered}(t) =& \{i \mid i \in \mathit{lastObs}(t) \wedge \\ &\qquad \exists j < i \cdot t[j] = \mathit{Ini}(registerCoWorker).v \wedge \\ &\qquad\quad samePerson(j, i, t)\} \\ \mathit{guests}(t) =& \mathit{lastObs}(t) \setminus \mathit{registered}(t)\end{aligned}$$

registered returns the set of indices of the last observation of a person, which previously registered as a member of the protected group. *guests* returns the last observation of all persons, except registered persons.

9.3.2 Formal Domain-driven Security Specification

We can now formalize whether messages are high or low according to the informal description given in the previous section. A call to `updateObservation` is high, if it provides an observation showing a person that registered as a member of the protected group. Further, a call to `registerCoworker` is high, if the provided observation shows a previously registered person or if it shows a person who has not previously been seen by the system. However, if a guest is shown in the observation, the call to `registerCoworker` is low, since the operator should be allowed to see the abstract figure disappearing from the screen.

Whether or not the existence of a message is high, depends on the trace previously communicated by the privacy store component. For the sake of simplicity, we define the termination events of the two services `updateObservation` and `registerCoworker` to be high, iff the previous call was high. Note that we were not able to provide this particular specification of high termination event with our framework described in Part I.

The service `getGuest` is called by the operator's desk and therefore its existence as well as the provided parameter is low. The same is true for its termination and the return value, since this is the information provided by the operator's desk to the operator by updating the abstract map.

We define the resulting trace-dependent equivalence relation $\overset{T}{\sim} \subseteq (\mathbb{T} \times \mathbb{M}) \times (\mathbb{T} \times \mathbb{M})$ as follows:

Definition 9.3. Given a trace t and a message $\alpha.v$. A message $\alpha.v$ after t is invisible, written $(t, \alpha.v) \overset{T}{\sim} \Box$, iff

$$\begin{aligned}
&(\alpha = \mathit{Ini}(\mathit{updateObservation})\wedge\\
&\qquad \exists i \in \mathit{registered}(t) \cdot t[i] = \beta.w \wedge \xi(w,v))\vee\\
&(\alpha = \mathit{Ini}(\mathit{registerCoworker})\wedge\\
&\qquad \neg(\exists i \in \mathit{guests}(t) \cdot t[i] = \beta.w \wedge \xi(w,v))\vee\\
&(\alpha \in \{\mathit{Fin}(\mathit{updateObservation}), \mathit{Fin}(\mathit{registerCoWorker})\}\wedge\\
&\qquad \exists t', \beta \in \{\mathit{Ini}(\mathit{updateObservation}), \mathit{Ini}(\mathit{registerCoWorker})\}, w \cdot\\
&\qquad\quad t = t' \frown \beta.w \wedge (t', \beta.w) \overset{T}{\sim} \Box)
\end{aligned}$$

Given traces t_1, t_2 and messages $\alpha.v$ and $\beta.w$. $\alpha.v$ after t_1 is trace-equivalent to $\beta.w$ after t_2, written $(t_1, \alpha.v) \overset{T}{\sim} (t_2, \beta.w)$, iff

$$((t_1, \alpha.v) \overset{T}{\sim} \Box \wedge (t_2, \beta.w) \overset{T}{\sim} \Box) \vee (\alpha = \beta \wedge v = w) \tag{9.10}$$

We now extend the definitions for strategy-based non-interference in Chapter 3 for our extended definition of equivalence relations over messages and traces. We re-define the projection function over traces such that all invisible events are removed from a trace. In the case of the example used in this chapter the equivalence class of a visible event only contains the event itself, we therefore do not need to consider equivalence classes here explicitly.

$$\begin{aligned}
\langle\rangle\restriction_{\overset{T}{\sim}} &:= \langle\rangle\\
(t \frown \alpha.v)\restriction_{\overset{T}{\sim}} &:= \begin{cases} t\restriction_{\overset{T}{\sim}} & \text{iff } (t, \alpha.v) \overset{T}{\sim} \Box\\ t\restriction_{\overset{T}{\sim}} \frown \alpha.v & \text{otherwise}\end{cases}
\end{aligned}$$

Note that we define the projection function recursively from the end of the trace. The main reason for this is because of the definition of $\overset{T}{\sim}$, the classification of a message as high or low only depends on the history of a message, never on the future.

Finally, we define trace equivalence analogous to Definition 3.1 in our framework in Part I.

Definition 9.4. Two traces t_1 and t_2 are trace-equivalent w.r.t. a specification $\overset{T}{\sim}$, written $t_1 \overset{T}{\sim} t_2$, iff

$$t_1\restriction_{\overset{T}{\sim}} = t_2\restriction_{\overset{T}{\sim}}$$

Given the equivalence relation defined in Definition 9.3 and the definition of trace equivalence in Definition 9.4, we can define trace-based non-interference as an extension of the definition in Chapter 3.

9.4 Trace-based Non-interference

In this section, we extend the notion of non-interference as introduced in Chapter 3 by allowing the trace-based equivalence relation $\overset{T}{\sim}$ as a specification. In Subsection 9.4.2 we then break the general trace-based non-interference notion for DSCs down to a trace-based non-interference property for the services provided by the privacy store component. Analogous to Part I, this gives us a property on service-level such that non-interferent services result in a non-interferent DSC. This non-interference property for services is still dependent on the history of the entire component, and we provide in Subsection 9.4.3 a non-interference property for the services provided by the privacy component which is independent from the entire trace of the DSC. This state-dependent non-interference property then can be checked using state-of-the-art deductive program analysis tools.

Please note that all service-local non-interference properties discussed in this section are specific to our example and do not directly provide a general concept for an unwinding of trace-based non-interference. However, we want to provide an idea on how a general framework for unwinding trace-based non-interference properties for DSCs can be developed.

9.4.1 Trace-based Component Non-interference

First, we extend the definition of strategies as a model of the environment. The original definition in Definition 3.3 requires a strategy to provide equivalent input messages for equivalent traces. We have to generalize this notion to *trace strategies*, since in this chapter message equivalence depends on the previously communicated trace.

Definition 9.5 (Trace Strategies)**.** A trace strategy ω is a function $\omega : \mathbb{T} \mapsto \mathcal{P}(\mathbb{M})$ such that
$\forall t_1 \overset{T}{\sim} t_2 \cdot \forall \alpha.v \in \omega(t_1) \cdot (t_1, \alpha.v) \overset{T}{\sim} \Box \;\vee\; \exists \beta.w \in \omega(t_2) \cdot (t_1, \alpha.v) \overset{T}{\sim} (t_2, \beta.w)$

We call the set of all trace strategies `TStrat`. The difference between trace strategies and the original strategies (Definition 3.3) is that the messages a strategy provides depend on the history of inputs and outputs earlier in an interaction. Analogue to the definition of strategies, the definition of trace strategies formalizes that the environment does not leak information.

We also extend the definition of equivalent strategies (Definition 3.5) to trace-equivalent strategies.

Definition 9.6 (Trace Strategy Equivalence)**.** Two trace strategies $\omega_1, \omega_2 \in$ `TStrat` are trace-equivalent, iff

$$\forall t, \alpha.v \in \omega_1(t) \cdot (t, \alpha.v) \overset{T}{\sim} \Box \vee \exists \beta.w \in \omega_2(t) \cdot (t, \alpha.v) \overset{T}{\sim} (t, \beta.w) \wedge$$
$$\forall t, \alpha.v \in \omega_2(t) \cdot (t, \alpha.v) \overset{T}{\sim} \Box \vee \exists \beta.w \in \omega_1(t) \cdot (t, \alpha.v) \overset{T}{\sim} (t, \beta.w)$$

Again, the definition is analogue to Definition 3.5 in the sense that two strategies are equivalent, if they produce equivalent traces after communicating equivalent traces in the past.

Finally, we define trace-non-interference for components using trace-strategies.

Definition 9.7 (Component Trace Non-interference)**.** A component c is trace-non-interferent w.r.t. a specification $\overset{T}{\sim}$, iff

$$\forall \omega_1, \omega_2 \in \texttt{TStrat}, t_1 \cdot \\ ((\omega_1 \overset{T}{\sim} \omega_2 \land \omega_1 \models c \xrightarrow{t_1}) \implies (\exists t_2 \cdot \omega_2 \models c \xrightarrow{t_2} \land t_1 \overset{T}{\sim} t_2))$$

The changes made compared to Definition 3.6 again are straight forward and only caused by the dependency of message equivalence on the entire trace.

We refrain here from a counterpart for cooperative strategies (Definition 3.8), since the privacy store component does not call services, and thus no strategy can block the execution of a component by refusing terminating messages for services called by the component.

On first sight, we merely replaced message equivalence $\sim$ with message-trace equivalence $\overset{T}{\sim}$ in the definitions. This replacement has profound consequences for the properties of the non-interference notion. Especially, trace-based non-interference is not compositional for LTS nor for DSCs.

We also would like to point out that whether or not some information in a message is high or low only depends on the *previously* communicated trace. We can not express specifications which would allow some information to be high or low depending on the future. For example, we can not express a message m to be low, if some time in the future a message m' is received by the component.

9.4.2 Trace-Based Service Non-Interference

Analogous to Definition 3.12 in Chapter 3 we provide a non-interference property for services such that non-interferent services can be composed to a non-interferent DSC. We re-use state equivalence $\approx$ to express the low information stored in a state. We extend the original notion of non-interference for services such that trace-equivalence is reflected in the non-interference property for services.

Note that all definitions from here are specific to the privacy store example. For the specific example of the privacy store component we know that between the call of a provided service and its termination, no other messages can be sent or received by the component. Further, for all messages it holds that either the message's existence itself is high, or the entire content of the communicated value is low.

Analogous to Chapter 3 we extend the original definition of visibility preserving services in Definition 3.11 to *trace-visibility-preserving* services as follows:

Definition 9.8 (Trace-Visibility-Preserving)**.** A service *serv* provided by `PrivacyStore` is trace-visibility-preserving w.r.t. $\approx$ and $\overset{T}{\sim}$, iff

$$\forall \sigma_1, \sigma_2, \alpha.v, \beta.w, t \cdot \tag{9.11}$$

$$(t, \alpha.v) \overset{T}{\sim} \Box \wedge \tag{9.12}$$

$$\langle body_c; \sigma_0 \rangle \xrightarrow{t} \langle body_c; \sigma_1 \rangle \wedge \tag{9.13}$$

$$\langle handler_{serv}; \sigma_1 \rangle \xrightarrow{\alpha.v \frown \beta.w} \langle SKIP; \sigma_2 \rangle \tag{9.14}$$

$$\Longrightarrow \tag{9.15}$$

$$\sigma_1 \approx \sigma_2 \tag{9.16}$$

The first important difference compared to Definition 3.11 in Line 9.13 is that we limit the consideration to states σ_1 which are actually reachable by the component after a full execution of a series of services. This is necessary, since we require the trace that is communicated by the component to the point when the new message $\alpha.v$ is received, in order to decide whether the message is visible or not. The second difference is that we do not need to consider intermediate messages communicated by the trace between the service call ($\alpha.v$) and its termination ($\beta.w$) (Line 9.14), since the services provided by the privacy store component do not call any services. Finally, we only require that the initial state is equivalent to the poststate of the service execution (Line 9.16), but not that the terminating messages are visible iff the initial message is visible. This is because the terminating message is invisible by definition iff the initial message is invisible (see Definition 9.3).

Analogous to Section 3.4, we extend the definition of non-interference for services in Definition 3.12 with trace-dependent equivalence.

Definition 9.9 (Service-Trace-Non-interference)**.** A service *serv* provided by `PrivacyStore` is *service-trace-non-interferent* w.r.t. $\approx$ and $\overset{T}{\sim}$ iff it is trace visibility preserving w.r.t. $\approx$ and $\overset{T}{\sim}$ (Definition 9.8) and

$$
\begin{aligned}
&\forall \sigma_1, \sigma_2, \sigma_1', \sigma_2', t_1, t_2 \cdot && (9.17)\\
&\quad \sigma_1 \approx \sigma_2 \wedge t_1 \overset{T}{\sim} t_2 \wedge && (9.18)\\
&\quad (t_1, \alpha.v) \overset{T}{\sim} (t_2, \alpha.v') \wedge && (9.19)\\
&\quad \langle \mathit{body}_{\mathit{PrivacyStore}}; \sigma_0\rangle \xrightarrow{t_1} \langle \mathit{body}_{\mathit{PrivacyStore}}; \sigma_1\rangle \wedge && (9.20)\\
&\quad \langle \mathit{body}_{\mathit{PrivacyStore}}; \sigma_0\rangle \xrightarrow{t_2} \langle \mathit{body}_{\mathit{PrivacyStore}}; \sigma_2\rangle \wedge && (9.21)\\
&\quad \langle \mathit{handler}_{\mathit{serv}}; \sigma_1\rangle \xrightarrow{\alpha.v \frown \beta.w} \langle \mathit{SKIP}; \sigma_1'\rangle \wedge && (9.22)\\
&\quad \langle \mathit{handler}_{\mathit{serv}}; \sigma_2\rangle \xrightarrow{\alpha.v' \frown \beta.w'} \langle \mathit{SKIP}; \sigma_2'\rangle \wedge && (9.23)\\
&\Longrightarrow && (9.24)\\
&\quad \sigma_1' \approx \sigma_2' \wedge && (9.25)\\
&\quad (t_1 \frown \alpha.v, \beta.w) \overset{T}{\sim} (t_2 \frown \alpha.v', \beta.w') && (9.26)
\end{aligned}
$$

Again, we only consider states which are reachable by a sequence of terminated service executions (Lines 9.20 and 9.21). Then, the states after execution of the service (Lines 9.22 and 9.23) have to be equivalent w.r.t. $\approx$ (Line 9.25). We do not directly require the resulting traces ($t_1 \frown \alpha.v \frown \beta.w$ and $t_2 \frown \alpha.v' \frown \beta.w'$) to be equivalent, but only the terminating messages (Line 9.26). For our simple example, this requirement implies $t_1 \frown \alpha.v \frown \beta.w \overset{T}{\sim} t_2 \frown \alpha.v' \frown \beta.w'$, but is simpler to analyze, as we will see in the remainder of this chapter.

Analogue to Theorem 3.3, if all services provided by the privacy store component are trace-non-interferent, then the component itself is trace-non-interferent.

Theorem 9.1. *If all services provided by* `PrivacyStore` *are service trace non-interferent w.r.t.* $\approx$ *and* $\overset{T}{\sim}$ *(Definition 9.9), then* `Privacy Store` *is trace non-interferent w.r.t.* $\overset{T}{\sim}$ *(Definition 9.7).*

The proof for Theorem 9.1 is simpler than the proof for Theorem 3.3 because we only have to show that it is valid for the concrete privacy store component with the concrete information flow specification $\overset{T}{\sim}$.

Proof. We prove inductively for two trace strategies $\omega_1 \overset{T}{\sim} \omega_2$ that for any trace t_1 which can be produced by *PrivacyStore* under an ω_1, there exists an equivalent trace t_2 which can be produced under ω_2. The induction is over the number n of terminated service calls in t_1.

Induction Hypothesis: For a trace t_1 with n terminated service calls, ending with a termination event, such that $\omega_1 \models \mathit{PrivacyStore} \xrightarrow{t_1} \langle \sigma_1; \mathit{SKIP}\rangle$, there exists t_2 such that $t_1 \overset{T}{\sim} t_2$ and $\omega_2 \models \mathit{PrivacyStore} \xrightarrow{t_2} \langle \sigma_2; \mathit{SKIP} \rangle$ and $\sigma_1 \approx \sigma_2$.

Induction start: $n = 0$: This case is trivially true, since $t_1 = t_2 = \langle\rangle$ and $\sigma_1 = \sigma_2 = \sigma_0$, i.e. the initial state of the *PrivacyStore* component.

Induction step: $n + 1$. Let $t_1 = t_1' \frown \mathit{Ini}(\mathit{serv})?v_1 \frown \mathit{Fin}(\mathit{serv})!w_1$, with $\omega_1 \models \mathit{PrivacyStore} \xrightarrow{t_1} \langle \sigma_1; \mathit{SKIP} \rangle$ and $\omega_1 \models \mathit{PrivacyStore} \xrightarrow{t_1'} \langle \sigma_1'; \mathit{SKIP} \rangle$. By the induction hypothesis, we know there exists a trace t_2' with $\omega_2 \models \mathit{PrivacyStore} \xrightarrow{t_2'} \langle \sigma_2'; \mathit{SKIP} \rangle$ with $t_1' \overset{T}{\sim} t_2'$ and $\sigma_1' \approx \sigma_2'$. Since t_1 can be communicated under ω_1 and $\mathit{Ini}(\mathit{serv})?v_1$ is an input, we know that $\mathit{Ini}(\mathit{serv})?v_1 \in \omega_1(t_1')$.

We make a case distinction over the visibility of $\mathit{Ini}(\mathit{serv})?v_1$.

Case 1: $(t_1, \mathit{Ini}(\mathit{serv})?v_1) \overset{T}{\sim} \square$. By definition of $\overset{T}{\sim}$, we also know that $(t_1 \frown \mathit{Ini}(\mathit{serv})?v_1, \mathit{Fin}(\mathit{serv})!w_1) \overset{T}{\sim} \square$. By the induction hypothesis, we know there exists a trace $t_2 \overset{T}{\sim} t_1'$, namely $t_2 = t_2'$ and therefore $t_2 \overset{T}{\sim} t_1$ which can be produced under ω_2. Further, since by assumption all services provided by *PrivacyStore* are service trace non-interferent (Definition 9.9), and therefore trace visibility preserving (Definition 9.8), we know that $\sigma_1 \approx \sigma_1'$ and therefore $\sigma_1 \approx \sigma_2'$.

Case 2: $\neg((t_1, \mathit{Ini}(\mathit{serv})?v_1) \overset{T}{\sim} \square)$. Since $\omega_1 \overset{T}{\sim} \omega_2$, there exists (by Definition 9.6) $\alpha.v \in \omega_2(t_1')$ with $(t_1', \mathit{Ini}(\mathit{serv})?v_1) \overset{T}{\sim} (t_1', \alpha.v)$ and by definition of trace strategies (Definition 9.5) and since $t_1' \overset{T}{\sim} t_2'$, we know there exists $\beta.w \in \omega_2(t_2')$ with $(t_1', \alpha.v) \overset{T}{\sim} (t_2', \beta.w)$ and therefore $(t_1', \mathit{Ini}(\mathit{serv})?1) \overset{T}{\sim} (t_2', \beta.w)$. By definition of $\overset{T}{\sim}$ (Definition 9.3), we know $\beta.w = \mathit{Ini}(\mathit{serv})?v_1$. Since *serv* terminates, we know there exists $\mathit{Fin}(\mathit{serv}).w_1$ and $\mathit{Fin}(\mathit{serv}).w_2$ with (due to Definition 9.9, and the induction hypothesis)
$\langle \mathit{handler}_{\mathit{serv}}; \sigma_1' \rangle \xrightarrow{\mathit{Ini}(\mathit{serv}).v_1 \frown \mathit{Fin}(\mathit{serv}).w_1} \langle \mathit{SKIP}; \sigma_1 \rangle$ and
$\langle \mathit{handler}_{\mathit{serv}}; \sigma_2' \rangle \xrightarrow{\mathit{Ini}(\mathit{serv}).v_2 \frown \mathit{Fin}(\mathit{serv}).w_2} \langle \mathit{SKIP}; \sigma_2 \rangle$.

Since by induction hypothesis $(t_1', \mathit{Ini}(\mathit{serv}).v_1) \overset{T}{\sim} (t_2', \mathit{Ini}(\mathit{ser}).v_2)$ and $\sigma_1' \approx \sigma_2'$ and since *serv* is service trace non-interferent (Definition 9.9), it also holds that
$(t_1' \frown \mathit{Ini}(\mathit{serv}).v_1, \mathit{Fin}(\mathit{serv}).w_1) \overset{T}{\sim} (t_2' \frown \mathit{Ini}(\mathit{serv}).v_2, \mathit{Fin}(\mathit{serv}).w_2)$ and $\sigma_1 \approx \sigma_2$.

Therefore the induction hypothesis for $n + 1$ holds. $\lhd$

9.4.3 State-based Service Non-interference

While Definition 9.9 is service-local, it still yields a proof obligation requiring reasoning about the entire trace of a DSC, since $\overset{T}{\sim}$ depends on the trace previously communicated. However, if a service provided by `PrivacyStore` is implemented in a secure way, all information necessary to decide whether some input provides high or low information has to be encoded in some way in the state of the DSC. It therefore has to be possible to express non-interference for services dependent on the prestate, and independent from the history of the DSC.

Definition 9.3 makes $\overset{T}{\sim}$ dependent on whether or not some observation shows a previously registered person with the functions *registered* and *guests*. We introduce a predicate *isRegistered* which encodes whether an observation shows a previously registered employee depending on the state. In a similar way, the predicate *isGuest* expresses whether an observation shows a previously observed guest. So the two predicates are a state-dependent formalization of the trace-dependent functions *registered* and *guests* from Subsection 9.3.1. We do not provide a concrete definition for the predicates yet, but formalize the connection between the state-dependent predicates and the trace-dependent functions as an invariant of all service executions of `PrivacyStore`.

Definition 9.10. The privacy store component preserves its trace-invariant, if there exists functions *isRegistered* : $\mathbb{S} \times \mathbb{D}$ and *isGuest* : $\mathbb{S} \times \mathbb{D}$ such that the following property holds:

$$\begin{aligned}
&\forall \sigma, v, t \cdot \langle body_{PrivacyStore}; \sigma_0\rangle \xrightarrow{t} \langle body_{PrivacyStore}; \sigma\rangle \implies \\
&\quad (isRegisterd(\sigma, v) \Leftrightarrow \\
&\qquad \exists i \in registered(t) \cdot t[i] = \beta.w \wedge \xi(w, v)) \wedge \\
&\quad (isGuest(\sigma, v)) \Leftrightarrow \\
&\qquad \exists i \in guests(t) \wedge t[i] = \beta.w \wedge \xi(w, v))
\end{aligned}$$

The privacy store component satisfies its invariant for *isRegistered* and *isGuest* if the following holds. Given a state σ after the termination of a service with trace t. *isRegistered* evaluates to true for all observation v, iff there exists a registered person whose last observation matches v according to the tracking predicate ξ. Similarly, *isGuest* evaluates to true, iff there exists a guest according to the trace which matches v according to ξ.

Given predicates *isRegistered* and *isGuest*, we define state-dependent message equivalence $\overset{T}{\approx} \subseteq (\mathbb{M} \times \mathbb{S}) \times (\mathbb{M} \times \mathbb{S})$ as follows:

Definition 9.11 (State-dependent Message Equivalence)**.**

$$\begin{aligned}
&(\sigma, \alpha.v) \overset{T}{\approx} \Box \Leftrightarrow \\
&\quad (\alpha = \mathit{Ini}(\mathit{updateObservation}) \wedge \mathit{isRegisterd}(\sigma, v))\vee \\
&\quad (\alpha = \mathit{Ini}(\mathit{registerCoworker}) \wedge \neg \mathit{isGuest}(\sigma, v))\vee \\
&\quad (\alpha \in \{\mathit{Fin}(\mathit{updateObservation}), \mathit{Fin}(\mathit{registerCoworker})\}\wedge \\
&\qquad\qquad \text{previous call was invisible})
\end{aligned}$$

Further, we define that a state σ_1 and a message $\alpha.v$ is state-dependent equivalent w.r.t. $\overset{T}{\approx}$ to σ_2 and $\beta.w$ as:

$$\begin{aligned}
&(\sigma_1, \alpha.v) \overset{T}{\approx} (\sigma_2, \beta.w) \Leftrightarrow \\
&\quad ((\sigma_1, \alpha.v) \overset{T}{\approx} \Box \wedge (\sigma_2, \beta.w) \overset{T}{\approx} \Box) \;\vee\; (\alpha = \beta \wedge v = w)
\end{aligned}$$

Note that state-dependence message equivalence is a state-dependent re-phrasing of the original trace-dependent specification from Definition 9.3.

Given the state-dependent equivalence relation for messages, we can formalize state-dependent visibility preserving services, a state-dependent notion of Definition 9.8.

Definition 9.12. A service *serv* provided by the privacy store component is state-dependent visibility preserving, w.r.t. $\overset{T}{\approx}$, iff

$$\forall \sigma_1, \sigma_2, \alpha.v, \beta.w, t \cdot \tag{9.27}$$

$$(\sigma_1, \alpha.v) \overset{T}{\approx} \Box\wedge \tag{9.28}$$

$$\langle \sigma_0; \mathit{body}_{\mathit{PrvicyStore}}\rangle \xrightarrow{t} \langle \sigma_1; \mathit{body}_{\mathit{PrvicyStore}}\rangle\wedge \tag{9.29}$$

$$\langle \mathit{handler}_{\mathit{serv}}; \sigma_1\rangle \xrightarrow{\alpha.v \frown \beta.w} \langle \mathit{SKIP}; \sigma_2\rangle \tag{9.30}$$

$$\Longrightarrow \tag{9.31}$$

$$\sigma_1 \approx \sigma_2 \tag{9.32}$$

We require the observation provided by the initial message of the service call to be invisible (Line 9.28) and the state σ_1 in which the service is called, to be reachable by a terminating sequence of service executions (Line 9.29). Then the prestate of the service call σ_1 and the poststate σ_2 have to be equivalent.

And finally, we define state-dependent non-interference for services, the state-dependent notion of Definition 9.9.

Definition 9.13. A service *serv* is service state-dependent non-interferent w.r.t. $\approx$, and $\overset{T}{\approx}$, iff *serv* is state-dependent visibility preserving and

$$\forall \sigma_1, \sigma_2, \sigma_1', \sigma_2', t_1, t_2 \cdot \tag{9.33}$$
$$\sigma_1 \approx \sigma_2 \wedge \tag{9.34}$$
$$(\sigma_1, \alpha.v) \overset{T}{\approx} (\sigma_2, \alpha.v') \wedge \tag{9.35}$$
$$\langle \mathit{handler}_{\mathit{serv}}; \sigma_1 \rangle \xrightarrow{\alpha.v \frown \beta.w} \langle \mathit{SKIP}; \sigma_1' \rangle \wedge \tag{9.36}$$
$$\langle \mathit{handler}_{\mathit{serv}}; \sigma_2 \rangle \xrightarrow{\alpha.v' \frown \beta.w'} \langle \mathit{SKIP}; \sigma_2' \rangle \wedge \tag{9.37}$$
$$\Longrightarrow \tag{9.38}$$
$$\sigma_1' \approx \sigma_2' \wedge (\sigma_1', \beta.w) \overset{T}{\approx} (\sigma_2', \beta.w') \tag{9.39}$$

We can verify for all services provided by the privacy store component that they are state-dependent visibility preserving w.r.t. $\overset{T}{\approx}$ and $\approx$ in order to show that the service is non-interferent w.r.t. $\overset{T}{\sim}$ and $\approx$.

Theorem 9.2. *If a service provided by the privacy store component is non-interferent w.r.t.* $\overset{T}{\approx}$ *and* $\approx$ *according to Definition 9.13, and if all services preserve the invariant defined in Definition 9.10, then the service is also non-interferent w.r.t.* $\overset{T}{\sim}$ *and* $\approx$ *according to Definition 9.9.*

Proof. Assume a service *serv*, provided by `PrivacyStore` is non-interferent w.r.t. Definition 9.13.

First, we show that *serv* is visibility preserving. Let $\sigma_1, \sigma_2, \alpha.v, \beta.w, t$ such that $(t, \alpha.v) \overset{T}{\sim} \Box$, $\langle \mathit{body}_{\mathit{PrivacyStore}}; \sigma_0 \rangle \xrightarrow{t} \langle \mathit{body}_{\mathit{PrivacyStore}}; \sigma_1 \rangle$, and $\langle \mathit{handler}_{\mathit{serv}}; \sigma_1 \rangle \xrightarrow{\alpha.v \frown \beta.w} \langle \mathit{SKIP}; \sigma_2 \rangle$. By definition of $\overset{T}{\sim}$ (Definition 9.3), we know $\exists i \in \mathit{registered}(t) \cdot t[i] = \gamma.x \wedge \xi(x, v)$ and by definition of *isRegistered* (Definition 9.10) and $\overset{T}{\approx}$ (Definition 9.11), we get $(\sigma_1, \alpha.v) \overset{T}{\approx} \Box$. Since *serv* is visibility preserving according to Definition 9.12, we get $\sigma_1 \approx \sigma_2$ which proves *serv* is visibility-preserving according to Definition 9.8.

Now, let $\sigma_1 \approx \sigma_2$, $t_1 \overset{T}{\sim} t_2$, $(t_1, \alpha.v) \overset{T}{\sim} (t_2, \alpha.v')$,
$\langle \mathit{body}_{\mathit{PrivacyStore}}; \sigma_0 \rangle \xrightarrow{t_1} \langle \mathit{body}_{\mathit{PrivacyStore}}; \sigma_1 \rangle$,
$\langle \mathit{body}_{\mathit{PrivacyStore}}; \sigma_0 \rangle \xrightarrow{t_2} \langle \mathit{body}_{\mathit{PrivacyStore}}; \sigma_2 \rangle$,
$\langle \mathit{handler}_{\mathit{serv}}; \sigma_1 \rangle \xrightarrow{\alpha.v \frown \beta.w} \langle \mathit{SKIP}; \sigma_1' \rangle$, and
$\langle \mathit{handler}_{\mathit{serv}}; \sigma_2 \rangle \xrightarrow{\alpha.v' \frown \beta.w'} \langle \mathit{SKIP}; \sigma_2' \rangle$.

If $(t_1, \alpha.v) \overset{T}{\sim} \Box$ the proof is according to the prove for visibility preserving above, extended by the definition of invisibility of termination events of invisible calls.

So, by definition of $\overset{T}{\sim}$ (Definition 9.3), we get $v = v'$, which gives us with definition of $\overset{T}{\approx}$ (Definition 9.11) $(\sigma_1, \alpha.v) \overset{T}{\approx} (\sigma_2, \alpha.v')$. Since *serv* is non-interferent w.r.t. Definition 9.13, we know $\sigma_1' \approx \sigma_2'$ and $(\sigma_1', \beta.w) \overset{T}{\approx} (\sigma_2', \beta.w')$ which again gives us according to (Definition 9.11) $w = w'$, and therefore, since the call is visible and by Definition 9.3, $(t_1 \frown \alpha.v, \beta.w) \overset{T}{\sim} (t_2 \frown \alpha.v', \beta.w')$, which shows Definition 9.9. $\triangleleft$

As a result, if we can show that all services provided by the privacy store component are state-dependent non-interferent, then the privacy store component is trace-non-interferent.

Corollary 9.3. *If there exists a state-dependent equivalence relation* $\overset{T}{\approx}$ *for messages and a relation* $\approx$ *for states, such that all services provided by* `PrivacyStore` *are state-dependent non-interferent w.r.t.* $\approx, \overset{T}{\approx}$ *and all services provided by the privacy component preserve the invariant in Definition 9.10, then* `PrivacyStore` *is trace-non-interferent w.r.t.* $\overset{T}{\sim}$.

Proof. Follows directly from Theorem 9.2 and Theorem 9.1. $\triangleleft$

We now gained a set service-local properties which are only dependent on the input to a service and the state of the component, which allow us to verify non-interference for the component using an off-the-shelf program verification tool. We can provide a JavaDL formalization for the proof obligations resulting from state-dependent non-interference (Definition 9.13) and the proof obligation expressing for a service that it preserves the invariant in Definition 9.10. In the following section, we present the JavaDL proof obligations for the privacy store component for a concrete implementation of the component and verify that it is non-interferent.

9.5 Implementation and Verification

In this section, we present a concrete implementation of the privacy store component, the JML specifications expressing the non-interference property from Definition 9.13, and the JML specifications expressing that the services preserve the invariant from Definition 9.10. We limit the presentation here to the most central methods and simplified specifications for the example. The full implementation of the privacy store together with the full specifications for the services and statistics on specification and verification can be found in Appendix A.3. The full specification for all methods can be found online with the supplemental material online[1].

We verify that the implementation of the services satisfies each of these specification using KeY, version 2.7. The tool can also be found online with

[1] `https://formal.iti.kit.edu/~greiner/niframework/`

```
 public final class PrivacyStore {
  private static final int NUM_FEATURES = 5;
  private static final int POS_X = 0;
  private static final int POS_Y = 1;
  private static final int FEAT1 = 2;
  private static final int FEAT2 = 3;
  private static final int FEAT3 = 4;
  private static final int BLURX = 4;
  private static final int BLURY = 4;

  private int[][] guestVectors;
  private int[][] coworkerVectors;

  public void updateObservation(int[] observation) {...}
  public void registerCoworker(int[] observation) {...}
  public int[] getGuest(int pos) {...}
}
```

Figure 9.4: Code skeleton of the implementation of the `PrivacyStore` component

the supplemental material. An extensive discussion of KeY is provided by Ahrendt et al. [2016] and more details on non-interference specifications and verification in KeY are presented by Scheben [2014].

The main reason for using KeY instead of our extension discussed in Chapter 8 is that it supports all features necessary for the proofs in this section, while supporting several optimizations for the proofs which make KeY more scaleable compared to our extension. Due of the size of the resulting proofs, the better scaleability showed itself to be necessary for our task.

9.5.1 Implementation

We implement the privacy store component as a Java class and Figure 9.4 shows the code skeleton of the implementation. The class declares the three public methods `updateObservation`, `registerCoworker`, and `getGuest`. The first two methods receive an `int` array as a parameter encoding the observation provided by the camera components and do not provide a return value. The third method takes an `int` value as a parameter, which is provided by the observation desk component, and returns an `int` array encoding the features of a guest under surveillance.

The privacy store manages observations with five features. The first two features are the position of the person described by the observation, the third

```
/*@ public normal_behaviour
  @ requires observation.length == toCompare.length &&
  @          observation.length == NUM_FEATURES;
  @ ensures \result ==
  @  (observation[POS_X] - toCompare[POS_X] < BLURX &&
  @   toCompare[POS_X] - observation[POS_X] < BLURX &&
  @   observation[POS_Y] - toCompare[POS_Y] < BLURY &&
  @   toCompare[POS_Y] - observation[POS_Y] < BLURY &&
  @   observation[FEAT1] == toCompare[FEAT1] &&
  @   observation[FEAT2] == toCompare[FEAT2] &&
  @   observation[FEAT3] == toCompare[FEAT3]);
  @ model public strictly_pure boolean sameObs(
  @         int[] observation, int[] toCompare) {...}*/
```

Figure 9.5: JML specification for the tracking predicate ξ

to fifth feature are underspecified here, but they may describe the height, hair color and similar information extracted by the camera. For each of these features, we introduce a constant which can be used by the program to access the respective positions in arrays. Additionally, we have to allow people under surveillance to move in space, however we assume a person to move at most four units on the x-axis and y-axis between two consecutive observation. This movement tolerance is again represented in the program with the constant values `BLURX` and `BLURY`.

The fields `guestVectors` and `coworkerVectors` are used by the program to store the last observation of registered and unregistered people. Whenever an observation is provided by a camera, these arrays are updated according to the tracking property, and if requested, the return values provided to the operator's desk are read from these vectors.

We use the model method `sameObs(int[], int[])`, i.e. a specification-only method, to define the tracking predicate ξ. The JML formalization of the predicate is shown in Figure 9.5.

The contract of `sameObs` formalizes the tracking predicate ξ. The precondition (following the `requires` keyword) requires the two parameters to be of the proper length, i.e. containing exactly `NUM_FEATURES` features. The postcondition (following the `ensures` keyword) of `sameObs`, i.e. ξ, states that it evaluates to true if the difference in location of the two observations is within the movement tolerance, and all other features are equal in both observations. Note that `sameObs` does not depend on the actual state, but only on the values of the two arrays provided as parameters.

9.5.2 Specifications for the Trace-invariant

To verify that all services preserve the invariant as defined in Definition 9.10 we need a formalization of the predicates *isRegistered* and *isGuest*. Given the arrays representing the guests and coworkers under surveillance, we can phrase the invariant from a state point of view.

Theorem 9.4 (Trace Invariant). `PrivacyStore` *satisfies its trace invariant, iff for all states* σ *and traces* t*, such that* $\langle body_{PrivacyStore}; \sigma_0\rangle \xrightarrow{t} \langle body_{PrivacyStore}; \sigma_1\rangle$*, and* $\alpha \in \{Ini(updateObservation), Ini(registerCoworker)\}$ *and* $v \in \mathbb{D}$ *it holds*

$$
\begin{aligned}
&(\exists i \in guests(t) \cdot t[i] = \alpha.v) \Leftrightarrow && (9.40)\\
&\quad(\exists 0 \leq j < \sigma(\mathtt{guestVectors.length}) \cdot && (9.41)\\
&\qquad\qquad \sigma(\mathtt{guestVectors[j]}) = v \vee && (9.42)\\
&(\exists i \in registered(t) \cdot t[i] = \alpha.v) \Leftrightarrow && (9.43)\\
&\quad(\exists 0 \leq k < \sigma(\mathtt{coworkerVectors.length}) \cdot && (9.44)\\
&\qquad\qquad \sigma(\mathtt{coworkerVectors[k]}) = v) && (9.45)
\end{aligned}
$$

Proof. Follows directly from Definition 9.10. ◁

The first equivalence (Line 9.40) states that if there exists an observation v which is among the last observations of guests of trace t, then there exists an entry in `guestVectors` with the same values as v. The second equivalence (Line 9.43) states that if there exists an observation v which is the last observation of a registered person in trace t, then there exists an entry in `coworkerVectors` with the same values as v. The invariant ensures that `guestVectors` and `coworkerVectors` actually record the last observation registered and unregistered people.

From Theorem 9.4, the definition of the predicates *isRegistered* and *isGuest* (introduced in Definition 9.10) directly follows.

$$
\begin{aligned}
isGuest(\sigma, v) &:= \exists\ 0 \leq i < \sigma(\mathtt{guestVectors.length}) \cdot\\
&\qquad \sigma(sameObs(\mathtt{guestVectors[i]}, v))\\
isRegistered(\sigma, v) &:= \exists\ 0 \leq i < \sigma(\mathtt{coworkerVectors.length}) \cdot\\
&\qquad \sigma(sameObs(\mathtt{coworkerVectors[i]}, v))
\end{aligned}
$$

We specify for each service provided by `PrivacyStore` a functional contract expressing that the service satisfies the trace invariant as stated in Theorem 9.4.

The contract for `getGuest` is rather simple. Neither the initial nor the terminating message for this service changes the values of *registered* or *guests*, so we ensure by contract that the service does not change the component's state, and thus it does not change the evaluation of *isGuest* nor *isRegistered*. The `modifies` clause of the following contract expresses this behavior.

```
/*@ public normal_behavior
  @ requires true;
  @ ensures true;
  @ modifies \nothing; */
public /*@ nullable */ int[] getGuest(int pos) {...}
```

The contract for service `updateObservation` is more complicated. Due to its verbosity, we sketch here a strong simplification using a more intuitive style of specification than a formal JML contract. The full contract, as verified, can be found in the appendix.

```
   /*@ public normal_behaviour
     @requires
     @ exists i: sameObs(observation, guestVectors[i]) ==>
     @  !exists i!=j:sameObs(observation,guestVectors[j])&&
 5   @  !exists k: sameObs(observation, guestVectors[k]) &&
     @  (... analogue for coworkerVectors ...);
     @ensures
     @ (exists i: sameObs(observation,guestVectors[i])@pre
     @  ==>
10   @  (values(guestVectors[i]) == values(observation) &&
     @   other values of guestVectors stay unchanged &&
     @   coworkerVectors stays unchanged)) &&
     @ (exists i: sameObs(observation,
     @                       coworkerVectors[i])@pre
15   @  ==>
     @  (values(coworkerVectors[i])==values(observation)&&
     @   other values of coworkerVectors stay unchanged &&
     @   guestVectors stays unchanged)) &&
     @ ((!exists i: sameObs(observation,
20   @                       guestVectors[i])@pre) &&
     @   !(exists i: sameObs(observation,
     @                       coworkerVectors[i])@pre)
     @   ==>
     @   (observation values added to guestVectors &&
25   @    other values of guestVectors stay unchanged &&
     @    coworkerVectors stays unchanged));
     *\
     public void updateObservation(int[] observation) {...}
```

The precondition (from Line 2) expresses perfect tracking (Definition 9.2): if the provided observation matches one last observation, it only matches this one last observation.

For the postcondition, we consider three different cases. If the observation matches a guest (Line 8), then the respective entry of `guestVectors` is updated to the values of the observations. All other entries of `guestVectors` and all entries of `coworkerVectors` are left unchanged. This is consistent with updating *guests* according to the changed trace, while leaving the result of *registered* unchanged for the new trace.

In the second case, when the observation matches a registered person (Line 13), the respective entry of `coworkerVectors` is updated. Again, all other entries of `coworkerVectors` and all entries of `guestVectors` are left unchanged. This is consistent with updating *registered* w.r.t. the new trace, while leaving *guests* unchanged.

The third case covers the situation, when the observation is a new person, i.e. it does not show a guest nor a registered person (Line 19). In this case the observation is added to `guestVectors`, while all entries of `guestVectors` and `coworkerVectors` are unchanged. Again, this is consistent with adding a new entry to *guests* while *registered* remains unchanged.

As a result, the contract ensures, if the trace invariant as formulated in Theorem 9.4 holds before execution of `updateObservation`, then it also holds after execution.

In a similar way, we provide the trace invariant contract for the service `registerCoworker`. Again, the full formal JML contract is presented in the appendix, we limit the presentation here to an intuitive, simplified version.

```
/*@ public normal_behaviour
  @requires
  @ exists i: sameObs(observation,guestVectors[i]) ==>
  @  !exists i!=j:sameObs(observation,guestVectors[j])&&
  @  !exists k: sameObs(observation,guestVectors[k]) &&
  @  (... analogue for coworkerVectors ...);
  @ensures
  @ (exists i: sameObs(observation,guestVectors[i])@pre
  @  ==>
  @    (i-th entry of guestVectors is removed &&
  @     other entries of guestVectors stay unchanged &&
  @     coworkerVectors stay unchanged &&
  @     observation is added to coworkerVectors)) &&
  @ (exists i: sameObs(observation,
  @                coworkerVectors[i])@pre
  @  ==>
  @   (values(coworkerVectors[i])==values(observation)&&
  @    other values of coworkerVectors stay unchanged &&
  @    guestVectors stays unchanged)) &&
  @ ((!exists i: sameObs(observation,
  @              guestVectors[i])@pre) &&
  @   !(exists i: sameObs(observation,
  @              coworkerVectors[i])@pre)
```

```
     @   ==>
     @     (guestVectors stays unchanged &&
     @      coworkerVectors stay unchanged &&
27   @      observation is added to coworkerVectors));
   */
     public void registerCoworker(int[] observation) {...}
```

The precondition (Line 2) expresses perfect tracking, as in the contract for `updateObservation`.

For the postcondition, we again distinguish three cases. If the observation shows a guest (Line 8), so a guest registers as a coworker, The respective entry is removed from the guests, while the other last observations of the guests are left unchanged. Also, all entries of the registered persons are left unchanged, while `observation` is added as new registered person.

In the second case, the observation maps to a registered person (Line 14), so basically the person re-registers. In this case, the observation of this person is updated, while all other last observations of registered person's stays unchanged, as do the last observations of the guests.

In the third case, the observation does not match a guest nor a registered person (Line 20), so a previously unknown person registers. In this case, the last observations of the guests are left unchanged while the observation is added as a new entry to the `coworkerVectors`.

Again, if the trace invariant as formulated in Theorem 9.4 holds in the prestate, it also holds in the poststate of the service `registerGuest`.

The three contracts for the services `getGuest`, `updateObservation`, and `registerCoworker` together ensure that `PrivacStore` satisfies the invariant stated in Definition 9.10, i.e. *isGuest* and *isRegisterd* are a state-based notion for the functions *guests* and *registered* which we can use in pre- and postconditions for non-interference contracts.

9.5.3 Service Non-interference Specification

In order to show non-interference for each service, we have to show that there exists an equivalence relation over states $\approx$ such that all services provided by the privacy store component are non-interferent w.r.t. $(\overset{T}{\approx}, \approx)$ according to Definitions 9.11 and 9.13. We provide the definition of $\approx$ indirectly by specifying the low part of the state as a model field defining a sequence of expressions over the state:

```
/*@ public instance model \seq lowvalues; */
/*@ public represents lowvalues <-
  @  \seq(guestVectors.length, guestVectors,
  @     (\seq_def int j; 0; guestVectors.length;
  @     (\seq_def int k; 0; NUM_FEATURES;
  @                             guestVectors[j][k]))); */
```

`lowvalues` is a sequence listing the length of `guestVectors` as well as all entries of the feature vectors held in it. We use in the following the equivalence relation $\approx$ to be defined as two states being equivalent, if `lowvalues` evaluates to the same lists in both states: $\sigma_1 \approx \sigma_2 \Leftrightarrow \sigma_1(\texttt{lowvalues}) = \sigma_2(\texttt{lowvalues})$. This way, we specify that the low part of the state is the content of `guestVectors`.

Now, we specify for each service one contract expressing that it is visibility preserving and after that for each service one contract expressing non-interference.

Visibility Preserving Services The service `getGuest` can never be called with an invisible call event according to Definition 9.13, and therefore the service is trivially visibility preserving.

The service `updateObservation` takes as a parameter a new observation, i.e. an array of features provided by the camera system. The following contract expresses that the service is visibility preserving in a simplified, intuitive version.

```
  /*@public normal_behavior
    @requires
    @ !(exists i: sameObs(observation,guestVectors[i]))&&
4   @  exists i:sameObs(observation,coworkerVectors[i]);
    @ensures lowvalues == lowvalues@pre;
    @*/
    public void updateObservation(int[] observation) {...}
```

In the precondition (Line 3), we limit the consideration of the contract to the case when the observation shows a registered person (and due to perfect tracking, not a guest). According to the definition of message invisibility in Definition 9.11 in combination with the invariant from Definition 9.10 this states that the service is called with an invisible message. The postcondition (Line 5) states that the low part of the state after execution is the same as the low part of the state before execution, i.e. pre- and poststate are lowequivalent according to $\approx$

The contract for `registerCoworker` follows a similar pattern. The precondition for `registerCoworker` only requires the observation not to be registered as a guest, which would be observable by the respective entry being removed from the feature vectors the operator's desk can observe. Otherwise, the contract follows the same reasoning as for `updateObservation`.

```
/*@public normal_behavior
  @requires
  @ !(exists i: sameObs(observation,guestVectors[i]));
  @ensures lowvalues == lowvalues@pre;
  @*/
  public void registerCoworker(int[] observation) {...}
```

The message calling `registerCoworker` is invisible according to Definition 9.11, if the observation does not show a guest. This is formalized in the precondition (Line 3). The postcondition (Line 4) states the the pre- and tho poststate are equivalent according to $\approx$.

Service Non-Interference For specifying non-interference of the services, we use the specification mechanism introduced in Scheben [2014]. Non-interference is specified in a contract by a `determines` clause. `@determines listAfter \by listBefore` as part of a contract states that the expressions in `listAfter` are at most influenced by the expressions in `listBefore`. A service is non-interferent if for two executions of a service started in states where `listBefore` evaluate to the same values, they terminate in states where `listAfter` evaluate to the same values. This specification allows declassification in the sense that the low information before and after service execution may differ.

In particular, this specification allows, in contrast to our extension JML extension in Chapter 8, to specify equivalence of state depending on parameters and the return value of a methods, or in our a case a service. We have to relate, however, parameters, return values and the state in order to formalize $\overset{T}{\approx}$ in JML.

For the service `getGuest`, the parameter `pos` is low, as is the part of the state described by `lowvalues`. The service has to guarantee that the values of the returned feature vector is low and that `lowvalues` still only contains low information. The resulting JML information flow contract is straight forward:

The service `getGuest` has to ensure that the poststates of two executions of the service are equivalent w.r.t. $\approx$, if they are equivalent in the prestates. Additionally, the return value has to be equivalent in both executions, given the parameter `pos` is equal. We have to make a case distinction in the specification for the case that the parameter does not refer to a legal element stored in the privacy store. The service returns `null` in case the parameter is out of bounds, otherwise the entries of the returned array is low. The following listing is the simplified JML representation of this non-interference property.

```
/*@ public normal_behaviour
  @ requires true;
  @ determines content of \result, lowvalues
  @ \by lowvalues, pos;
 */
public /*@ nullable */ int[] getGuest(int pos) {...}
```

We do not consider in this simplified contract the case that the parameter is out of bounds, in which case, the return value of the service is `null`. The full contract in the appendix covers this case.

The following listing shows the non-interference contract for the service `updateObservation`.

```
  /*@ public normal_behaviour
    @ requires
    @  exists i: sameObs(observation,guestVectors[i])||
4   @  !(exists i: sameObs(observation,
    @                                coworkerVectors[i]));
    @ determines lowvalues
    @ \by lowvalues,
    @  exists i: sameObs(observation,coworkerVectors[i]),
9   @  exists i: sameObs(observation, guestVectors[i]),
    @  values(observation);
    */
  public void updateObservation(int[] observation) {...}
```

The precondition limits the validity of the contract to the case when the initial message is visible, i.e. the observation shows a guest or it does not show a registered person.

In this case, the service has to ensure that the low part of the state, expressed by `lowvalues`, only contains low information after execution (Line 7).

We know that the low part of the state only contains low information in the prestate (Line 8). Further, the information whether or not the observation shows a registered person is low (Line 8), as is the information whether or not the observation shows a guest (Line 9). And, of course, the content of the observation itself is low (Line 10).

In a similar way, we specify the service `registerCoworker`.

```
  /*@ public normal_behavior
    @ requires
3   @  !(exists i: sameObs(observation,guestVectors[i]))||
    @  !(exists i: sameObs(observation,
    @                                   coworkerVectors[i]));
    @  determines lowvalues
    @ \by lowvalues,
8   @   values(observation),
    @   (exists i: sameObs(observation,guestVectors[j]));
    @*/
  public void registerCoworker(int[] observation) {...}
```

The precondition (Line 2) again limits the validity to the case when the initial message is low. This is the case, when the observation does not show a guest. Additionally, we need for technical reasons to express perfect tracking in the precondition, namely that the observation either does not show a guest or it does not show a registered person (ξ can not map to both). As a result, we show non-interference for all observations which are consistent with perfect tracking.

We have to show that the service guarantees that the low part of the state only contains low information (Line 6). We can assume that the low part of the state in the prestate only contains low information (Line 7). Further, the information in the observation is low (Line 8), as is the information whether or not the observation shows a guest (Line 9).

9.5.4 Verification

We verified all specifications as described in the previous subsection using the KeY tool in version 2.7. All specifications, including helper specifications, e.g. loop invariants, block contracts, and helper methods, as well as the tool together with the implementation can be found at online[2]. We present here some findings we made during the verification of the privacy store component. Detailed numbers for the specifications can be found in Section B.3 and for the verification in Section B.4 in the Appendix.

The implementation of `PrivacyStore` consists of 13 methods, including services and helper methods. For easier verification, we provided separate specifications for proving that the services preserve the trace invariant, that services are visibility preserving and that services are non-interferent. The implementation consists of about 100 lines of program code. For invariant preservation, we required 262 lines of specification, for visibility preserving 167 lines and for non-interference 247 lines.

In general the 1:2 ratio between lines of code and lines of specification is a common ratio for non-trivial specifications with JML, however it stands out that the functional specification for invariant preservation was the largest. The main reason for this is that for our compact implementation, we basically required a full specification of the functionality of each method plus a representation of the trace properties for each service, while for non-interference and visibility properties we could omit specifications for parts not concerning the low part of the state.

For verification, we required a total of about 700,000 proof steps for invariant preserving, 350,000 for visibility preserving and 1,400,000 for non-interference. Since the specification for invariant and visibility preservation are functional properties, while non-interference is a relational program property, i.e. two symbolic executions are compared, we expected the result that non-interference proofs are more complicated. This can also be seen by comparing necessary manual interactions during verification (1049 for

[2]`https://formal.iti.kit.edu/~greiner/niframework/`

invariant preservation, 341 for visibility preservation, and 2973 for non-interference proofs), and run-time of KeY during auto mode (420 minutes for invariant preservation, 59 minutes for visibility preservation and 550 minutes for non-interference proofs).

The two main reason for manual interaction were, for one, case splits on rather big concatenations necessary for class invariants, as well as instantiations for quantifier. While KeY is in general quite good in finding correct quantifier instantiations in easier cases, it typically has problems with finding the required instantiations for nested quantifier. Since the main functionality in our example is implemented using two-dimensional arrays, we frequently required nested quantifier for the class invariants. Especially for non-interference proofs, we required frequently instantiations of corresponding quantifier for the two runs of the program as well as manual interaction for unrolling sequences used for specification.

One additional problem with the non-interference proofs in our case was, that for `registerCoworker` in some cases elements are removed from two-dimensional arrays, where the contents of the arrays are low. In this case, indices of equivalent contents of `guestVectors` are shifted by one, however only for those elements with a higher index than the removed index. Manually instantiating these indices in nested quantifier is error prone and lead to additional time effort during verification.

In summary, the entire verification effort takes about 2 weeks of full-time work, not counting additional work for finding helper specifications which are sufficiently precise for a verification.

9.6 Related Work

In the work as presented in this chapter, we had to unwind the overall non-interference property, which is defined over trace-equivalence, into smaller specifications which guarantee some form of well-behavior of each individual service. Of course, it would be easier to directly express the class invariant with the history of communicated events. How the history can be constructed in general in a deductive program verification approach was introduced in Chapter 8. However, in order to gain an overall proof for our security property, we would require some form of temporal logic in the deductive verification framework. In Beckert and Bruns [2013], the authors define Dynamic Trace Logic, a combination of dynamic logic for program verification with a temporal logic. Traces in their work are sequences of intermediate states of a program, not input or output events. However, the temporal extension of the logic presented in their work should in general also be extendable for traces of events.

The specification of our non-interference property defines inputs to be high or low depending on the history of communicated events. A very natural way to express such properties is linear time logic (LTL). Balliu et al. [2011] studies non-interference using epistemic temporal logic and Clarkson et al. [2014] for LTL formulas, for expressing non-interference of a system by comparing traces which can be communicated by the system. Both approaches allow declassification of information for a program implemented in a simple while language. However, both approaches do not explicitly support intermediate input events (they do support output events), therefore all information has to be encoded as part of the initial state. Our non-interference property therefore can not be easily expressed in their approaches. Further, in both approaches, the entire system has to be analyzed, a modular analysis of partial programs, e.g. services, is not considered in both approaches.

The approach closest to our solution is a line of work presented in Kanav et al. [2014]; Bauereiß et al. [2016]. The authors define Bounded-Deducability (BD) security, a notion of bounded non-interference. A relation B over the secrets of two traces expresses what parts of the secrets in a trace an attacker may learn. A system is non-interferent w.r.t. a bound B, if for a trace of events tr and a sequence of secrets sl', the secrets in tr and sl' are in relation according to B, and there exists a trace tr' such that the secrets in tr' are sl' and tr and tr' contain the same public events.

In order to verify that a system is BD secure, the authors propose an unwinding theorem, which relates intermediate states and intermediate secrets of two runs of a system. Our approach basically follows this unwinding idea (although, we did not plan this). We ensure equivalent intermediate states of our privacy store by showing non-interference for the provided services. We also ensure that our state correctly stores the relevant secrets contained in the trace by verifying the trace invariant. For verifying case studies for BD security, the authors use Isabell/HOL (Nipkow et al. [2002]; Nipkow and Klein [2014]).

9.7 Conclusion

In this chapter we showed on the example of a privacy preserving video surveillance system how our framework from Part I can be extended to support non-interference specifications where the secrecy of a message depends on the history of events. We could show that the non-interference property can be expressed by extending our framework for this one example. However, gaining a program property which we can analyze with the theorem prover KeY, is laborious and takes a lot of effort even for this small example. Additionally, the verification itself takes huge effort and useful properties, like compositionality of non-interference, are lost.

For certain types of systems, trace-dependent secrecy of input and output information has become a very common security policy. For example social networks specify the content of a message secret depending on whether a person was previously added as a friend or if certain privacy settings were set (both actions a represented as messages in the trace of the overall system). Also in cloud based systems, it is very common to provide a functionality where a file or folder is labeled public by providing a special link which can be sent to friends or colleagues. It is easy to see that with the rise of relationship-based access control (e.g. see Fong et al. [2009]; Fong [2011]; Carminati et al. [2011]; Cheng et al. [2012]) these properties become more and more commonly used in practice.

It would be interesting to find out whether our framework can be extended to support relationship-based access control, at least with limitations, while keeping the resulting non-interference notion to be compositional. We do assume that limitations of the computational model to DSCs that we introduced in Chapter 2 helps to achieve this. The general assumption that there is no concurrent computation with shared heaps in a single component in an otherwise highly distributed system should be true for the biggest part of architectures for social networks and cloud storage systems.

10 Conclusion

10.1 Summary

We have presented a framework for non-interference in distributed component-based systems. Our framework serves as the theoretical basis for integrated consideration of non-interference properties during requirement elicitation, system design, implementation and quality assurance in development processes for component-based systems. We instantiated our theoretical framework for two concrete applications.

In the first part of this thesis, we presented a novel notion of non-interference for DSCs, our formalization of components. Specifications for non-interference properties are given as equivalence relations, which allows very precise what-declassification of information exchanged between a component and its environment. Further, we allow messages existence to be specified as sensitive information conditional on the content of the messages.

Our non-interference property for DSCs is based on an explicit environment, which is modeled using strategies. Since an attacker gained from a domain-motivated attacker analysis can be seen as a part of the environment, a mapping between a domain-motivated security requirement and a non-interference specification during the requirement elicitation phase is intuitive.

Our non-interference property is compositional, an important requirement to be useful in component-based systems. During system design, non-interference specifications for individual components can easily be derived from a system-wide non-interference specification or vice versa.

We further provide a novel non-interference notion for services, and we show that non-interference for services implies non-interference for DSCs. Service-local non-interference specifications provide a requirement for the developer of a service which has to be followed when implementing the functionality.

We introduced Dependency Cluster as building blocks for non-interference specifications on service-, component-, and system-level. Apart from the general relation between system-, component-, and service-non-interference as discussed in Chapter 3, Dependency Cluster support a bottom-up approach for developing secure systems. Since Dependency Cluster for services are independent from other services, components, and the environment, they are a powerful tool to support secure re-use of components in different environments and system-evolution, i.e. when parts of a system are re-implemented or optimized. Dependency cluster are in particular helpful during quality assurance, since every Dependency Cluster can be checked individually, using different approaches for quality assurance.

Pure top-down or pure bottom-up development processes are practically never used. In our framework the direction of development can be switched individually for every service, component or composition, and thus supports these development processes.

In the second part of this thesis we instantiate our framework and provide an extension for non-interference specifications in the graphical specification language Palladio. We show that theoretical results from the first part of the thesis hold in our new specification language by mapping elements in a Palladio model to elements in our framework. We further provide a novel deductive verification approach for Dependency Cluster specifications formalized in JML for components implemented in Java using the JavaEE framework. We show that our approach can be used to verify information flow security of Java beans and illustrate the relationship between the novel specification language and our framework.

Our work has been picked up by other researchers in the community. Greiner et al. [2017b] and Greiner et al. [2017a] present an automatic, but less precise program analysis technique for Dependency Cluster, based on the tool JOANA. Our graphical specification language for non-interference properties was applied for the specification of a cash register system (Greiner and Herda [2017a]) and in an extended version for a cloud storage system (Kramer et al. [2017]). Currently ongoing work develops a code-generation technique where Java code skeletons from Palladio models are generated, and enriched with non-interference specifications based on our work.

In the third part of this thesis, we discuss the limitation that our framework does not allow temporal declassification of information. Such properties become more and more interesting, as privacy properties, for example in social media platforms, are based on information flow depending on actions performed in the past. We hinted several ideas on how our framework may be extended to support temporal declassification, however we do not expect a respective solution to be straight forward.

10.2 Future Work

Our results leave several open problems, raise new questions and allow for novel approaches making use of our results.

Future work on the framework Currently, our framework is limited to synchronous service calls, where a service halts execution when calling another service until this service terminates. JavaEE supports so-called message-driven beans, where services can be called asynchronously, but asynchronous services must not provide a return value. We expect asynchronous service calls, where only an initial message is generated but no terminating message, to be relatively easily integrated into our framework.

Further, our framework does not allow concurrent execution of services within a component. At least with limitations, it should be possible to allow concurrent execution of non-interferent services, while still achieving non-interference for components. Well-researched rely/guarantee approaches ensure non-interference of concurrent threads with shared memory by ensuring that a thread (or service) provides a guarantee on its memory usage while assuming other threads to provide certain guarantees on theirs. We expect that rely/guarantee approaches to be a good starting point to identify non-interference properties for services such that non-interferent services result in non-interferent components.

In Chapter 9 we extensively discussed the limitation that our framework does not allow temporal declassification on the example of a smart surveillance system. It would be very interesting to see if our framework can be extended to allow temporal declassification, while still ensuring compositionality properties of the resulting non-interference notion and compositionality of non-interferent services. However, we expect this task to be non-trivial.

Finally, we presented our framework mainly from a perspective where non-interference is the basis for confidentiality properties of a system. It is often noted that non-interference is also the basis for integrity. It would be interesting to see if our framework can directly, or at least with small changes, be used to enforce integrity requirements for systems, components, and services.

Future work on instantiations of the framework We provided two concrete instantiations of our framework, one as a graphical specification language and the other as a program analysis technique. Others provided an additional program analysis technique based on program dependency graphs. We are confident that our framework can be used as the basis for other novel analysis techniques in other domains or based on other techniques.

For example, databases can be seen as DSCs according to our definition. When a database is only used via so-called stored procedures, i.e. predefined

SQL commands where some fields are parametrized, it should be possible to describe the effect of each query using relational algebraic data structures. Also, parameters can easily be specified as high or low, such that based on our framework, especially the results in Chapter 4, as powerful non-interference checker for databases can be implemented. We also expect our framework to be re-usable in embedded systems, for example in control units as used in the automotive systems. In this domain, integrity may be more important than privacy in order to ensure that safety-critical actions are not influenced by untrusted inputs, for example from an infotainment system. Our framework may be able to serve as a theoretical basis for the development of novel secure architectures or program analysis techniques for components.

Finally, we would like to point out that our framework, especially non-interference for components, can be the basis for novel security testing approaches. A common technique for security testing is fuzz-testing, where more or less random inputs to a system are generated in order to check whether the system under test crashed. This technique can be extended to allow direct non-interference testing. The main problem with automatic non-interference testing is that two equivalent runs have to be compared, i.e. for each input, another, equivalent input has to be generated. Non-interference for components is based on equivalence classes of messages, such that each equivalence class directly provides for an arbitrary input another equivalent input, which is gained by calculating the representative of the respective equivalence class. It is then relatively easy to check outputs for equivalence.

Part IV

Appendix

Running Example

A.1 Implementation

Implementation of the shop system used as a running example in Part I. The implementation is provided using the simple while language introduced in Chapter 2.

```
Component Cart {
  //State variables
  int product, prodprice, prodamount;
  int countbuy, countpay, countcheck;

  int buy(int prod, int price, int amount) {
    product := prod;
    prodprice := price;
    prodamount:= amount;
    countbuy := countbuy + 1;
    return 0;
  }

  (int, int, int) checkCart(int x) {
    countcheck := countcheck + 1;
    return (product, prodprice, prodamount);
  }

  int clearCart(int x) {
    product := 0;
    prodprice := 0;
    prodamount := 0;
    return 0;
  }

  int pay(int ccnr) {
    countpay := countpay + 1;
    registerSale(product, prodprice, prodamount, ccnr);
    return 0;
  }

  (int, int, int) getAllNums(int x) {
    return (countbuy, countpay, countcheck);
  }
}

Component Sales {
  //state variables
  int lastprod, lastprice, lastamount;
  int lastccnr;

  (int, int, int, int) lastSale(int x) {
    return (lastprod, lastprice, lastamount, lastccnr);
  }

  int registerSale(int prodId, price, amount, ccnr) {
    lastprod := prodId;
    lastprice := price;
    lastamount := amount;
    lastccnr := ccnr;
    //Alternative with declassification:
    //lastccnr := ccnr % 10 000;
    return 0;
  }
}

Component Controlling {
  // no state variables managed
  int numBuys(int x) {
    return (getAllNums(0)#1);
  }
  int numPays(int x) {
    return (getAllNums(0)#2);
  }
  int numChecks(int x) {
    return (getAllNums(0)#3);
  }
}
```

A.2 Specification

Definition of the equivalence relation $\sim$ specifying a domain-motivated security specification for the running example.

A.2.1 Billing Department

For the `Cart` component:

$$
\begin{aligned}
&\mathit{Ini}(\mathit{buy}).(\mathit{prod},\mathit{price},\mathit{am}) \sim \Box && \Leftrightarrow \mathit{false}\\
&\mathit{Ini}(\mathit{buy}).(\mathit{prod},\mathit{price},\mathit{am}) \sim \mathit{Ini}(\mathit{buy}).(\mathit{prod}',\mathit{price}',\mathit{am}')\\
&\qquad \Leftrightarrow \mathit{prod} = \mathit{prod}' \wedge \mathit{price} = \mathit{price}' \wedge \mathit{am} = \mathit{am}'\\
&\mathit{Fin}(\mathit{buy}).(r) \sim \Box && \Leftrightarrow \mathit{false}\\
&\mathit{Fin}(\mathit{buy}).(r) \sim \mathit{Fin}(\mathit{buy}).(r') && \Leftrightarrow \mathit{true}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Ini}(\mathit{checkCart}).(x) \sim \Box && \Leftrightarrow \mathit{true}\\
&\mathit{Ini}(\mathit{checkCart}).(x) \sim \mathit{Ini}(\mathit{checkCart}).(x') && \Leftrightarrow \mathit{true}\\
&\mathit{Fin}(\mathit{checkCart}).(r1,r2,r3) \sim \Box && \Leftrightarrow \mathit{true}\\
&\mathit{Fin}(\mathit{checkCart}).(r1,r2,r3) \sim \mathit{Fin}(\mathit{checkCart}).(r1',r2',r3') && \Leftrightarrow \mathit{true}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Ini}(\mathit{clearCart}).(x) \sim \Box && \Leftrightarrow \mathit{false}\\
&\mathit{Ini}(\mathit{clearCart}).(x) \sim \mathit{Ini}(\mathit{clearCart}).(x') && \Leftrightarrow \mathit{true}\\
&\mathit{Fin}(\mathit{clearCart}).(r) \sim \Box && \Leftrightarrow \mathit{false}\\
&\mathit{Fin}(\mathit{clearCart}).(r) \sim \mathit{Fin}(\mathit{clearCart}).(r') && \Leftrightarrow \mathit{true}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Ini}(\mathit{pay}).(\mathit{ccnr}) \sim \Box && \Leftrightarrow \mathit{false}\\
&\mathit{Ini}(\mathit{pay}).(\mathit{ccnr}) \sim \mathit{Ini}(\mathit{pay}).(\mathit{ccnr}') && \Leftrightarrow \mathit{ccnr} = \mathit{ccnr}'\\
&\mathit{Fin}(\mathit{pay}).(r) \sim \Box && \Leftrightarrow \mathit{false}\\
&\mathit{Fin}(\mathit{pay}).(r) \sim \mathit{Fin}(\mathit{pay}).(r') && \Leftrightarrow \mathit{true}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Ini}(\mathit{getAllNums}).(x) \sim \Box && \Leftrightarrow \mathit{true}\\
&\mathit{Ini}(\mathit{getAllNums}).(x) \sim \mathit{Ini}(\mathit{getAllNums}).(x') && \Leftrightarrow \mathit{true}\\
&\mathit{Fin}(\mathit{getAllNums}).(r1,r2,r3) \sim \Box && \Leftrightarrow \mathit{true}\\
&\mathit{Fin}(\mathit{getAllNums}).(r1,r2,r3) \sim \mathit{Fin}(\mathit{getAllNums}).(r1',r2',r3')\\
& && \Leftrightarrow \mathit{true}
\end{aligned}
$$

For the `Sales` component:

$$\begin{aligned}
&\mathit{Ini}(\mathit{lastSale}).(x) \sim \Box && \Leftrightarrow \mathit{false}\\
&\mathit{Ini}(\mathit{lastSale}).(x) \sim \mathit{Ini}(\mathit{lastSale}).(x') && \Leftrightarrow \mathit{true}\\
&\mathit{Fin}(\mathit{lastSale}).(r1, r2, r3, r4) \sim \Box && \Leftrightarrow \mathit{false}\\
&\mathit{Fin}(\mathit{lastSale}).(r1, r2, r3, r4) \sim \mathit{Fin}(\mathit{lastSale}).(r1', r2', r3', r4')\\
&\quad \Leftrightarrow r1 = r1' \wedge r2 = r2' \wedge r3 = r3' \wedge r4 = r4'
\end{aligned}$$

$$\begin{aligned}
&\mathit{Ini}(\mathit{registerSale}).(po, pr, am, cc) \sim \Box && \Leftrightarrow \mathit{false}\\
&\mathit{Ini}(\mathit{registerSale}).(po, pr, am, cc) \sim \mathit{Ini}(\mathit{registerSale}).(po', pr', am', cc')\\
&\quad \Leftrightarrow po = po' \wedge pr = pr' \wedge am = am' \wedge cc = cc'\\
&\mathit{Fin}(\mathit{registerSale}).(r) \sim \Box && \Leftrightarrow \mathit{false}\\
&\mathit{Fin}(\mathit{registerSale}).(r) \sim \mathit{Fin}(\mathit{registerSale}).(r') && \Leftrightarrow \mathit{true}
\end{aligned}$$

For the `Controlling` component:

$$\begin{aligned}
&\mathit{Ini}(\mathit{numBuys}).(x) \sim \Box && \Leftrightarrow \mathit{true}\\
&\mathit{Ini}(\mathit{numBuys}).(x) \sim \mathit{Ini}(\mathit{numBuys}).(x') && \Leftrightarrow \mathit{true}\\
&\mathit{Fin}(\mathit{numBuys}).(r) \sim \Box && \Leftrightarrow \mathit{true}\\
&\mathit{Fin}(\mathit{numBuys}).(r) \sim \mathit{Fin}(\mathit{numBuys}).(r') && \Leftrightarrow \mathit{true}
\end{aligned}$$

$$\begin{aligned}
&\mathit{Ini}(\mathit{numPays}).(x) \sim \Box && \Leftrightarrow \mathit{true}\\
&\mathit{Ini}(\mathit{numPays}).(x) \sim \mathit{Ini}(\mathit{numPays}).(x') && \Leftrightarrow \mathit{true}\\
&\mathit{Fin}(\mathit{numPays}).(r) \sim \Box && \Leftrightarrow \mathit{true}\\
&\mathit{Fin}(\mathit{numPays}).(r) \sim \mathit{Fin}(\mathit{numPays}).(r') && \Leftrightarrow \mathit{true}
\end{aligned}$$

$$\begin{aligned}
&\mathit{Ini}(\mathit{numChecks}).(x) \sim \Box && \Leftrightarrow \mathit{true}\\
&\mathit{Ini}(\mathit{numChecks}).(x) \sim \mathit{Ini}(\mathit{numChecks}).(x') && \Leftrightarrow \mathit{true}\\
&\mathit{Fin}(\mathit{numChecks}).(r) \sim \Box && \Leftrightarrow \mathit{true}\\
&\mathit{Fin}(\mathit{numChecks}).(r) \sim \mathit{Fin}(\mathit{numChecks}).(r') && \Leftrightarrow \mathit{true}
\end{aligned}$$

A.2.2 Controlling Department

For the `Cart` component:

$$
\begin{aligned}
&\mathit{Ini}(\mathit{buy}).(\mathit{prod}, \mathit{price}, \mathit{am}) \sim \Box && \Leftrightarrow \mathit{false}\\
&\mathit{Ini}(\mathit{buy}).(\mathit{prod}, \mathit{price}, \mathit{am}) \sim \mathit{Ini}(\mathit{buy}).(\mathit{prod}', \mathit{price}', \mathit{am}')\\
& && \Leftrightarrow \mathit{true}\\
&\mathit{Fin}(\mathit{buy}).(r) \sim \Box && \Leftrightarrow \mathit{false}\\
&\mathit{Fin}(\mathit{buy}).(r) \sim \mathit{Fin}(\mathit{buy}).(r') && \Leftrightarrow \mathit{true}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Ini}(\mathit{checkCart}).(x) \sim \Box && \Leftrightarrow \mathit{false}\\
&\mathit{Ini}(\mathit{checkCart}).(x) \sim \mathit{Ini}(\mathit{checkCart}).(x') && \Leftrightarrow \mathit{true}\\
&\mathit{Fin}(\mathit{checkCart}).(r1, r2, r3) \sim \Box && \Leftrightarrow \mathit{false}\\
&\mathit{Fin}(\mathit{checkCart}).(r1, r2, r3) \sim \mathit{Fin}(\mathit{checkCart}).(r1', r2', r3')\\
& && \Leftrightarrow \mathit{true}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Ini}(\mathit{clearCart}).(x) \sim \Box && \Leftrightarrow \mathit{true}\\
&\mathit{Ini}(\mathit{clearCart}).(x) \sim \mathit{Ini}(\mathit{clearCart}).(x') && \Leftrightarrow \mathit{true}\\
&\mathit{Fin}(\mathit{clearCart}).(r) \sim \Box && \Leftrightarrow \mathit{true}\\
&\mathit{Fin}(\mathit{clearCart}).(r) \sim \mathit{Fin}(\mathit{clearCart}).(r') && \Leftrightarrow \mathit{true}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Ini}(\mathit{pay}).(\mathit{ccnr}) \sim \Box && \Leftrightarrow \mathit{false}\\
&\mathit{Ini}(\mathit{pay}).(\mathit{ccnr}) \sim \mathit{Ini}(\mathit{pay}).(\mathit{ccnr}') && \Leftrightarrow \mathit{true}\\
&\mathit{Fin}(\mathit{pay}).(r) \sim \Box && \Leftrightarrow \mathit{false}\\
&\mathit{Fin}(\mathit{pay}).(r) \sim \mathit{Fin}(\mathit{pay}).(r') && \Leftrightarrow \mathit{true}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Ini}(\mathit{getAllNums}).(x) \sim \Box && \Leftrightarrow \mathit{false}\\
&\mathit{Ini}(\mathit{getAllNums}).(x) \sim \mathit{Ini}(\mathit{getAllNums}).(x') && \Leftrightarrow \mathit{true}\\
&\mathit{Fin}(\mathit{getAllNums}).(r1, r2, r3) \sim \Box && \Leftrightarrow \mathit{false}\\
&\mathit{Fin}(\mathit{getAllNums}).(r1, r2, r3) \sim \mathit{Fin}(\mathit{getAllNums}).(r1', r2', r3')\\
&\qquad \Leftrightarrow r1 = r1' \wedge r2 = r2' \wedge r3 = r3'
\end{aligned}
$$

For the `Sales` component:

$$\begin{aligned}
&\mathit{Ini}(\mathit{lastSale}).(x) \sim \Box && \Leftrightarrow \mathit{true}\\
&\mathit{Ini}(\mathit{lastSale}).(x) \sim \mathit{Ini}(\mathit{lastSale}).(x') && \Leftrightarrow \mathit{true}\\
&\mathit{Fin}(\mathit{lastSale}).(r1, r2, r3, r4) \sim \Box && \Leftrightarrow \mathit{true}\\
&\mathit{Fin}(\mathit{lastSale}).(r1, r2, r3, r4) \sim \mathit{Fin}(\mathit{lastSale}).(r1', r2', r3', r4')\\
& && \Leftrightarrow \mathit{true}
\end{aligned}$$

$$\begin{aligned}
&\mathit{Ini}(\mathit{registerSale}).(po, pr, am, cc) \sim \Box && \Leftrightarrow \mathit{true}\\
&\mathit{Ini}(\mathit{registerSale}).(po, pr, am, cc) \sim \mathit{Ini}(\mathit{registerSale}).(po', pr', am', cc')\\
& && \Leftrightarrow \mathit{true}\\
&\mathit{Fin}(\mathit{registerSale}).(r) \sim \Box && \Leftrightarrow \mathit{true}\\
&\mathit{Fin}(\mathit{registerSale}).(r) \sim \mathit{Fin}(\mathit{registerSale}).(r') && \Leftrightarrow \mathit{true}
\end{aligned}$$

For the `Controlling` component:

$$\begin{aligned}
&\mathit{Ini}(\mathit{numBuys}).(x) \sim \Box && \Leftrightarrow \mathit{false}\\
&\mathit{Ini}(\mathit{numBuys}).(x) \sim \mathit{Ini}(\mathit{numBuys}).(x') && \Leftrightarrow \mathit{true}\\
&\mathit{Fin}(\mathit{numBuys}).(r) \sim \Box && \Leftrightarrow \mathit{false}\\
&\mathit{Fin}(\mathit{numBuys}).(r) \sim \mathit{Fin}(\mathit{numBuys}).(r') && \Leftrightarrow r = r'
\end{aligned}$$

$$\begin{aligned}
&\mathit{Ini}(\mathit{numPays}).(x) \sim \Box && \Leftrightarrow \mathit{false}\\
&\mathit{Ini}(\mathit{numPays}).(x) \sim \mathit{Ini}(\mathit{numPays}).(x') && \Leftrightarrow \mathit{true}\\
&\mathit{Fin}(\mathit{numPays}).(r) \sim \Box && \Leftrightarrow \mathit{false}\\
&\mathit{Fin}(\mathit{numPays}).(r) \sim \mathit{Fin}(\mathit{numPays}).(r') && \Leftrightarrow r = r'
\end{aligned}$$

$$\begin{aligned}
&\mathit{Ini}(\mathit{numChecks}).(x) \sim \Box && \Leftrightarrow \mathit{false}\\
&\mathit{Ini}(\mathit{numChecks}).(x) \sim \mathit{Ini}(\mathit{numChecks}).(x') && \Leftrightarrow \mathit{true}\\
&\mathit{Fin}(\mathit{numChecks}).(r) \sim \Box && \Leftrightarrow \mathit{false}\\
&\mathit{Fin}(\mathit{numChecks}).(r) \sim \mathit{Fin}(\mathit{numChecks}).(r') && \Leftrightarrow r = r'
\end{aligned}$$

Web Shop Case Study

A.3 System-wide Security Property

Domain-motivated specification of who may know what for the web shop system.

	Customer	DeliveryDept	BillingDebt
`CartIF.getCartContent`			
`call`	✓	X	X
`CartIF.addToCart`			
`call`	✓	✓	✓
`prodId`	✓	✓	✓
`amount`	✓	✓	✓
`AccountIF.orderElementsInCart`			
`call`	✓	✓	✓
`AccountIF.setName`			
`call`	✓	✓	✓
`name`	✓	✓	✓
`AccountIF.setAdress`			
`call`	✓	✓	✓
`adr`	✓	✓	✓
`AccountIF.setCCNr`			
`call`	✓	X	✓
`ccnr`	✓	X	%10000
`AccountIF.setCVC`			
`call`	✓	X	X
`cvc`	✓	X	X
`BillingIF.getBillsToSend`			
`call`	✓	✓	✓
`DeliveryIF.getdeliverySheets`			
`call`	✓	✓	✓
`BankIF.makePayment`			
`result`	✓	✓	✓
`ProductDBIF.getProductPrice`			
`result`	✓	X	✓

Implementation and Verification of Privacy Store

B.1 Implementation

In the following, we present the implementation of the Privacy Store component as discussed in Chapter 8. The specifications for the methods can be found in the online available resources linked in the main chapter.

```
public final class PrivacyStore {

  private static final int NUM_FEATURES = 5;
  private static final int POS_X = 0;
  private static final int POS_Y = 1;
  private static final int FEAT1 = 2;
  private static final int FEAT2 = 3;
  private static final int FEAT3 = 4;

  private static final int BLURX = 4;
  private static final int BLURY = 4;

  private int[][] guestVectors;

  private int[][] coworkerVectors;

  private void registerGuest(int[] observation) {
    int[][] temp =
        getNewVectors(guestVectors.length + 1);

    for (int i = 0; i < guestVectors.length; i++) {
      temp[i] = guestVectors[i];
    }
    temp[guestVectors.length] = observation;
    this.guestVectors = temp;
  }

  private boolean helper(int[] observation) {
    int wasUpdated = updateGuest(observation);
    if (wasUpdated == -1) {
      wasUpdated = updateCoworker(observation);
    }
    return (wasUpdated >= 0);
  }

  private int updateGuest(int[] observation) {
    int pos = -1;
    for (int i = 0;  pos == -1 &&
                     i < guestVectors.length; i++) {
      if (sameObservation(observation,
                          guestVectors[i])) {
        pos = i;
      }
    }
    if (pos >= 0) {
      updatePerson(guestVectors, observation, pos);
    }
    return pos;
  }

  private int updateCoworker(int[] observation) {
    int pos = -1;
    for (int i = 0;  pos == -1 &&
                     i < coworkerVectors.length; i++) {
      if (sameObservation(observation,
                          coworkerVectors[i])) {
      pos = i;
      }
    }
    if (pos >= 0) {
      updatePerson(coworkerVectors, observation, pos);
    }
    return pos;
  }

  private void updatePerson(int[][] featVec,
                       int[] observation, int pos) {
    for (int i = 0; i < observation.length; i++) {
      featVec[pos][i] = observation[i];
    }
  }

  public void updateObservation(int[] observation) {
    boolean wasUpdated = helper(observation);
    if (!wasUpdated) {
      registerGuest(observation);
    }
  }

  public void registerCoworker(int[] observation) {
    int pos = findSimilar(observation,
                        this.guestVectors);
    if (pos > -1) {
      moveGuestToCoworker(pos, observation);
    } else {
      pos = findSimilar(observation,
                      this.coworkerVectors);
      if (pos >= 0) {
        updatePerson(coworkerVectors, observation, pos);
      } else {
        addCoworker(observation);
      }
    }
  }

  private void moveGuestToCoworker(int pos,
                                int[] observation) {
    removeGuest(pos);
    addCoworker(observation);
  }

  public void removeGuest(int pos) {
    int[][] newGuest =
          getNewVectors(this.guestVectors.length-1);
    int counter = 0;
    for (int i = 0; i < guestVectors.length; i++) {
      if (pos > i) {
        newGuest[i] = this.guestVectors[i];
      } else if (pos < i) {
        newGuest[i-1] = this.guestVectors[i];
      }
    }
    this.guestVectors = newGuest;
  }

  public void addCoworker(int[] observation) {
    int[][] newCoworker =
          getNewVectors(this.coworkerVectors.length+1);
    for (int m = 0; m < coworkerVectors.length; m++) {
      newCoworker[m] = this.coworkerVectors[m];
    }
    newCoworker[newCoworker.length-1] = observation;
    this.coworkerVectors=newCoworker;
  }

  private int findSimilar(int[] observation,
                          int[][] toCompare) {
    int pos = -1;
    for (int i = 0;  pos == -1 &&
                     i < toCompare.length; i++) {
      if (sameObservation(observation, toCompare[i])) {
        pos =  i;
      }
    }
    if(pos == -1) {
      return -1;
    } else {
      return pos;
    }
  }

  private int[][] getNewVectors(int l) {
    return new int[l][NUM_FEATURES];
  }

  private boolean sameObservation(int[] observation,
                                  int[] toCompare) {
    int deltaXPos =
              observation[POS_X] - toCompare[POS_X];
    if (deltaXPos < 0)
      deltaXPos = deltaXPos * -1;
    int deltaYPos =
              observation[POS_Y] - toCompare[POS_Y];
    if (deltaYPos < 0)
      deltaYPos = deltaYPos * -1;
    if (deltaXPos < BLURX && deltaYPos < BLURY) {
      if (observation[FEAT1] == toCompare[FEAT1] &&
          observation[FEAT2] == toCompare[FEAT2] &&
          observation[FEAT3] == toCompare[FEAT3]) {
        return true;
      } else {
        return false;
      }
    } else {
      return false;
    }
  }

  public int[] getGuest(int pos) {
    if (pos < 0 || pos >= guestVectors.length) {
      return null;}
    int[] ret = new int[NUM_FEATURES];
    for (int i = 0; i < NUM_FEATURES; i++) {
      ret[i] = guestVectors[pos][i];
    }
    return ret;
  }
}
```

B.2 Service Specification

B.2.1 Definition of the tracking predicate

```
/*@ public normal_behaviour
  @ requires observation.length == toCompare.length &&
  @          observation.length == NUM_FEATURES;
  @ ensures \result ==
  @  ((observation[POS_X] - toCompare[POS_X]) < BLURX &&
  @   (-1*(observation[POS_X]-toCompare[POS_X])<BLURX)&&
  @   (observation[POS_Y] - toCompare[POS_Y]) < BLURY &&
  @   (-1*(observation[POS_Y]-toCompare[POS_Y])<BLURY)&&
  @   observation[FEAT1] == toCompare[FEAT1] &&
  @   observation[FEAT2] == toCompare[FEAT2] &&
  @   observation[FEAT3] == toCompare[FEAT3]);
  @ model public strictly_pure boolean sameObs(
  @         int[] observation, int[] toCompare) {...}*/
```

B.2.2 Trace Invariant contracts

```
/*@ public normal_behavior
  @ requires true;
  @ ensures true;
  @ modifies \nothing; */
public /*@ nullable */ int[] getGuest(int pos) {...}
```

```
/*@ public normal_behaviour
  @requires observation.length == NUM_FEATURES &&
  @  (\forall int i; 0 <= i && i < guestVectors.length;
  @                  guestVectors[i] != observation) &&
  @  (\forall int i; 0<=i && i<coworkerVectors.length;
  @                coworkerVectors[i] != observation) &&
  @  (\forall int i; 0 <= i && i < guestVectors.length;
  @            sameObs(observation, guestVectors[i]) ==>
  @     (!(\exists int j;0<=j&&j<coworkerVectors.length;
  @        sameObs(observation, coworkerVectors[j]))) &&
  @     (\forall int k; 0<=k && k<guestVectors.length;
  @      sameObs(observation,guestVectors[k])==>i==k))&&
  @  (\forall int i; 0<=i && i<coworkerVectors.length;
  @         sameObs(observation, coworkerVectors[i]) ==>
  @      !(\exists int j; 0<=j && j<guestVectors.length;
  @            sameObs(observation, guestVectors[j])) &&
  @      (\forall int k; 0<=k&&k<coworkerVectors.length;
  @    sameObs(observation,coworkerVectors[k])==>i==k));
  @ensures
  @ ((\exists int i;0<=i&&i<\old(guestVectors.length);
  @       \old(sameObs(observation,guestVectors[i])))==>
  @   (\old(guestVectors.length)==guestVectors.length &&
  @   (\forall int j; 0<=j && j<guestVectors.length;
  @        (\old(sameObs(observation,guestVectors[j])))?
  @     (\forall int k; 0<=k && k<guestVectors.length;
  @                  guestVectors[j][k]==observation[k]):
  @     (\forall int k; 0<=k && k<guestVectors.length;
  @       guestVectors[j][k]==
  @                         \old(guestVectors[j][k])))&&
  @   (\old(coworkerVectors.length)==
  @                           coworkerVectors.length) &&
  @   (\forall int j; 0<=j && j<coworkerVectors.length;
  @    (\forall int k;0<=k&&k<coworkerVectors[j].length;
  @        coworkerVectors[j][k] ==
@                    \old(coworkerVectors[j][k]))))) &&
@ ((\exists int i;0 <= i&&
@                    i < \old(coworkerVectors.length);
@    \old(sameObs(observation,coworkerVectors[i])))==>
@   (\old(coworkerVectors.length)==
@                             coworkerVectors.length &&
@   (\forall int j; 0<=j && j<coworkerVectors.length;
@     (\old(sameObs(observation,coworkerVectors[j])))?
@     (\forall int k; 0<=k&&k<coworkerVectors.length;
@              coworkerVectors[j][k]==observation[k]):
@     (\forall int k; 0<=k&&k<coworkerVectors.length;
@     coworkerVectors[j][k]==
@                       \old(coworkerVectors[j][k])))&&
@   (\old(guestVectors.length) ==
@                              guestVectors.length) &&
@   (\forall int j; 0<=j && j<guestVectors.length;
@      (\forall int k; 0<=k&&k<guestVectors[j].length;
@         guestVectors[j][k]==
@                        \old(guestVectors[j][k])))))&&
@ (((!(\exists int i; 0 <= i &&
@                       i < \old(guestVectors.length);
@      \old(sameObs(observation,guestVectors[i]))))&&
@ (!(\exists int i; 0<=i &&
@                       i<\old(coworkerVectors.length);
@   \old(sameObs(observation, coworkerVectors[i])))))
@  ==>
@   (guestVectors.length ==
@                         \old(guestVectors.length)+1&&
@    coworkerVectors.length ==
@                       \old(coworkerVectors.length) &&
@   (\forall int i;0<=i&&i<\old(guestVectors.length);
@     (\forall int j; 0<=j &&
@                             j<guestVectors[i].length;
@      guestVectors[i][j] ==
@                         \old(guestVectors[i][j]))) &&
@   (\forall int i; 0 <= i && i < observation.length;
@   guestVectors[guestVectors.length-1][i]==
@                                   observation[i]) &&
@   (\forall int i; 0<=i &&
@                       i<\old(coworkerVectors.length);
@       (\forall int j; 0<=j &&
@                           j<coworkerVectors[i].length;
@        coworkerVectors[i][j]==
@                    \old(coworkerVectors[i][j])))));
```

```
 *\
 public void updateObservation(int[] observation) {...}
```

```
/*@ public normal_behaviour
  @ requires observation.length == NUM_FEATURES &&
  @ (\forall int i; 0 <= i && i < guestVectors.length;
  @                   guestVectors[i] != observation) &&
  @ (\forall int i; 0<=i && i<coworkerVectors.length;
  @                coworkerVectors[i] != observation) &&
  @ (\forall int i; 0 <= i && i < guestVectors.length;
  @           sameObs(observation, guestVectors[i]) ==>
  @   (!(\exists int j; 0<=j&&j<coworkerVectors.length;
  @         sameObs(observation,coworkerVectors[j]))) &&
  @   (\forall int k; 0 <= k && k < guestVectors.length;
  @      sameObs(observation,guestVectors[k])==>i==k))&&
  @ (\forall int i; 0<=i && i<coworkerVectors.length;
  @         sameObs(observation, coworkerVectors[i]) ==>
  @   !(\exists int j; 0<=j &&j< guestVectors.length;
  @           sameObs(observation, guestVectors[j])) &&
  @   (\forall int k; 0<=k && k<coworkerVectors.length;
  @    sameObs(observation,coworkerVectors[k])==>i==k));
  @ ensures
  @ ((\exists int i; 0<=i&&i<\old(guestVectors.length);
  @      \old(sameObs(observation,guestVectors[i])))==>
  @   (\old(guestVectors.length)==
  @                             guestVectors.length+1 &&
  @    (\forall int j; 0<=j && j<guestVectors.length;
  @       (\old(sameObs(observation, guestVectors[j])))?
  @      ((\forall int m; 0 <= m && m < j;
  @        (\forall int k; 0<=k&&k<guestVectors.length;
  @          guestVectors[m][k]==
  @                       \old(guestVectors[m][k]))) &&
  @        (\forall int m; j<m && m<guestVectors.length;
  @          (\forall int k;0<=k&&k<guestVectors.length;
  @               guestVectors[m-1][k]==
  @                        \old(guestVectors[m][k])))):
  @       true) &&
  @    (\old(coworkerVectors.length) + 1 ==
  @                            coworkerVectors.length) &&
  @    (\forall int j;0<=j&&j<coworkerVectors.length-1;
  @    (\forall int k;0<=k&&k<coworkerVectors[j].length;
  @          coworkerVectors[j][k] ==
@                     \old(coworkerVectors[j][k]))) &&
@   (\forall int k; 0 <= k &&
@         k < coworkerVectors[
@                    coworkerVectors.length-1].length;
@      coworkerVectors[coworkerVectors.length-1][k] ==
@                                 observation[k]))) &&
@ ((\exists int i; 0 <= i &&
@                    i < \old(coworkerVectors.length);
@    \old(sameObs(observation,coworkerVectors[i])))==>
@    (\old(coworkerVectors.length) ==
@                            coworkerVectors.length &&
@    (\forall int j; 0 <= j &&
@                          j < coworkerVectors.length;
@     (\old(sameObs(observation,coworkerVectors[j])))?
@     (\forall int k;0<=k && k<coworkerVectors.length;
@              coworkerVectors[j][k]==observation[k]):
@     (\forall int k;0<=k && k<coworkerVectors.length;
@          coworkerVectors[j][k]==
@                     \old(coworkerVectors[j][k]))) &&
@    (\old(guestVectors.length) ==
@                              guestVectors.length) &&
@    (\forall int j; 0<=j && j<guestVectors.length;
@       (\forall int k; 0 <= k &&
@                          k < guestVectors[j].length;
@           guestVectors[j][k] ==
@                      \old(guestVectors[j][k]))))) &&
@ (((!(\exists int i; 0 <= i &&
@                       i < \old(guestVectors.length);
@      \old(sameObs(observation,guestVectors[i])))) &&
@ (!(\exists int i; 0 <= i &&
@                    i < \old(coworkerVectors.length);
@  \old(sameObs(observation,coworkerVectors[i])))))==>
@  (guestVectors.length==\old(guestVectors.length) &&
@  coworkerVectors.length ==
@                   \old(coworkerVectors.length)+1 &&
@  (\forall int i; 0 <= i && i < guestVectors.length;
@    (\forall int j; 0<=j && j<guestVectors[i].length;
@       guestVectors[i][j] ==
@                       \old(guestVectors[i][j]))) &&
@  (\forall int i; 0 <= i &&
@                    i < \old(coworkerVectors.length);
@    (\forall int j; 0 <= j &&
@                       j < coworkerVectors[i].length;
@       coworkerVectors[i][j] ==
  @                   \old(coworkerVectors[i][j]))) &&
  @  (\forall int i; 0 <= i && i < observation.length;
  @     coworkerVectors[coworkerVectors.length-1][i] ==
  @                                   observation[i])));
*/
  public void registerCoworker(int[] observation) {...}
```

B.2.3 Visibility-preserving contracts

Model field for state equivalence JML formalization of the low-part of the state, i.e. JML formalization for $\approx$:

```
/*@ public instance model \seq lowvalues; */
/*@ public represents lowvalues <-
  @  \seq(guestVectors.length, guestVectors,
  @    (\seq_def int j; 0; guestVectors.length;
  @    (\seq_def int k; 0; NUM_FEATURES;
  @                        guestVectors[j][k]))); */
```

```
/*@public normal_behavior
  @requires observation.length == NUM_FEATURES &&
  @ (\forall int j; 0<=j && j<coworkerVectors.length;
  @                coworkerVectors[j] != observation) &&
  @ (\forall int j; 0 <= j && j < guestVectors.length;
  @                   guestVectors[j] != observation) &&
  @ (!(\exists int j; 0<=j && j<guestVectors.length;(
  @            sameObs(observation,guestVectors[j])))) &&
  @  (\exists int j; 0<=j && j<coworkerVectors.length;(
  @           sameObs(observation,coworkerVectors[j])));
  @ensures lowvalues == \old(lowvalues);
  @*/
  public void updateObservation(int[] observation) {...}
```

```
/*@public normal_behavior
  @requires observation.length == NUM_FEATURES &&
  @ (\forall int j; 0<=j && j<coworkerVectors.length;
  @                coworkerVectors[j] != observation) &&
  @ (\forall int j; 0 <= j && j < guestVectors.length;
  @                   guestVectors[j] != observation) &&
  @ (!(\exists int j; 0<=j && j<guestVectors.length;(
  @          sameObs(observation,guestVectors[j]))));
  @ensures lowvalues == \old(lowvalues);
  @*/
  public void registerCoworker(int[] observation) {...}
```

B.2.4 Non-Interference Contracts

```
/*@ public normal_behaviour
  @ requires true;
  @ determines ((pos < 0 ||
  @              pos >= guestVectors.length))
  @             ?(null)
  @              :((\seq_def int k; 0;
  @                     NUM_FEATURES; \result[k])),
  @            lowvalues
  @ \by lowvalues, pos;
 */
public /*@ nullable */ int[] getGuest(int pos) {...}
```

```
/*@ public normal_behaviour
  @ requires observation.length == NUM_FEATURES &&
  @    (\forall int i; 0<=i && i<guestVectors.length;
  @                guestVectors[i] != observation) &&
  @  (\forall int i; 0<=i && i<coworkerVectors.length;
  @              coworkerVectors[i] != observation) &&
  @  ((\exists int j; 0<=j && j<guestVectors.length;(
  @         sameObs(observation, guestVectors[j]))) ||
  @   (!(\exists int j;0<=j&&j<coworkerVectors.length;(
  @        sameObs(observation, coworkerVectors[j])))));
  @ determines lowvalues \by lowvalues,
  @     (\exists int j; 0<=j&&j<coworkerVectors.length;(
  @          sameObs(observation, coworkerVectors[j]))),
  @     (\exists int j; 0<=j && j<guestVectors.length;(
  @             sameObs(observation, guestVectors[j]))),
  @   ((\seq_def int j; 0;
  @                observation.length; observation[j]));
  */
public void updateObservation(int[] observation) {...}
```

```
/*@ public normal_behavior
  @ requires observation.length == NUM_FEATURES &&
  @  (\forall int j; 0<=j && j<coworkerVectors.length;
  @              coworkerVectors[j] != observation) &&
  @  (\forall int j; 0<=j && j<guestVectors.length;
  @                 guestVectors[j] != observation) &&
  @  (!(\exists int j; 0<=j && j<guestVectors.length;
  @         (sameObs(observation,guestVectors[j]))) ||
  @   !(\exists int j; 0<=j && j<coworkerVectors.length;
  @         (sameObs(observation,coworkerVectors[j]))));
  @  determines lowvalues \by lowvalues,
  @     observation, observation.length,
  @     (\seq_def int k; 0; observation.length;
  @                                      observation[k]),
  @     (\exists int j; 0<=j && j<guestVectors.length;(
  @              sameObs(observation,guestVectors[j])));
  @*/
public void registerCoworker(int[] observation) {...}
```

B.3 Specification Statistics

The following table shows the relation between size of the source code and the size of the specifications for each method. The table shows separate statistics for specifications for trace-invariant preservation, visibility preservation and secure information flow.

Method	**LoC**	**Invariant LoS**	**Visibility LoS**	**InfFlow LoS**
Class level	2	23	23	24
`registerGuest`	6	12	6	15
`helper`	5	35	13	25
`updateGuest`	8	26	18	33
`updateCoworker`	8	27	19	42
`updatePerson`	3	8	11	11
`updateObservation`	4	29	9	11
`registerCoworker`	10	33	9	16
`moveGuestToCoworker`	3	14	5	12
`removeGuest`	9	11	11	12
`addCoworker`	6	14	13	13
`findSimilar`	9	17	17	20
`sameObservation`	16	4	4	4
`getGuest`	7	9	9	9
Sum	**96**	**262**	**167**	**247**

B.4 Verification

Trace-Invariant Verification The following table illustrates the effort required for verifying that each service preserves the trace invariant. We provide for each method the number of rule applications required during the proof, the number of required manual rule applications and the amount of time the tool performed in auto mode.

Method	#Rule Apps	#Manual Rules	Automode Time (sec.)
`registerGuest`	18,014	4	86
`helper`	102,825	346	7,735
`updateGuest`	42,789	9	288
`updateCoworker`	52,285	10	366
`updatePerson`	6,545	11	17
`updateObservation`	90,014	255	10,215
`registerCoworker`	226,353	244	4,768
`moveGuestToCoworker`	46,811	55	624
`removeGuest`	55,116	103	841
`addCoworker`	17,291	6	56
`findSimilar`	22,886	8	97
`sameObservation`	45,712	7	128
`getGuest`	11,372	2	32
Sum	**738,013**	**1,049**	**25,255**

State-Visiblity Preserving Verification The following table illustrates the effort required for verifying that each service is state-visibility-preserving. We provide for each method the number of rule applications required during the proof, the number of required manual rule applications and the amount of time the tool performed in auto mode.

Method	#Rule Apps	#Manual Rules	Automode Time (sec.)
`registerGuest`	2	0	0
`helper`	36,541	90	654
`updateGuest`	52,041	15	324
`updateCoworker`	65,160	85	889
`updatePerson`	8,854	1	21
`updateObservation`	19,306	29	433
`registerCoworker`	44,912	15	290
`moveGuestToCoworker`	23,715	33	339
`removeGuest`	26,150	47	300
`addCoworker`	15,630	1	47
`findSimilar`	22,886	8	92
`sameObservation`	45,712	7	124
`getGuest`	11,848	10	33
Sum	**372,757**	**341**	**3,545**

Non-Interference Verification Each information flow contract yields two proof obligations. The first proof obligation consists of a functional proof for post-conditions, loop invariants and similar. The second proof obligation is the actual relational proof comparing two executions of the service.

The following table illustrates the effort required for verifying the functional part of the information flow contract. We provide for each method the number of rule applications required during the proof, the number of required manual rule applications and the amount of time the tool performed in auto mode.

Method	**#Rule Apps**	**#Manual Rules**	**Automode Time (sec.)**
`registerGuest`	21,829	13	215
`helper`	53,353	166	5,124
`updateGuest`	57,002	28	354
`updateCoworker`	67,794	61	907
`updatePerson`	8,854	1	18
`updateObservation`	56,482	48	1,068
`registerCoworker`	126,081	35	916
`moveGuestToCoworker`	35,015	42	360
`removeGuest`	22,907	35	237
`addCoworker`	15,474	3	45
`findSimilar`	22,180	15	84
`sameObservation`	45,712	7	115
`getGuest`	10,556	2	28
Sum	**543,239**	**456**	**9,470**

The following table illustrates the effort required for verifying the relational part of the information flow contract. If the table does not contain numbers for a service, then we did not require an information flow contract for specification but a functional contract was sufficient for verification of the overall information-flow properties. We provide for each method the number of rule applications required during the proof, the number of required manual rule applications and the amount of time the tool performed in auto mode.

Method	**#Rule Apps**	**#Manual Rules**	**Automode Time (sec.)**
`registerGuest`	36,796	264	459
`helper`	154,885	362	12,924
`updateGuest`	142,834	543	1,099
`updateCoworker`	23,992	97	104
`updatePerson`	—-	—	—
`updateObservation`	258,997	280	5,566
`registerCoworker`	86,950	374	1,168
`moveGuestToCoworker`	50,673	219	445
`removeGuest`	58,384	164	1,087
`addCoworker`	—-	—	—
`findSimilar`	32,929	119	327
`sameObservation`	—-	—	—
`getGuest`	33,710	95	358
Sum	**880,150**	**2,517**	**23,536**

Bibliography

Wolfgang Ahrendt and Maximilian Dylla. A system for compositional verification of asynchronous objects. *Science of Computer Programming*, 77(12):1289–1309, 2012. doi: 10.1016/j.scico.2010.08.003. URL `https://doi.org/10.1016/j.scico.2010.08.003`. (Cited on page 148.)

Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book: From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, December 2016. doi: 10.1007/978-3-319-49812-6. (Cited on pages 102, 107, and 172.)

M. M. Alam, R. Breu, and M. Breu. Model driven security for web services (mds4ws). In *8th International Multitopic Conference, 2004. Proceedings of INMIC 2004.*, pages 498–505, Dec 2004. doi: 10.1109/INMIC.2004.1492930. (Cited on page 94.)

Torben Amtoft and Anindya Banerjee. Information flow analysis in logical form. In Roberto Giacobazzi, editor, *Static Analysis*, volume 3148 of *LNCS*. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-22791-5. doi: 10.1007/978-3-540-27864-1_10. (Cited on page 63.)

Torben Amtoft and Anindya Banerjee. Verification condition generation for conditional information flow. In *Proceedings of the 2007 ACM Workshop on Formal Methods in Security Engineering*, FMSE '07, pages 2–11, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-887-9. doi: 10.1145/1314436.1314438. (Cited on pages 64 and 148.)

Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. A logic for information flow in object-oriented programs. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming*

Languages, POPL '06, pages 91–102, New York, NY, USA, 2006. ACM. ISBN 1-59593-027-2. doi: 10.1145/1111037.1111046. URL `http://doi.acm.org/10.1145/1111037.1111046`. (Cited on page 148.)

Torben Amtoft, John Hatcliff, Edwin Rodríguez, Robby, Jonathan Hoag, and David Greve. *Specification and Checking of Software Contracts for Conditional Information Flow*, pages 229–245. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-68237-0. doi: 10.1007/978-3-540-68237-0_17. URL `https://doi.org/10.1007/978-3-540-68237-0_17`. (Cited on page 148.)

Torben Amtoft, John Hatcliff, and Edwin Rodríguez. *Precise and Automated Contract-Based Reasoning for Verification and Certification of Information Flow Properties of Programs with Arrays*, pages 43–63. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-11957-6. doi: 10.1007/978-3-642-11957-6_4. URL `https://doi.org/10.1007/978-3-642-11957-6_4`. (Cited on page 148.)

Musard Balliu, Mads Dam, and Gurvan Le Guernic. Epistemic temporal logic for information flow security. In *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*, PLAS '11, pages 6:1–6:12, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0830-4. doi: 10.1145/2166956.2166962. (Cited on page 183.)

Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *IEEE CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA*. doi: 10.1109/CSFW.2004.17. (Cited on pages 63 and 148.)

Thomas Bauereiß, Armando Pesenti Gritti, Andrei Popescu, and Franco Raimondi. *CoSMed: A Confidentiality-Verified Social Media Platform*, pages 87–106. Springer International Publishing, Cham, 2016. ISBN 978-3-319-43144-4. doi: 10.1007/978-3-319-43144-4_6. URL `https://doi.org/10.1007/978-3-319-43144-4_6`. (Cited on page 183.)

Thomas Bauereiß, Simon Greiner, Mihai Herda, Michael Kirsten, Ximeng Li, Heiko Mantel, Martin Mohr, Matthias Perner, David Schneider, and Markus Tasch. Rifl 1.1: A common specification language for information-flow requirements. Technical Report TUD-CS-2017-0225, TU Darmstadt, August 2017.

Bernhard Beckert and Daniel Bruns. *Dynamic Logic with Trace Semantics*, pages 315–329. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-38574-2. doi: 10.1007/978-3-642-38574-2_22. URL `https://doi.org/10.1007/978-3-642-38574-2_22`. (Cited on page 182.)

Bernhard Beckert, Daniel Bruns, Vladimir Klebanov, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. Information flow in object-oriented software. In Gopal Gupta and Ricardo Peña, editors, *LOPSTR 2013, Revised Selected Papers*, number 8901 in Lecture Notes in Computer Science. Springer. (Cited on pages 130, 133, 134, 135, and 149.)

Najah Ben Said, Takoua Abdellatif, Saddek Bensalem, and Marius Bozga. *Model-Driven Information Flow Security for Component-Based Systems*, pages 1–20. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. ISBN 978-3-642-54848-2. doi: 10.1007/978-3-642-54848-2_1. URL `https://doi.org/10.1007/978-3-642-54848-2_1`. (Cited on page 95.)

Pascal Birnstill and Alexander Pretschner. Enforcing privacy through usage-controlled video surveillance. In *Advanced Video and Signal Based Surveillance (AVSS), 2013 10th IEEE International Conference on*, pages 318–323. IEEE, 2013. (Cited on page 156.)

Pascal Birnstill, Sebastian Bretthauer, Simon Greiner, and Erik Krempel. Privacy-preserving surveillance: an interdisciplinary approach. *International Data Privacy Law*, 5(4):298–308, September 2015. doi: 10.1093/idpl/ipv021. (Cited on page 156.)

Daniel Bruns, Huy Quoc Do, Simon Greiner, Mihai Herda, Martin Mohr, Enrico Scapin, Tomasz Truderung, Bernhard Beckert, Ralf Küsters, Heiko Mantel, and Richard Gay. Poster: Security in e-voting. In Sophie Engle, editor, *36th IEEE Symposium on Security and Privacy (S&P 2015), Poster Session*, May 2015. URL `https://www.ieee-security.org/TC/SP2015/posters/paper_10.pdf`.

Richard Bubel, Reiner Hähnle, and Benjamin Weiß. Abstract interpretation of symbolic execution with explicit state updates. In *Formal Methods for Components and Objects, 7th International Symposium, FMCO 2008, Sophia Antipolis, France, October 21-23, 2008, Revised Lectures*, pages 247–277, 2008. doi: 10.1007/978-3-642-04167-9_13. URL `https://doi.org/10.1007/978-3-642-04167-9_13`. (Cited on page 148.)

Florian Böhl, Simon Greiner, and Patrik Scheidecker. Proving correctness and security of two-party computation implemented in java in presence of a semi-honest sender. In Dimitris Gritzalis, Aggelos Kiayias, and Ioannis G. Askoxylakis, editors, *13th International Conference on Cryptology and Network Security (CANS 2014)*, volume 8813 of *Lecture Notes in Computer Science*, pages 175–190. Springer, October 2014. ISBN 978-3-319-12279-3. doi: 10.1007/978-3-319-12280-9_12. URL `http://dx.doi.org/10.1007/978-3-319-12280-9_12`.

Barbara Carminati, Elena Ferrari, Raymond Heatherly, Murat Kantarcioglu, and Bhavani Thuraisingham. Semantic web-based social network access control. *Comput. Secur.*, 30(2-3):108–115, March 2011. ISSN 0167-4048. doi: 10.1016/j.cose.2010.08.003. URL `http://dx.doi.org/10.1016/j.cose.2010.08.003`. (Cited on page 184.)

Y. Cheng, J. Park, and R. Sandhu. Relationship-based access control for online social networks: Beyond user-to-user relationships. In *2012 International Conference on Privacy, Security, Risk and Trust and 2012 International Confernece on Social Computing*, pages 646–655, Sept 2012. doi: 10.1109/SocialCom-PASSAT.2012.57. (Cited on page 184.)

David Clark and Sebastian Hunt. Non-interference for deterministic interactive programs. In Pierpaolo Degano, Joshua Guttman, and Fabio Martinelli, editors, *Formal Aspects in Security and Trust*, volume 5491 of *Lecture Notes in Computer Science*. Springer, 2009. ISBN 978-3-642-01464-2. doi: 10.1007/978-3-642-01465-9_4. (Cited on pages 4, 29, 61, and 63.)

Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal logics for hyperproperties. In Martín Abadi and Steve Kremer, editors, *POST*, volume 8414 of *Lecture Notes in Computer Science*, pages 265–284. Springer, 2014. ISBN 978-3-642-54791-1. (Cited on page 183.)

Ellis Cohen. Information transmission in computational systems. *SIGOPS Oper. Syst. Rev.*, 11(5), November 1977. ISSN 0163-5980. doi: 10.1145/1067625.806556. (Cited on page 61.)

Ádám Darvas, Reiner Hähnle, and David Sands. A theorem proving approach to analysis of secure information flow. In *2nd Int. Conf SPC 2005.* doi: 10.1007/978-3-540-32004-3_20. (Cited on pages 63 and 148.)

Karsten Diekhoff. Specification and verification of javaee services with key, 2017. Bachelor Thesis at Karlsruhe Institute for Technology. (Cited on page 98.)

Crystal Chang Din and Olaf Owe. A sound and complete reasoning system for asynchronous communication with shared futures. *J. Log. Algebr. Meth. Program.*, 83(5-6):360–383, 2014. doi: 10.1016/j.jlamp.2014.03.003. URL `https://doi.org/10.1016/j.jlamp.2014.03.003`. (Cited on page 148.)

Crystal Chang Din and Olaf Owe. Compositional reasoning about active objects with shared futures. *Formal Asp. Comput.*, 27(3):551–572, 2015. doi: 10.1007/s00165-014-0322-y. URL `https://doi.org/10.1007/s00165-014-0322-y`. (Cited on page 148.)

Crystal Chang Din, Johan Dovland, Einar Broch Johnsen, and Olaf Owe. Observable behavior of distributed systems: Component reasoning for concurrent objects. *J. Log. Algebr. Program.*, 81(3):227–256, 2012. doi: 10.1016/j.jlap.2012.01.003. URL `https://doi.org/10.1016/j.jlap.2012.01.003`. (Cited on page 148.)

Crystal Chang Din, Richard Bubel, and Reiner Hähnle. Key-abs: A deductive verification tool for the concurrent modelling language ABS. In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, pages 517–526, 2015a. doi: 10.1007/978-3-319-21401-6_35. URL `https://doi.org/10.1007/978-3-319-21401-6_35`. (Cited on page 148.)

Crystal Chang Din, Silvia Lizeth Tapia Tarifa, Reiner Hähnle, and Einar Broch Johnsen. History-based specification and verification of scalable concurrent and distributed systems. In *Formal Methods and Software Engineering - 17th International Conference on Formal Engineering Methods, ICFEM 2015, Paris, France, November 3-5, 2015, Proceedings*, pages 217–233, 2015b. doi: 10.1007/978-3-319-25423-4_14. URL `https://doi.org/10.1007/978-3-319-25423-4_14`. (Cited on page 148.)

Johan Dovland, Einar Broch Johnsen, and Olaf Owe. Verification of concurrent objects with asynchronous method calls. In *2005 IEEE International Conference on Software - Science, Technology and Engineering (SwSTE 2005), 22-23 February 2005, Herzelia, Israel*, pages 141–150, 2005. doi: 10.1109/SWSTE.2005.24. URL `https://doi.org/10.1109/SWSTE.2005.24`. (Cited on page 148.)

EJB 3.1 Expert Group. *JSR 318: Enterprise JavaBeans, Version 3.1*. Sun Microsystems, 2009. URL `https://jcp.org/en/jsr/detail?id=366`. Accessed 29/01/2016. (Cited on pages 11 and 109.)

Riccardo Focardi and Roberto Gorrieri. A classification of security properties for process algebras. *J. of Comp. Sec.*, 3:5–33, 1994. (Cited on pages 3 and 61.)

Philip W. L. Fong, Mohd M. Anwar, and Zhen Zhao. A privacy preservation model for facebook-style social network systems. In *Computer Security - ESORICS 2009, 14th European Symposium on Research in Computer Security, Saint-Malo, France, September 21-23, 2009. Proceedings*, pages 303–320, 2009. doi: 10.1007/978-3-642-04444-1_19. URL `https://doi.org/10.1007/978-3-642-04444-1_19`. (Cited on page 184.)

Philip W.L. Fong. Relationship-based access control: Protection model and policy language. In *Proceedings of the First ACM Conference on Data and Application Security and Privacy*, CODASPY '11, pages 191–202, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0466-5. doi: 10.1145/1943513.1943539. URL `http://doi.acm.org/10.1145/1943513.1943539`. (Cited on page 184.)

Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE S&P*, 1982. (Cited on page 61.)

Daniel Grahl and Simon Greiner. Non-interference with what-declassification in component-based systems. Technical Report 2015,10, Department of Informatics, Karlsruhe Institute of Technology, November 2015. URL `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000050422`.

Simon Greiner and Daniel Grahl. Non-interference with what-declassification in component-based systems. In *Proceedings of the Computer Security Foundations Symposium (CSF 2016)*, June 2016. ISBN 978-3-319-12279-3. (Cited on pages 12 and 24.)

Simon Greiner and Mihai Herda. Cocome with securitys, 2017a. URL `http://dx.doi.org/10.5445/IR/1000065106`. (Cited on pages 95 and 186.)

Simon Greiner and Mihai Herda. Cocome with security. Technical report, Karlsruhe Institute of Technology, Faculty of Informatics, Karlsruhe, April 2017b.

Simon Greiner and Jie Yang. Privacy protection in an electronic chronicle system. Proceedings of the 34th Annual Northeas Bioengineering Conference.

Simon Greiner, Pascal Birnstill, Erik Krempel, Bernhard Beckert, and Jürgen Beyerer. Privacy preserving surveillance and the tracking paradox. In *Proceedings, Future Security Conference 2013, 15–19 September 2013, Berlin*, September 2013. (Cited on pages 156 and 157.)

Simon Greiner, Martin Mohr, and Bernhard Beckert. Modular verification of information flow security in component-based systems – proofs and proof of concept. Technical Report 2017,9, Department of Informatics, Karlsruhe Institute of Technology, Karlsruhe, June 2017a. URL `http://dx.doi.org/10.5445/IR/1000070463`. (Cited on pages 49, 60, 98, 143, 145, 150, and 186.)

Simon Greiner, Martin Mohr, and Bernhard Beckert. Modular verification of information flow security in component-based systems. In Alessandro Cimatti and Marjan Sirjani, editors, *15th International Conference on Software Engineering and Formal Methods (SEFM 2017)*, volume 10469 of *Lecture Notes in Computer Science*, pages 300–315. Springer, September 2017b. doi: 10.1007/978-3-319-66197-1_19. (Cited on pages 49, 60, 98, 143, 145, 150, and 186.)

Michael Hafner and Ruth Breu. *Modeling Security Critical SOA Applications*, pages 93–119. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-540-79539-1. doi: 10.1007/978-3-540-79539-1_7. URL `https://doi.org/10.1007/978-3-540-79539-1_7`. (Cited on pages 3 and 94.)

Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, December 2009. doi: 10.1007/s10207-009-0086-1. (Cited on page 149.)

David Harel, Jerzy Tiuryn, and Dexter Kozen. *Dynamic Logic*. MIT Press, Cambridge, MA, USA, 2000. ISBN 0262082896. (Cited on page 102.)

Daniel Hedin and Andrei Sabelfeld. A perspective on information-flow control. In *Software Safety and Security - Tools for Analysis and Verification*, pages 319–347. 2012. doi: 10.3233/978-1-61499-028-4-319. URL `https://doi.org/10.3233/978-1-61499-028-4-319`. (Cited on page 149.)

George T. Heineman and William T. Councill, editors. *Component-based Software Engineering: Putting the Pieces Together*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. ISBN 0-201-70485-4. (Cited on pages 3 and 11.)

C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969. ISSN 0001-0782. (Cited on pages 65 and 148.)

JavaEE. *Enterprise JavaBeans, version 3.2*, 2013. URL `http://jcp.org/en/jsr/detail?id=345`. (Cited on pages 3 and 98.)

Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. *ABS: A Core Language for Abstract Behavioral Specification*, pages 142–164. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-25271-6. doi: 10.1007/978-3-642-25271-6_8. (Cited on page 147.)

Rajeev Joshi and K. Rustan M. Leino. A semantic approach to secure information flow. *Sci. Comput. Program.*, 37(1-3), 2000. doi: 10.1016/S0167-6423(99)00024-6. (Cited on pages 63 and 148.)

Jan Jürjens. *Model-Based Security Engineering with UML*, pages 42–77. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. ISBN 978-3-540-31936-8. doi: 10.1007/11554578_2. URL `https://doi.org/10.1007/11554578_2`. (Cited on pages 3 and 94.)

Sudeep Kanav, Peter Lammich, and Andrei Popescu. *A Conference Management System with Verified Document Confidentiality*, pages 167–183. Springer International Publishing, Cham, 2014. ISBN 978-3-319-08867-9. doi: 10.1007/978-3-319-08867-9_11. URL `https://doi.org/10.1007/978-3-319-08867-9_11`. (Cited on page 183.)

Ioannis T. Kassios. *Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions*, pages 268–283. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-37216-5. doi: 10.1007/11813040_19. URL `http://dx.doi.org/10.1007/11813040_19`. (Cited on pages 54, 65, and 104.)

Kuzman Katkalov, Peter Fischer, Kurt Stenzel, Nina Moebius, and Wolfgang Reif. *Evaluation of Jif and Joana as Information Flow Analyzers in a Model-Driven Approach*, pages 174–186. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-35890-6. doi: 10.1007/978-3-642-35890-6_13. URL `https://doi.org/10.1007/978-3-642-35890-6_13`. (Cited on page 94.)

Max E. Kramer, Anton Hergenröder, Martin Hecker, Simon Greiner, and Kaibin Bao. Specification and verification of confidentiality in component-based systems. Poster at the 35th IEEE Symposium on Security and Privacy, 2014. URL `http://www.ieee-security.org/TC/SP2014/posters/KRAME.pdf`. (Cited on page 74.)

Max E. Kramer, Martin Hecker, Simon Greiner, and Kateryna Yurchenko. Model-driven specification and analysis of confidentiality in component-based systems. Technical report, Karlsruhe Institute of Technology, Faculty of Informatics, Karlsruhe, November 2017. (Cited on pages 74, 84, 96, and 186.)

Jonas Krämer. Specification and verification of service-local dependency clusters in key, 2017. Bachelor Thesis at Karlsruhe Institute for Technology. (Cited on page 98.)

Thomas Lauscher and Simon Greiner. Wirtschaftlichkeit bei der verbesserung von systemspezifikationen durch uml-modellierung. *Signal und Draht*, 104 (12):21, December 2012.

Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, and Patrice Chalin. Jml reference manual, 2008. (Cited on page 106.)

Luísa Lourenço and Luís Caires. Dependent information flow types. *SIGPLAN Not.*, 50(1):317–328, January 2015. ISSN 0362-1340. doi: 10.1145/2775051.2676994. (Cited on page 149.)

Heiko Mantel. Possibilistic definitions of security – an assembly kit. In *13th IEEE CSFW '00*. doi: 10.1109/csfw.2000.856936. (Cited on pages 3 and 61.)

Heiko Mantel, David Sands, and Henning Sudbrock. Assumptions and guarantees for compositional noninterference. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium, CSF 2011, Cernay-la-Ville, France, 27-29 June, 2011*, pages 218–232, 2011. doi: 10.1109/

CSF.2011.22. URL `https://doi.org/10.1109/CSF.2011.22`. (Cited on page 64.)

Bertrand Meyer. The grand challenge of trusted components. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 660–667, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1877-X. (Cited on pages 3 and 11.)

Toby C. Murray, Robert Sison, Edward Pierzchalski, and Christine Rizkallah. Compositional verification and refinement of concurrent value-dependent noninterference. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, pages 417–431, 2016. doi: 10.1109/CSF.2016.36. URL `https://doi.org/10.1109/CSF.2016.36`. (Cited on page 64.)

Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif 3.0: Java information flow, July 2006. URL `http://www.cs.cornell.edu/jif`. (Cited on page 149.)

A. Nanevski, A. Banerjee, and D. Garg. Verification of information flow and access control policies with dependent types. In *2011 IEEE Symposium on Security and Privacy*, pages 165–179, May 2011. doi: 10.1109/SP.2011.12. (Cited on page 149.)

Phu H. Nguyen, Max Kramer, Jacques Klein, and Yves Le Traon. An extensive systematic review on the model-driven development of secure systems. *Information and Software Technology*, 68(Supplement C):62 – 81, 2015. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2015.08.006. (Cited on pages 3 and 94.)

Tobias Nipkow and Gerwin Klein. *Concrete Semantics: With Isabelle/HOL*. Springer Publishing Company, Incorporated, 2014. ISBN 3319105418, 9783319105413. (Cited on page 183.)

Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002. ISBN 3-540-43376-7. (Cited on page 183.)

Kevin R. O'Neill, Michael R. Clarkson, and Stephen Chong. Information-flow security for interactive programs. In *19th IEEE CSF 2006*, Piscataway, NJ, USA, July . IEEE Press. (Cited on pages 4 and 61.)

François Pottier. A simple view of type-secure information flow in the pi-calculus. In *15th IEEE CSF 2002*, CSFW '02, Washington, DC, USA. IEEE Computer Society. ISBN 0-7695-1689-0. (Cited on pages 3 and 61.)

Willard Rafnsson, Daniel Hedin, and Andrei Sabelfeld. Securing interactive programs. In *25th IEEE CSF 2012*, 2012. doi: 10.1109/CSF.2012.15. (Cited on pages 4, 12, 31, 33, 62, and 63.)

Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and F Plasil. The common component modeling example. *Lecture notes in computer science*, 5153, 2008. (Cited on page 96.)

Ralf H. Reussner, Steffen Becker, Jens Happe, Robert Heinrich, Anne Koziolek, Heiko Koziolek, Max Kramer, and Klaus Krogmann. *Modeling and Simulating Software Architectures: The Palladio Approach.* The MIT Press, 2016. ISBN 026203476X, 9780262034760. (Cited on pages 3, 11, 74, 75, and 77.)

John C Reynolds. Idealized algol and its specification logic. *Tools and notions for program construction*, pages 121–161, 1982. (Cited on page 65.)

John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 55–74. IEEE, 2002. (Cited on page 65.)

Peter Y. A. Ryan and Steve A. Schneider. Process algebra and non-interference. In *CSFW 1999*. IEEE Computer Society. ISBN 0-7695-0201-6. (Cited on pages 3 and 61.)

Andrei Sabelfeld and Heiko Mantel. Static Confidentiality Enforcement for Distributed Programs. In Manuel Hermenegildo and Germán Puebla, editors, *Static Analysis*, volume 2477 of *LNCS*. Springer, 2002. doi: 10.1007/3-540-45789-5_27. (Cited on pages 3 and 61.)

Andrei Sabelfeld and Andrew C. Myers. A model for delimited information release. In Kokichi Futatsugi, Fumio Mizoguchi, and Naoki Yonezaki, editors, *Software Security - Theories and Systems*, pages 174–191, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. ISBN 978-3-540-37621-7. (Cited on page 63.)

Andrei Sabelfeld and David Sands. A PER model of secure information flow in sequential programs. *Higher-Order and Symbolic Computation*, 14(1), 2001. doi: 10.1023/A:1011553200337. (Cited on pages 63 and 149.)

Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *J. Comput. Secur.*, 17(5), October 2009. ISSN 0926-227X. (Cited on pages 24 and 62.)

Christoph Scheben. *Program-level Specification and Deductive Verification of Security Properties.* PhD thesis, Karlsruhe Institute of Technology, 2014. (Cited on pages 102, 172, and 179.)

Christoph Scheben and Simon Greiner. Information flow analysis. In *Deductive Software Verification - The KeY Book: From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*, chapter 13, pages 453–471. Springer, December 2016. doi: 10.1007/978-3-319-49812-6_13.

Christoph Scheben and Peter H. Schmitt. *Verification of Information Flow Properties of Java Programs without Approximations*, pages 232–249. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-31762-0. doi: 10.1007/978-3-642-31762-0_15. URL `https://doi.org/10.1007/978-3-642-31762-0_15`. (Cited on page 149.)

Christoph Scheben and Peter H. Schmitt. *Efficient Self-composition for Weakest Precondition Calculi*, pages 579–594. Springer International Publishing, Cham, 2014. ISBN 978-3-319-06410-9. doi: 10.1007/978-3-319-06410-9_39. URL `http://dx.doi.org/10.1007/978-3-319-06410-9_39`. (Cited on pages 147 and 149.)

Kurt Stenzel, Kuzman Katkalov, Marian Borek, and Wolfgang Reif. A model-driven approach to noninterference. pages 30–43, 2014. (Cited on pages 3 and 94.)

Nikhil Swamy, Juan Chen, and Ravi Chugh. *Enforcing Stateful Authorization and Information Flow Policies in Fine*, pages 529–549. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-11957-6. doi: 10.1007/978-3-642-11957-6_28. URL `https://doi.org/10.1007/978-3-642-11957-6_28`. (Cited on page 149.)

Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 266–278, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0865-6. doi: 10.1145/2034773.2034811. URL `http://doi.acm.org/10.1145/2034773.2034811`. (Cited on page 149.)

Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software: Beyond Object-oriented Programming*. Pearson Education, 2002. ISBN 978-0-201-74572-6. (Cited on pages 3 and 11.)

Mathy Vanhoef, Willem De Groef, Dominique Devriese, Frank Piessens, and Tamara Rezk. Stateful declassification policies for event-driven programs. In *IEEE 27th CSF 2014*. IEEE, July . doi: DOI10.1109/CSF.2014.28. (Cited on page 62.)

Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, Reading, MA, 1999. ISBN 978-0-201-37940-2. (Cited on page 87.)

Benjamin Weiß. *Deductive Verification of Object-oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction.* PhD thesis, Karlsruhe Institute of Technology, January 2011. URL `http://digbib.ubka.uni-karlsruhe.de/volltexte/documents/1600837`. (Cited on pages 102, 106, 107, and 116.)

Andy Wigley, Stephen Wheelwright, Robert Burbridge, Rory MacLoed, and Mark Sutton. *Microsoft .NET Compact Framework (Core Reference)*. Microsoft Press, 2003. (Cited on pages 3 and 11.)

J. Todd Wittbold and Dale M. Johnson. Information flow in nondeterministic systems. In *IEEE S&P 1990*. IEEE Computer Society. ISBN 0-8186-2060-9. (Cited on pages 4 and 61.)

Kateryna Yurchenko, Moritz Behr, Heiko Klare, Max Kramer, and Ralf Reussner. Architecture-driven reduction of specification overhead for verifying confidentiality in component-based software systems. In *MoDeVVa at MoDELS*, 2017. accepted. (Cited on page 150.)

Lantian Zheng and Andrew C. Myers. Dynamic security labels and static information flow control. *Int. J. Inf. Sec.*, 6(2-3):67–84, 2007. doi: 10.1007/s10207-007-0019-9. URL `https://doi.org/10.1007/s10207-007-0019-9`. (Cited on page 149.)

Publication List

Peer-Reviewed full paper

- Simon Greiner, Martin Mohr, and Bernhard Beckert. Modular verification of information flow security in component-based systems. In Alessandro Cimatti and Marjan Sirjani, editors, *15th International Conference on Software Engineering and Formal Methods (SEFM 2017)*, volume 10469 of *Lecture Notes in Computer Science*, pages 300–315. Springer, September 2017b. doi: 10.1007/978-3-319-66197-1_19

- Simon Greiner and Daniel Grahl. Non-interference with what-declassification in component-based systems. In *Proceedings of the Computer Security Foundations Symposium (CSF 2016)*, June 2016. ISBN 978-3-319-12279-3

- Florian Böhl, Simon Greiner, and Patrik Scheidecker. Proving correctness and security of two-party computation implemented in java in presence of a semi-honest sender. In Dimitris Gritzalis, Aggelos Kiayias, and Ioannis G. Askoxylakis, editors, *13th International Conference on Cryptology and Network Security (CANS 2014)*, volume 8813 of *Lecture Notes in Computer Science*, pages 175–190. Springer, October 2014. ISBN 978-3-319-12279-3. doi: 10.1007/978-3-319-12280-9_12. URL `http://dx.doi.org/10.1007/978-3-319-12280-9_12`

- Simon Greiner, Pascal Birnstill, Erik Krempel, Bernhard Beckert, and Jürgen Beyerer. Privacy preserving surveillance and the tracking paradox. In *Proceedings, Future Security Conference 2013, 15–19 September 2013, Berlin*, September 2013

Journal article

- Pascal Birnstill, Sebastian Bretthauer, Simon Greiner, and Erik Krempel. Privacy-preserving surveillance: an interdisciplinary approach. *International Data Privacy Law*, 5(4):298–308, September 2015. doi: 10.1093/idpl/ipv021

- Thomas Lauscher and Simon Greiner. Wirtschaftlichkeit bei der verbesserung von systemspezifikationen durch uml-modellierung. *Signal und Draht*, 104(12):21, December 2012

Book Chapter

- Christoph Scheben and Simon Greiner. Information flow analysis. In *Deductive Software Verification - The KeY Book: From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*, chapter 13, pages 453–471. Springer, December 2016. doi: 10.1007/978-3-319-49812-6_13

Peer-Reviewed Poster and Short Paper

- Daniel Bruns, Huy Quoc Do, Simon Greiner, Mihai Herda, Martin Mohr, Enrico Scapin, Tomasz Truderung, Bernhard Beckert, Ralf Küsters, Heiko Mantel, and Richard Gay. Poster: Security in e-voting. In Sophie Engle, editor, *36th IEEE Symposium on Security and Privacy (S&P 2015), Poster Session*, May 2015. URL `https://www.ieee-security.org/TC/SP2015/posters/paper_10.pdf`

- Max E. Kramer, Anton Hergenröder, Martin Hecker, Simon Greiner, and Kaibin Bao. Specification and verification of confidentiality in component-based systems. Poster at the 35th IEEE Symposium on Security and Privacy, 2014. URL `http://www.ieee-security.org/TC/SP2014/posters/KRAME.pdf`

- Simon Greiner and Jie Yang. Privacy protection in an electronic chronicle system. Proceedings of the 34th Annual Northeas Bioengineering Conference

Non-Peer-Reviewed Publications

- Max E. Kramer, Martin Hecker, Simon Greiner, and Kateryna Yurchenko. Model-driven specification and analysis of confidentiality in component-based systems. Technical report, Karlsruhe Institute of Technology, Faculty of Informatics, Karlsruhe, November 2017
- Simon Greiner, Martin Mohr, and Bernhard Beckert. Modular verification of information flow security in component-based systems – proofs and proof of concept. Technical Report 2017,9, Department of Informatics, Karlsruhe Institute of Technology, Karlsruhe, June 2017a. URL `http://dx.doi.org/10.5445/IR/1000070463`
- Thomas Bauereiß, Simon Greiner, Mihai Herda, Michael Kirsten, Ximeng Li, Heiko Mantel, Martin Mohr, Matthias Perner, David Schneider, and Markus Tasch. Rifl 1.1: A common specification language for information-flow requirements. Technical Report TUD-CS-2017-0225, TU Darmstadt, August 2017
- Simon Greiner and Mihai Herda. Cocome with security. Technical report, Karlsruhe Institute of Technology, Faculty of Informatics, Karlsruhe, April 2017b
- Daniel Grahl and Simon Greiner. Non-interference with what-declassification in component-based systems. Technical Report 2015,10, Department of Informatics, Karlsruhe Institute of Technology, November 2015. URL `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000050422`