UiO : **University of Oslo**

Shukun Tokas

Analysis and Enforcement of GDPR-related Privacy Principles in Object-Oriented Distributed Systems

Thesis submitted for the degree of Philosophiae Doctor

Department of Informatics
Faculty of Mathematics and Natural Sciences

Faculty of Mathematics and Natural Sciences

2021

© Shukun Tokas, 2021

Series of dissertations submitted to the
Faculty of Mathematics and Natural Sciences, University of Oslo
No. 1234

ISSN 1234-5678

All rights reserved. No part of this publication may be
reproduced or transmitted, in any form or by any means, without permission.

Cover: Hanne Baadsgaard Utigard.
Print production: Reprosentralen, University of Oslo.

To Sushila Devi, Satyapal Singh Tokas, Dilbag Singh Tokas, and Ravi Tokas.

Abstract

It is important to have a meaningful balance between innovation, economic growth, and fundamental privacy rights. Data protection laws and regulations have evolved as a result of the interactive relationship between businesses, technologies, citizens, and governments. With an ever-burgeoning amount of data and the requirement to comply with numerous data protection regulations, it is essential to align the software ecosystem with the privacy-related requirements for better data protection. To strengthen data protection and protect privacy of the individuals within the European Union (EU) and the European Economic Area (EEA), the European Union Parliament approved the General Data Protection Regulation (GDPR). The requirement of *data protection by design* have been formally embedded in Article 25 of the GDPR, which requires the data controllers to design and develop products with a built-in ability to demonstrate compliance towards the data protection obligations. GDPR is particularly oriented towards service-oriented systems, which involves processing of personal information by an actor(s) in such systems and communication between the actors and the users of the system. The overall goal is to look at analysis of GDPR-related privacy compliance in a language setting where these two aspects are highlighted.

This thesis studies specification and enforcement of security and privacy requirements in distributed systems, more specifically through language-based techniques, considering an object-oriented setting, in particular, the active object paradigm. The protection of personal information involves two aspects: information security and data protection. For security awareness, the language is enriched with secrecy constructs such as secrecy levels, and a notion of an interaction non-interference policy is defined, which is followed by a formalization of a secrecy type system and static trace analysis. This approach restricts information access based on confidentiality levels. However, to regulate the access to a narrower set of the authorized uses with respect to requirements that are integral to the data protection such as purpose, privacy by design and privacy by default, data subject access rights, consent etc, several key data protection aspects are developed and embedded into the language. In particular, we use a combination of static and dynamic approaches for their specification and enforcement.

Preface

This thesis is submitted in partial fulfilment of the requirements for the degree of *Philosophiae Doctor* at the University of Oslo. The research presented here is conducted under the supervision of Professor Olaf Owe, Professor Martin Steffen, and Postdoctoral Fellow Toktam Ramezanifarkhani.

The thesis is a collection of four papers, presented in chronological order. The common theme to them is a LaTeX thesis template. The papers are preceded by an introductory chapter that relates them together and provides background information and motivation for the work. All papers are joint work with supervisors.

Acknowledgements

The true delight is in the finding out rather than in the knowing.
-Isaac Asimov

I thank my supervisor, Olaf Owe, for hours of insightful discussions and for teaching me to think and write scientifically. I could not thank him enough for believing in me and supporting me in what I wanted to do. I have been very fortunate to have collaborated directly with him. I vividly remember long hours of arguments and discussions on improving the formalisms (and a lot of just staring at the type rules on the screen with mutual skepticism). Just by observing his dedication and passion for work have kept me on track. Many fond memories of receiving his comments and feedbacks from metro, flights, airports. We survived two deadlines communicating through in-flight WiFi. Humility and infinite patience are two of his traits that stands out for me. This research work and dissertation would never exist without him. I am thankful to my co-supervisor, Martin Steffen for his detailed comments and constructive criticism on papers drafts, presentations, and thesis draft. Throughout these four years, he was always ready to answer my often silly questions. His objective thinking, suggestions, and guidance have been invaluable, and have immensely contributed to clarity of the thesis. I am thankful to my second co-supervisor, Toktam Ramezanifarkhani for sharing her expertise in the field of security and information flow control. I have learnt a great deal from her, and i would to thank her for helping me understand the area better. Just by observing her, I have tried to learn to maintain a healthy work-life balance.

PhD would have been colorless without my dear friends: Shareq, Divya, Shannu, Sudeepta, Hogne, Lykke, Anne, Michelle, Gaurav, Harish, Pallavi, Chinmayi, Isaac, Vasileios, Kamer, Elahe, Hamed, Florian, Ijlal, Tim, Anya, Babita, Meghna, Vatsal, Lazlo, Laura, Magdalena, Karan, Raja, I am fortunate to have you as my friends. Am thankful to Daniel for his support and guidance in most critical times during PhD. Being new to formal methods, fear of working in the unknown field, and feeling utterly useless, wouldn't have sailed through without your encouragement and (in particular) the "snail story". Grateful to have met Fabrice, two novice having long conversations on spirituality and futility of words. Antonio is the one that I missed the most in final year of PhD, thank you for all the chocolates, fudges, cakes, endless laughter and (most importantly) bearing with me. I am thankful to Konstantin for joining in for sharing the joys of small walks to get some sun, coffee, chai latte, and spontaneous eating out plans. I am always grateful to Fahri, thank you for always being around and guiding me to the right path.

It is impossible to thank my parents enough for their unconditional love and for their sacrifices for my upbringing and education. Everything I do is for you. I am thankful to my brothers for their unconditional love and support, their trust in me, and of-course sometimes for much needed distractions.

Shukun Tokas
Oslo, February 2021

Contents

Preface iii

Contents v

1 Introduction 1
1.1 Motivation 1
1.2 Research questions 2
1.3 Methodology 4
1.4 Outline 4

2 GDPR related data protection principles and research focus in GDPR 5
2.1 Personal data 6
2.2 Data subject 6
2.3 Data controller and data processor 6
2.4 Privacy policy and privacy notice 7
2.5 Our research focus in GDPR 7
2.6 Principles relating to processing of personal data (Article 5) 7
2.7 Lawfulness of processing (Article 6) 9
2.8 Right of access by the data subject (Article 15) 9
2.9 Data protection by design and by default (Article 25) 10

3 Language-based approach to privacy specification and enforcement 11
3.1 An imperative programming language 11
3.2 Operational semantics 12
3.3 Type and effect analysis 13
3.4 Soundness 13
3.5 Static enforcement 14
3.6 Runtime enforcement 16

4 Overview of the research papers 17
4.1 Paper I : A secrecy-preserving language for distributed and object-oriented systems 17
4.2 Paper II : Language-based mechanisms for privacy by design 18
4.3 Paper III : Static checking of GDPR-related privacy compliance for object-oriented distributed systems 18
4.4 Paper IV : A formal framework for consent management 19
4.5 Additional papers 20

5 Discussion **21**
5.1 Summary of the contributions . . . 21
5.2 Limitations and future work . . . 24

Papers **28**

6 A secrecy-preserving language for distributed and object-oriented systems **29**
6.1 Introduction . . . 30
6.2 Behavior of object-oriented distributed systems . . . 33
6.3 Interaction non-interference . . . 37
6.4 The SeCreol language . . . 39
6.5 Secrecy-type system . . . 45
6.6 Network level leakage . . . 51
6.7 Theoretical results . . . 61
6.8 Operational semantics . . . 63
6.9 Related work . . . 68
6.10 Conclusion . . . 70

7 Language-based mechanisms for privacy by design **73**
7.1 Introduction . . . 73
7.2 Language constructs for policy specification . . . 75
7.3 Embedding policy with program constructs . . . 80
7.4 An effect system for privacy . . . 82
7.5 Case study . . . 84
7.6 Related work . . . 86
7.7 Conclusion . . . 88

8 Static checking of GDPR-related privacy compliance for object-oriented distributed systems **89**
8.1 Introduction . . . 90
8.2 Relevance to the GDPR and research focus . . . 92
8.3 Formalization of static privacy policies and policy compliance 94
8.4 An imperative programming language . . . 106
8.5 An effect system for privacy . . . 111
8.6 Awareness of subject . . . 123
8.7 Operational semantics . . . 125
8.8 Related work . . . 134
8.9 Conclusion . . . 137

9 A formal framework for consent management **139**
9.1 Introduction . . . 139
9.2 Language setting . . . 141
9.3 Consent management . . . 145
9.4 Runtime system . . . 147
9.5 Related work . . . 153

9.6 Conclusion 155

Bibliography **157**

Chapter 1

Introduction

1.1 Motivation

Rapid progress and adoption of ICT in day to day market solutions results in collection and processing of personal information for transactions for commercial goods or public services. Although these developments offer significant economic advantages, they also have adverse impact on the privacy of individuals. It is therefore important to have a meaningful balance between innovation, economic growth, and fundamental privacy rights. In response to the emerging privacy concerns, the European Union Parliament has approved the General Data Protection Regulation (GDPR) [41] to strengthen and impose data protection across the European Union (EU) and the European Economic Area (EEA). Moreover, the requirements of *data protection by design* and *data protection by default* have been formally embedded in Article 25 of the GDPR, which requires the individuals and organizations that process personal data of EU citizens or provide services in EU, to design and develop products with a built-in ability to demonstrate compliance towards the data protection obligations. On similar grounds, Bennette and Raab in [13] mention that "if one accepts, however, that at least part of the privacy problem is caused by the properties inherent in the design of certain information technologies, then it follows that the same technologies can be effectively shaped to protect privacy, rather than to invade it". This principle can be approached by considering *privacy* as a primary concern rather than a secondary one, i.e., by building privacy into the applications.

The GDPR is particularly oriented towards service-oriented systems, which involves processing of personal information by the actors in the system as well as the communication between the actors and the external users. In particular, we are interested in a framework where these two main aspects can be studied at a high level of abstraction and with a simple and modular semantics, so that the security and privacy type and effect system can be defined with relatively fewer and simple rules. In all papers that contributes to this dissertation, we target distributed, object-oriented and service-oriented systems and consider a general concurrency model for distributed systems, based on concurrent objects communicating by asynchronous methods, often called the active object model.

The main theme for the thesis is *information privacy*. However, privacy or data protection is a wide area, ranging from privacy-enhancing technologies, networks, hardware, to support through programming languages. This work aims to contribute in privacy engineering by providing concepts and constructs to capture and formally verify critical aspects of data protection that facilitates the design of systems that are privacy-compliant by construction. In particular, we use formal language-based approaches with static analysis to enforce security

and privacy requirements.

Traditionally, security is defined in terms of the CIA triad, i.e., the three quality attributes: confidentiality, integrity, and availability. In this thesis, a security policy (henceforth "security level") refers to the *confidentiality* attribute, meaning that information with a *high* secrecy level is only accessible to users tagged with *high* secrecy level. The two secrecy levels, denoted *high* and *low*, encode two confidentiality levels, and the so called *non-interference policy* is an information flow policy that enforces access restrictions on the use of confidential data and information derived from the data. This approach restricts access to the confidential information based on secrecy levels. We extend this to regulate the access to the confidential information with respect to regulatory data protection requirements such as purpose limitation, consent. The privacy relevant information need to be enforced to impose such restrictions. We propose a policy specification language for specifying purpose, access right, policy, and consent. A notion of static and runtime policy *compliance* is formalized to compare policies. The syntax and semantics of the core language is extended with privacy specifications and policy compliance. Privacy compliance is established by static and runtime analysis. Overall, the approach provides means to make legal requirements into tangible and measurable compliance control.

The thesis proposes several interfaces and classes needed to design and build solutions with privacy in mind. For example, the interfaces *Sensitive*, *Principal*, *Subject*. We believe that "Data subject" is a key concept in the GDPR and it is the data subject that benefits from privacy. So this notion, if taken into account in designing systems, may contribute greatly to privacy protection. The privacy-by-design approach is presented in the papers included in the dissertation shows that i) processing of information is restricted by stated policies, i) a transparent execution of *Right of Access* request, i.e., there is no hiding (in system) from the data subject, and ii) a direct subject-system interaction for managing the subject's privacy preferences.

This work is a step towards integrating data protection requirements and system's functional requirements, and addressing them using a formal approach.

1.2 Research questions

The main research goal of the thesis is to develop language features that allow us to give precise meaning to privacy related notions, and leverage program analysis for their enforcement.

The papers included in this thesis address this challenge, by choosing language mechanisms that are useful for modeling service-oriented distributed systems and modular system analysis. In particular, we consider a general concurrency model [61] based on the active object paradigm [16] using asynchronous method calls as the only interaction mechanisms. This captures the main mechanisms for structured and efficient communication in service-oriented systems. This concurrency model is combining the Actor model [57] with object-oriented concepts. The considered language has a compositional semantics, which is

beneficial to analysis.

The overall goal is as follows :

> *To formalize and explore a suitable high-level language for modeling of service-oriented distributed systems, extend the* syntax *and the* semantics *of the language with security and privacy-related notions, and to develop analysis techniques for the extended language ensuring security and privacy properties.*

To achieve this goal, this thesis will address the following specific research questions:

1. *RQ1:* How to formalize and enforce *confidentiality* properties and policies?
2. *RQ2:* How to formalize privacy requirement specifications?
3. *RQ3:* How to define notions of *static* and *run-time* privacy compliance?
4. *RQ4:* How to enforce and check enforcement of *static* privacy compliance?
5. *RQ5:* How to enforce and check a policy compliant access and support dynamic consent changes at *runtime*?

This thesis includes the following research publications, which describe research results in light of these research questions:

1. **Paper I** : A Secrecy-Preserving Language for Distributed and Object-Oriented Systems
   Authors: Toktam Ramezanifarkhani, Olaf Owe and Shukun Tokas.
   Publication: The Journal of Logic and Algebraic Programming, 2018 [93].

2. **Paper II** : Language-Based Mechanisms for Privacy by Design
   Authors: Shukun Tokas, Olaf Owe and Toktam Ramezanifarkhani.
   Publication: Proceedings of the 14th IFIP International Summer School on Privacy and Identity Management (2019) [109].

3. **Paper III** : Static Checking of GDPR-Related Privacy Compliance for Object-Oriented Distributed Systems
   Authors: Shukun Tokas, Olaf Owe and Toktam Ramezanifarkhani.
   Publication: Submitted to the Journal of Logic and Algebraic Programming, in first round revision [110].

4. **Paper IV** : A Formal Framework for Consent Management
   Authors: Shukun Tokas and Olaf Owe.
   Publication: Proceedings of the 40th International Conference on Formal

Techniques for Distributed Objects, Components, and Systems, FORTE 2020 [107].

1.3 Methodology

To answer these research questions, the methodology used in the thesis employs a combination of static and dynamic approaches. We define suitable languages augmented with awareness of security and privacy aspects. The static approach involves: Devise a semantics for the programming language to specify a notion of security and privacy policies; Formulate static analysis (of interaction non-interference and policy compliance, respectively) as type and effect systems; Define small-step operational semantics for the programming language; Establish consistency of the type and effect system with respect to the operational semantics. For dynamic checking and enforcement of privacy compliance, the runtime system is extended with runtime checks. We have made a theoretical evaluation of the static and dynamic checks by means of soundness theorems, and have demonstrated the suitability of the approach by examples and case studies.

1.4 Outline

The rest of the introductory part of the dissertation gives a general background on the underlying programming language family and data protection notions, and sets forth the context of the work presented in the research papers in Part II of the thesis. *Chapter* 6 review concepts of security, privacy, the GDPR and our research focus in GDPR domain. *Chapter* 7 introduces the programming model and discusses its underlying concepts. *Chapter* 8 gives a short summary of each of the research paper included in the dissertation. *Chapter* 9 concludes the introductory part with a discussion on research contribution with respect to the research questions and suggests possible directions for future work.

Chapter 2

GDPR related data protection principles and research focus in GDPR

The significance of *privacy* is reflected by the fact that the fundamental documents that define human rights (such as Universal Declaration of Human Rights, the Organization for Economic Co-operation and Development guidelines, the European Convention on Human Rights, etc.) incorporate reference to privacy or privacy related concepts [24]. Interpreted broadly, privacy has a rich history in law and philosophy, and many definitions attempting to define privacy consider one or more distinct perspectives on privacy [17]. In [24], Clarke presents comprehensive interpretation of privacy as *about the integrity of the individual*. Furthermore, privacy is also said to encompass different perspectives: privacy of the person, privacy of personal behaviour, privacy of personal communication, and privacy of personal data. In [103], Sieghart promotes privacy in terms of ensuring that "the right data are used by the right people for the right purposes". The *right data* requires the information to be accurate, complete, relevant and timely. The *right purpose* requires that the purposes are expressly or implicitly agreed by the data subject or are sanctioned by the law. The *right people* are the entities that will use the data for only those purposes. Absence of any of these conditions may jeopardize critical rights, interests, and services [103]. These definitions of privacy are still valid in current times.

Privacy is considered as a fundamental human right [40] in the European Union, giving the right to a private life and associated freedoms (such as in control of information about yourself) to its citizens. Lately, a new term is introduced in the privacy literature, namely *data protection*. In the context of the GDPR, privacy and data protection are connected. However the term data protection means something more specific. Data protection is about concepts such as specified purposes, legitimate basis of processing personal information, protection of personally identifiable information, etc. "Privacy is recognized as a universal human right while data protection is not – at least not yet" [39]. Furthermore, the notion of data protection derives from the *right to privacy* and is instrumental in preserving and promoting fundamental values and rights [39].

The rest of this chapter gives a description of GDPR's core data protection concepts relevant for the thesis.

2.1 Personal data

The concept of *personal data* is central to data protection and its definition in the GDPR is kept intensionally broad. Article 4(1) of the GDPR defines personal data as “any information relating to an identified or identifiable natural person” [41]. Example of general personal data include date of birth, gender, marital status, citizenship, association with organizations, address, phone number, identity verification information. This also includes information such as dynamic IP addresses and cookies, as this information can be used to track online activities and generate a user profile which can be linked to devices and in most cases, an individual [114]. Only personal information is subject to the regulation.

The regulation identifies certain kinds of personal data, i.e., the information that their processing could create significant risks to data subject’s fundamental rights. Genetic data, biometric data, data concerning health, personal data revealing philosophical beliefs or ethnic origin, etc. are considered as *sensitive personal data*.

In our work, we do not differentiate between personal data and sensitive data. In several places, we use the terms *sensitive data* and *personal data* interchangeably.

2.2 Data subject

The concept of data subject is a key concept in data protection as it is the data subject that benefits from privacy. However, surprisingly it is not defined explicitly in the GDPR. Instead, it is defined parenthetically within the definition of personal data, as *an identified or identifiable natural person* as being a data subject [114]. In particular, the data subject is the individual about whom or from whom the information is being collected and processed.

2.3 Data controller and data processor

A *data controller* is a natural person, organization, public authority, or agency, which collects information about data subjects, determines the purposes of processing personal information, and processes the information (including its storage, disclosure). A *data processor* is a natural person, organization, public authority, or agency, that processes personal data on behalf of the data controller, which essentially means that a data processor is simply a service provider for a data processor [114]. The data controllers are the ones that exercise the decisions about collection, disclosure, processing, retention and destruction of personal data. As a result, a data controller is responsible for most of the compliance requirements (Article 5(1)). Through Article 24 and Article 25 of the GDPR, the requirements of integrating necessary safeguards into processing of personal information are imposed on the data controller.

The solution proposed in this dissertation demonstrates how such requirements can be addressed by the data controller, using a language-based approach focused on compliance-by-design (a.k.a. compliance-by-construction).

2.4 Privacy policy and privacy notice

In general, there are two types of documents that communicate privacy practices: a privacy policy and a privacy notice. A *privacy policy* is an internal document addressed to employees accessing personal information, clearly stating how the personal information will be collected, stored and disclosed to meet the organizational/regulatory privacy requirements. A *privacy notice* is an external document and a transparent notification that is addressed to data subjects that describes how their personal information is being handled, including information on the legal basis of processing and specific legitimate interests pursued by the data controller. In principle, the privacy notices should be in alignment with the privacy policies.

The work presented in this dissertation is mostly focused towards privacy policies, it is hinted in the papers that it is necessary that the policy terminology used towards the data subjects should be with a formal connection to the underlying programming elements.

2.5 Our research focus in GDPR

The GDPR contains 99 articles covering quite diverse aspects of privacy such as data protection principles, accountability, data protection impact assessment, certification, penalties, etc. In this dissertation, we have focused on (mostly) static and runtime analysis of privacy compliance on the intersection of Article 5, Article 6, Article 15 and Article 25. However, we do not claim to cover these articles thoroughly; our main focus is on those aspects that are susceptible to formalization from the language perspective. In particular, we use a combination of static and dynamic approaches, for the enforcement of initial policies (by system designers) and redefined policies (by data subjects), respectively. Figure 8.1 illustrates our research focus in the GDPR.

2.6 Principles relating to processing of personal data (Article 5)

The GDPR's processing principles, as set out in Article 5, are required to be followed by the entities responsible for processing personal data. In fact the data controllers are prescribed with the duty to demonstrate compliance with processing principles. The principles are broadly interpreted, but their violators may incur large administrative fines. The processing principles are: lawfulness, fairness, and transparency of processing; purpose limitation; data minimization; accuracy; storage limitation; integrity and confidentiality; and accountability.

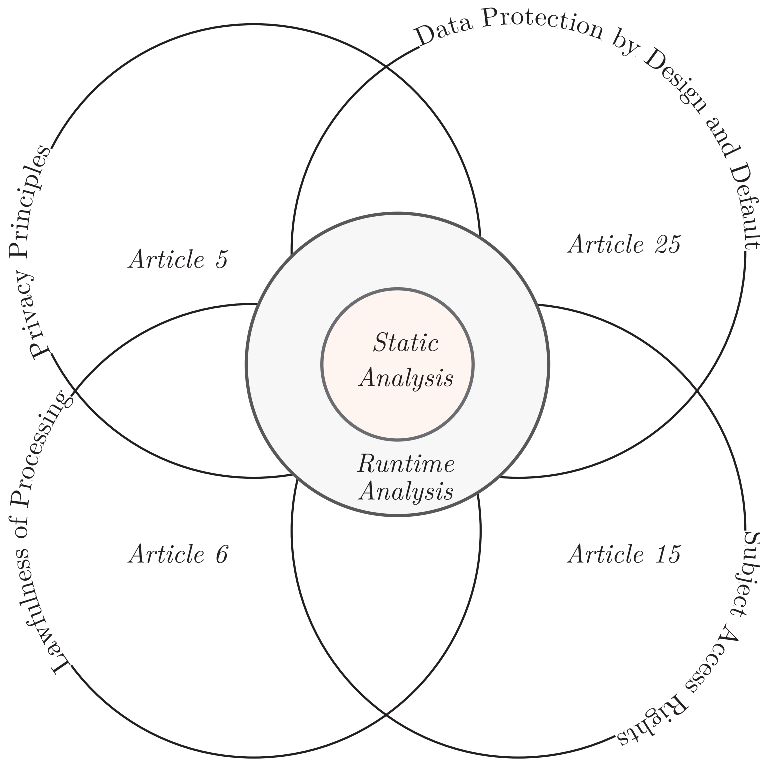

Figure 2.1: Our research focus in the context of GDPR

In this thesis, we have focused only on:

1. *Lawfulness, fairness, and transparency of processing*, which requires honest usage and communication with the data subject about their personal data. The three components here are linked with one another, and requires that the controllers are open and clear towards data subjects.

2. *Purpose limitation* requires the collection and processing of personal data for specific, explicit and legitimate purposes only. To determine if the personal data could be used for secondary purposes (i.e., purposes for which information was not collected in first place), the GDPR provides guidelines to assess the compatibility of secondary purpose with the original purpose (but the discussion is not so relevant here).

3. *Confidentiality and integrity* require protection of personal data against unauthorized processing. This regulation promotes the use of techniques such as pseudonymisation, implementing an information security framework, etc.

4. *Accountability* means that the organizations process personal data responsibly by complying with the different obligations and need to show and evidence their compliance.

2.7 Lawfulness of processing (Article 6)

Lawfulness means the personal data must only be processed for valid legal grounds of processing. Following are the six lawful grounds of processing for the controllers: contractual necessity, consent, legal obligation, vital interests, public interests and legitimate interests. The processing must be carried out within the limit of the applicable processing grounds. *Consent* is the lawful ground that reflects a data subject's agreement and provides the data controller with permission to process a subject's personal data for a *specific purpose*. Article 7 sets out the conditions for processing when relying on consent, and requires the controller to keep records of consent as they may be obligated to demonstrate that it was obtained in first place. Additionally, if a controller relies on consent as a lawful ground for processing then the data subject must be able to withdraw his or her consent.

The solution proposed in this dissertation, in *Chapter* 9 in particular, demonstrates how these consent based requirements can be addressed at runtime using predefined interfaces between the system and the data subject.

2.8 Right of access by the data subject (Article 15)

The regulation prescribes that the data subject has Right of Access, which requires the data controllers to provide any data subject (that requests to know) with his or her personal data, the purposes of processing, the legal basis for doing so, recipients of data when personal information has been or will be disclosed, and more such relevant information. Transparency is a key concept in the GDPR and is promoted to assure the right to privacy for the individuals, by properly informing the individuals how their personal data is processed. In practice, processing such requests are likely to pose a substantial administrative burden, so the organizations should take into consideration having processes in place to assist with the task [114]. WP29 [1] recommends controllers to introduce tools, such as a privacy dashboards through which the data subject can be informed and engaged regarding the processing of their personal data [5].

The solution proposed in this dissertation demonstrates how these requirements can be addressed at compile time and at runtime. *Chapter* 8 presents an approach to handle static awareness of a data subject, by using predefined interfaces and types, and a default policy to give each subject read access to personal data about himself and herself. *Chapter* 9 presents an approach to handle data subject access request by using predefined interfaces between the

[1] WP29 is an abbreviation for Article 29 Working Party, which was an advisory body composed of representatives of the national data protection authority of each EU member state, the European Data Protection Supervisor and the European Commission.

system and the data subject, and a broadcast method call that enables collection of personal data from all objects processing information about the given subject.

2.9 Data protection by design and by default (Article 25)

Article 25 introduces *data protection by design* and *data protection by default* obligations on data controllers. The data protection by design requirement requires the controller to *implement appropriate technical and organizational measures* [41], but it is not clear how such measures can be translated and *embedded* into the design. However, this generality gives the designers the flexibility while still strategically promoting privacy (as integral to design) and data protection compliance as forethought in the development of product, services, or technologies.

In addition, it introduces a specific *data protection by default* obligation. This requires that an organization may only process the personal data that is necessary for the fulfillment of stated purposes. In practice, the by-default setting could mean that the strictest privacy settings apply automatically when the subject acquires a new product or service [114]. "It is about how to build things that people can *trust*" [27]. For static checking of compliance, when the lawful basis of processing of personal information is the performance of the contract or other valid bases but not the consent, the policies should be formulated in a way that ensures that they are built into the system by default, i.e., no measures are required by the data subject in order to maintain his or her privacy. This is demonstrated in *Chapter* 7 and *Chapter* 8. In our work, when consent is the basis of processing, the choices of the data subjects are captured at runtime as outlined in *Chapter* 9.

Chapter 3

Language-based approach to privacy specification and enforcement

In this chapter, we present an overview of the syntax and semantics of the core language on which our work is based, in Section 9.2.2 and Section 3.2. The methodology used in the thesis employs a combination of compile time and runtime approaches. Section 3.3 present a general discussion on type and effect systems as a well known approach to program analysis. Section 3.4 presents a general discussion on soundness of the analysis. Section 3.5 and 3.6 presents a discussion on static enforcement and on the use of runtime information for runtime enforcement.

3.1 An imperative programming language

The programming model is chosen in such a way that allows us to focus on major challenges concerning specification and analysis of security and privacy properties in modern service-oriented systems, without the complications related to shared variables and low-level synchronization mechanisms such as explicit signaling and notification.

In order to study the security and privacy properties at a high level of abstraction, we consider a small imperative, high-level language supporting the active object programming paradigm [16], based on the actor model [57]. In the actor formalism, a distributed system is seen as a collection of concurrent entities (called *actor*) that are isolated from each other and communicate via message passing only. Languages that integrate the actor model with object-oriented concept, are often called active object languages [16]. The active object paradigm is based on concurrent autonomous objects and offers both synchronous and asynchronous communication. We will consider a small but expressive language based on the active object concurrency model.

The language studied in this thesis supports interface abstraction, i.e., an object can only be accessed through an interface and remote field access is illegal. This enables us to precisely articulate the interface between an object and its environment [118]. Object communication is only by asynchronous method calls, implemented by means of asynchronous message passing. The language provides non-blocking method calls, to avoid undesirable waiting in a distributed setting. By means of a suspension mechanism, incomplete method invocations in an object may be placed on the object's process queue. The process will be enabled when the object receive the response.

The considered language has a compositional semantics, as presented in [61, 93], which is beneficial to the analysis. The language is imperative and strongly typed, using data types for defining data structures locally inside a class. The data type sublanguage is side-effect-free. The programs we consider are defined by a sequence of declarations of interfaces (containing method declarations), classes (containing class parameters, fields, methods and class constructors), and data type definitions.

For this language setting, we present two different formalizations of static type and effect analyses (*Chapter* 6 and *Chapter* 7, 8) and a runtime analysis that uses static and runtime information (*Chapter* 9), covering different security and privacy goals of this dissertation.

3.2 Operational semantics

Semantics is the mapping of syntax to meaning. Formal semantics may serve as the basis for static and dynamic analysis. In general, there are three main categories of formal semantics: operational semantics, denotational semantics, and axiomatic semantics. The idea behind *operational semantics* is to give a precise description and meaning of a statement or program by specifying the effects of running it on a machine [101]. The effect in an imperative setting is a sequence of changes in its object-state, where an object-state is mapping from variable to values. The intended behaviour of each statement is given in rigorous mathematical terms. Moreover, there are sub-categories among the operational semantics: *natural operational semantics*, in which the rules are used to determine the final result of the execution of a program, and the *small-steps operational semantics* [92], in which the goal is in determining the precise meaning of a program by examining the complete sequence of the state changes for a program execution. *Denotational semantics* is an alternative to operational semantics, in which each language construct is defined by a mathematical entity and a function that maps constructs of the language onto instances of the mathematical entity [101].

Thus the meaning of a program construct denotes a mathematical object, and the aim of denotational semantics is to construct such mathematical objects denoting the meaning of their corresponding syntactic constructs. *Axiomatic semantics* is based on mathematical logic, and specifies what can be proven about the program [101]. The meaning of a statement is defined by the statement's effect on assertions about the data affected by the statement.

The goal of this dissertation is to find potential non-compliant accesses to sensitive information. We check at compile-time whether a program will possibly run into a state where sensitive information is not accessed as per security or privacy specifications. For this purpose, we chose to use *structural operational semantics* because it is structural (compositional) and intensional [92]. It allows interpretation and prototyping of programs provided the operational semantics is executable. It is compositional because an object proceeds with a computation step if and only if a phrase within the object can make the step, and it is

intensional because the derivation tree drawn to take a computational step contains all intermediate branches that generated the step [72]. In particular, this style enables us to reason about intermediate stages in a program execution and allows to deal with non-terminating programs [83]. These formal descriptions serves as a semantically sound basis for analysis of the proposed static analysis. A compositional semantics is used which facilitates proof by structural induction on program syntax. It is also needed for scalability of analyses techniques.

The possible execution traces for the extended languages, proposed in *Chapter* 6, *Chapter* 8, and *Chapter* 9, are defined by the semantics as a structural operational semantics, covering different security and privacy goals of this dissertation. *Chapter* 6 use an environment with a mapping, binding variable names to confidentiality levels, and the rules in *Chapter* 8 use an environment with a mapping from program variable names to policy sets, while the rules in *Chapter* 9 use an environment with a mapping from variable names to tagged values.

3.3 Type and effect analysis

A program variable can take a range of values during a program execution [21], and the range is defined by the *type* of the variable [21]. A *type system* is a component of the programming language that comprises of a set of rules that assigns types to program constructs such as variables or expressions. The type rules take a general form: a type is associated with a program construct relative to a type environment that provide the type for each free variable [83], and other declared entities such as functions, methods, classes and interfaces. An effect system extends the notion of type, the typing rules take a general form: a type and an effect associated with a program construct relative to a type environment. Conceptually, an effect describes intensional information about what takes place during program evaluation [83], such as changes in the typing environment.

The process of checking for type errors in a program at compile time is called *type checking*. Type checking is one of most widely used forms of static program analysis. A program that successfully passes the type checking is said to be *well-typed*, which means that all executions of the program are guaranteed to cause only allowed executions, i.e, executions where the variables have values within the declared types. *Ill-typed* programs mean that it could not be guaranteed to be well-behaved [83], for example assigning boolean values to object variables.

A natural extension of type checking techniques is to enrich the types with *annotations*. In this dissertation, we annotate the types with *high* and *low* secrecy levels (in *Chapter* 6), and with privacy policy and policy sets (in *Chapter* 7, 8). The effects changes these annotations.

3.4 Soundness

Once a type system and a semantics are formalized, we can prove *type soundness*. Type soundness is a basic property of a type system stating that programs

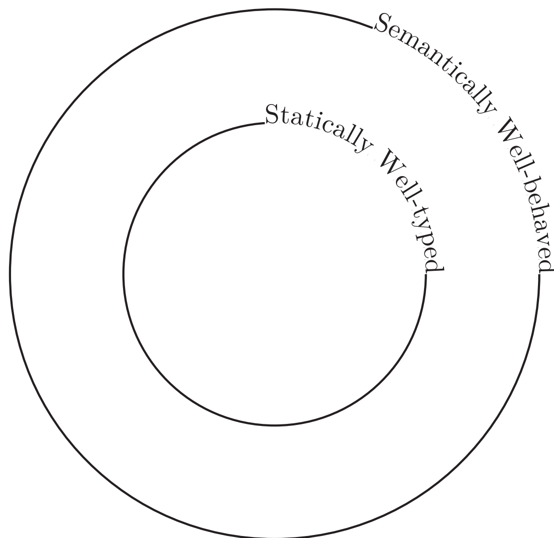

Figure 3.1: Static enforcement

do not cause forbidden errors, i.e., well-typed programs are well behaved [21]. There are two different systems operating here: *(i)* a *type checker* that checks the program and never consider a program execution (like runtime values) in detail, and *(ii)* an *operational semantics* that describes how a program actually runs dynamically. The conformance of the two systems is discussed by means of soundness, i.e., a type system is sound with respect to the operational semantics. A soundness theorem says that if we have an expression e with a type T, i.e., e_T, and when e is executed it produces a value v, then v also have a type T, i.e., v_T. The connection is made from both sides, the static side where e has a type T and the dynamic side where e is going to produce a value with a type T [67]. Then we can safely say that the static type checker was able to correctly predict the type of the value, without running the program. The soundness proof is carried out in inductive manner.

In the dissertation, we proved soundness of the network analysis (*Chapter* 6) and compliance analysis (*Chapter* 7), with respect to the respective operational semantics.

3.5 Static enforcement

The static type checker in our work ensures that an information with higher policy level cannot be accessed from a lower policy level context and that a variable with lower policy type may not be assigned information with higher policy type. Type systems are given for static enforcement, based on security and privacy policies outlined in the papers. Static enforcement means enforcing polices on static entities in a way that does not depend on concrete runtime

information. In *Chapter* 6, static secrecy levels are types that are associated with variables and parameters, and its enforcement is built on the approach introduced in [116]. In *Chapter* 7 and *Chapter* 8, types and methods are annotated with privacy policies. Each command in the program is analyzed to ensure that its execution causes allowed flow of information with respect to the security lattice (*Chapter* 6) and lattice over sets of policies (in *Chapter* 7 and *Chapter* 8).

Static checking over-approximates the security and privacy restrictions, i.e., it ensures that the system is *safe*, meaning that it accepts programs that surely satisfy our security/privacy property, while rejecting programs that might still be safe. Consider a statement:

$$\texttt{if}\ m(u_{policyL}) \neq n(u_{policyL})$$

$$\texttt{then}\ u_{policyL} := v_{policyH}\ \texttt{else}\ u_{policyL} := u_{policyL}\ \texttt{fi}$$

where m and n are semantically equivalent functions ($m(x) = n(x)$ for all values of x), u is a variable with a low policy (*policyL*) and v is a variable with a high policy (*policyH*), such that $policyL \sqsubseteq policyH$. The sub-policy relation $\sqsubseteq$ expresses *policy compliance*, meaning that a less restrictive policy (*policyL*) complies with a more restrictive policy (*policyH*). A detailed description of the policy compliance can be found in *Chapter* 7. The static policy of a variable is indicated by a subscript, i.e., $u_{policyL}$. The expression $m(u_{policyL}) \neq n(u_{policyL})$ will always be false, and $u_{policyL} := u_{policyL}$ is a valid assignment, which means that the statement causes only allowed executions. However, the fact that m and n are equivalent cannot (in general) be statically checked. In addition, in static analysis both the branches of the **if** statement are analyzed. This will lead to rejection of the program by static analysis. In particular, the static analysis over-approximates the privacy and security restrictions by ensuring that ill-typed programs are not accepted, while it might reject the programs that are actually well-behaved at runtime. All well-typed programs are semantically well-behaved, but not all semantically correct program are statically compliant. This is depicted in Figure 3.1. In general, static analysis for checking some undesired issues of a program gives 3 possible outcomes: *yes* meaning that there is not an issue, *no* meaning that there is an issue, and *maybe* meaning that there maybe an issue. There are two ways of grouping the results: yes versus no/maybe, and yes/maybe versus no. The first approach is good when the maybe cases are few, which is adopted in *Chapter* 6. The second approach is appropriate when the maybe cases are many (as is the case with compliance w.r.t. consent), and one can then use runtime checking to detect the bad programs. In this case, runtime testing is needed to check the maybe programs; however, the number of runtime errors is reduced since programs classified as “no” do not pass the static test. We use a combination of the two approaches for different aspects of program behavior.

In this thesis we have limited our work on static program analysis to fully automated methods. We have therefore not considered program logics such as Hoare Logic for proving partial correctness, since they in general depend on user interaction, for instance in the specification of loop invariants. However,

in order to ensure that there is no information leakage at the network level, by attackers that may observe aspects of the communication interaction between active objects, we have devised a simplified Hoare Logic. This logic is dealing with trace expressions rather than state predicates, and invariant-like trace expressions are generated automatically. The logic for detecting possible leakage through observation of communication traces can be applied without user interaction. This is part of the work in *Chapter* 6.

3.6 Runtime enforcement

Some information becomes known only at runtime, such as changes in policies or the policy on newly created data. Dynamic enforcement means to enforce polices through the use of information available at runtime. This is done in *Chapter* 9, which includes tagging of values with privacy tags at runtime and checking that the program execution causes allowed flow of information with respect to the dynamically changing privacy policies. Coming back to the example given in Section 3.5, the runtime system will allow execution of the statement, since the **then** branch is never reached, because $m(u_{policyL}) \neq n(u_{policyL})$ is false.

In particular, this thesis combines the two methods by detecting the static security and privacy violations by an extended type and effect system, and enhancing permissiveness of the static analysis by using runtime information, in particular privacy issues that are not guaranteed by the static checking.

Chapter 4

Overview of the research papers

This chapter presents a short summary of the research papers in Part II of this dissertation. The full contents of the papers appear as in their original publications, but have been reformatted to fit the dissertation structure.

4.1 Paper I : A secrecy-preserving language for distributed and object-oriented systems

Authors: Toktam Ramezanifarkhani, Olaf Owe and Shukun Tokas
Publication: The Journal of Logic and Algebraic Programming, 2018 [93]
Summary: It is often necessary to distinguish between confidential (high-level) and non-confidential (low-level) information, and ensure that confidential information is only accessible to authorized users. This paper studies specification and enforcement of an information flow policy within programs to protect the confidentiality of information. Secure information flows are often expressed by semantic models of program execution in the form of a *non-interference policy*.

To formalize the approach, we consider a high-level core language based on the chosen concurrency model, namely the paradigm of so-called *active objects* using asynchronous method calls as the only interaction mechanism, thereby combining the Actor model and object-orientation. The paper presents a core language called *SeCreol*, supporting the active object paradigm with cooperative scheduling. The communication mechanisms are extended with the awareness of secrecy levels as well as secrecy type information.

Due to the non-deterministic nature of objects in this setting and the non-trivial implicit information flow leakage related to the observation of communication pattern, standard definitions of non-interference are not suitable in this setting. A notion of *interaction non-interference* is defined, which is tailored to this language setting. Interaction non-interference stipulates indistinguishability of interactions, i.e., program executions to be equivalent in the view of attackers observing method call events. Interaction non-interference property is enforced by two kinds of static analysis: *i)* a secrecy type system and *ii)* trace analysis system, in inter-object and network level communication, respectively. The approach is illustrated with several versions of an example from a news subscription service. We prove that interaction non-interference is satisfied by the combination of these analysis techniques. Thus any deviation from the policy caused by implicit information leakage visible through observation of network communication patterns, can be detected.

4.2 Paper II : Language-based mechanisms for privacy by design

Authors: Shukun Tokas, Olaf Owe and Toktam Ramezanifarkhani.
Publication: Proceedings of the 14th IFIP International Summer School on Privacy and Identity Management (2019) [109].
Summary: There is a growing demand for verifiable privacy compliance in order to produce evidence for regulatory requirements such as *privacy by design*, *accountability*, *transparency* etc. This paper presents a formal language based approach to bake privacy into the design of systems and verify if the design satisfies the privacy specification. In particular, the paper focuses on static privacy compliance, i.e., privacy notions that can be expressed and checked statically.

To express privacy policies we define a policy specification language. In this setting, a privacy policy is a statement that expresses permitted use of the personal information by the declared program entities. A policy is given by a triple of the form (*principal*, *purpose*, *access-right*). A policy set is given by set of such policy triples. A *principal* is given by an Interface and these interfaces are organized in an open-ended inheritance hierarchy. *Purposes* and *access-rights* are organized in a directed-acyclic graph and a lattice, respectively. A formal notion of (static) *policy compliance* is defined to compare policies, in order to make access decisions and enforce appropriate use of personal data.

To embed the privacy compliance with a core modelling language [61], the syntax and semantics of the language is extended with policy specification and policy compliance. In particular, we formulate a type-based static analysis using a set of syntax-directed rules. The rules are defined with respect to the *policies* and *policy compliance*. Policies are specified only for data types and methods that deal with personal information. An access is valid only if the policy on the method is compliant with that of the data type of the accessed value, ensuring an authorized *access* and appropriate *use* of personal data. This approach of incorporating legal privacy requirements and employing static compliance checking is demonstrated on an example from the healthcare domain.

Furthermore, to create the sensitive types, i.e., specify policy on types for personal data, *sensitive functions* are introduced, which produce sensitive types when *i)* a data subject's information is combined with personal or non-personal information, or *ii)* personal information is used.

By making privacy an intrinsic component in design and development of systems, this approach provides a way to turn privacy needs into tangible controls and a way to verify if the design is privacy compliant.

4.3 Paper III : Static checking of GDPR-related privacy compliance for object-oriented distributed systems

Authors: Shukun Tokas, Olaf Owe and Toktam Ramezanifarkhani.
Publication: Submitted to the Journal of Logic and Algebraic Programming,

in first round revision [110]. This paper is an extended version of research work done in [109].
Summary: This paper extends [109], by including (i) an elaborate explanation on the proposed policy specification language, formalization of policy compliance, policy rules (including addition of more rules), (ii) *self* as an additional access right to cover a key concept in GDPR, i.e., *data subject access request*, and (iii) an operational semantics, and (iv) proof of soundness and progress.

Since it is the *data subject* that benefits from privacy, it is important that the notion of data subject is taken into account in designing privacy compliant systems. An additional access right, *self*, is introduced which allows us to specify and control access to self data for a data subject. This is useful in particular when considering subjects at compile time. For example, analysis can detect expressions involving the data subject and give the subject *read* access on its own data. The benefit of this approach is in maintaining *transparency* towards the data subject, as it gives $(Subject, all, self \sqcap read)$ as a default policy on personal information, meaning the data subject has read access to data about itself.

Sensitive functions are introduced, which are used in construction of *sensitive data types*, i.e., combining a type with either a sensitive type or a subject entity. Not all classes represent a principal and will therefore not be a natural part of policies on data type. This is addressed by introducing a notion of transfer of principal rights from caller to callee, i.e., allow the caller to act as a principal inside the method body of callee. In this paper, we continue with the case study from [109], and demonstrate how to use policies in combination with types, methods, and interfaces.

We have defined an operational semantics with policy tagging on data values, and prove *policy soundness*, i.e., any policy level obtained at runtime guarantees the one calculated by the static policy typing. Finally, we prove a progress property stating that the execution of each object in a program will continue unless the object is idle and there are no incoming messages reflecting method calls.

4.4 Paper IV : A formal framework for consent management

Authors: Shukun Tokas and Olaf Owe.
Publication: Proceedings of the 40th International Conference on Formal Techniques for Distributed Objects, Components, and Systems, FORTE 2020 [107].
Summary: This paper presents a formal framework for consent management that can be used to construct software systems that comply with several requirements in European Union's GDPR. In particular, we focus on the GDPR specific privacy requirements of *lawfulness and transparency*, *data protection by design* and *data subject access request*, that can be expressed at runtime and included in the overall system design. For this purpose we consider a high-level modeling language for distributed active object-systems.

The framework provides a general solution for data subjects, to view all

personal data about himself/herself in the system and change their privacy settings. The formalization is based on a policy and consent specification language, a notion of runtime policy compliance, a set of predefined interfaces and classes to deal with the consent, and an operational semantics which includes runtime checks for privacy compliance.

Privacy-enabling interfaces and classes are introduced that allow a data subject to inspect its own personal information and update previously given consent decisions. In order to check policy compliant access, personal data is tagged with sets of $(subject, purpose)$ pairs and a method accessing personal data is tagged with a *purpose*. The runtime system combines this information with consented policies for a given subject to derive an *effective* policy, and checks that every access to personal data complies with the consented policies. In addition, the approach includes a built-in generation of runtime privacy tags when new personal information is created. This approach to specification of privacy notions at runtime, dynamic compliance checking, and data subject's interaction with the system is demonstrated by a case study from the healthcare domain.

4.5 Additional papers

This section lists papers to which the author of this dissertation has contributed during her Ph.D. research, and which are not directly included as a part of this dissertation. The first two extended abstracts and research report corresponds to preliminary versions of papers included in the thesis. The last is a survey of code diversification techniques to improve security in internet of things.

1. **Title:** Language-Based Support for GDPR-Related Privacy Requirements
   Authors: Shukun Tokas and Toktam Ramezanifarkhani.
   Publication: Proceedings of the 30th Nordic Workshop on Programming Theory (NWPT), October 2018 [111].

2. **Title:** A Formal Framework for Consent Management
   Authors: Shukun Tokas and Olaf Owe.
   Publication: Proceedings of the 31st Nordic Workshop on Programming Theory (NWPT), November 2019 [106].

3. **Title:** Code Diversification Mechanisms for Internet of Things
   Authors: Shukun Tokas, Olaf Owe and Christian Johansen.
   Publication: A long version has been published as a UiO Research Report 473, January 2020 [108].

Chapter 5

Discussion

In this chapter, we return to the research questions formulated in Section 1.2 and summarize the contributions of the dissertation towards the overall research goal, and discuss the individual research goals in connection with the contributions, in Section 5.1. In addition, we discuss in Section 5.2 the limitations of the dissertation and potential future work.

5.1 Summary of the contributions

The overall goal of the dissertation is to investigate how to formalize non-functional requirements or properties pertaining to privacy and security, and leverage program analysis to enforce these requirements in the settings of object-oriented distributed systems. To tackle the complexity of distributed service-oriented systems, we have chosen a language setting that allows a simple and compositional semantics, which is beneficial for class-wise analysis. As motivated in *Chapter* 3, it is valuable to apply checks thoroughly and consistently in order to detect problems in the code at compile time, without actually executing the code. It makes the system more stable, since the number of runtime errors is reduced. Therefore, this thesis has focused on static checking when possible and considered runtime (compliance) checking of statically checked programs. The computational aspects of our language setting such as: method oriented communication, black box view of objects, and interface abstraction, makes our results relevant for a large part of object-oriented distributed systems. In particular the active object paradigm is suitable for modelling of service-oriented systems.

The information within a system is accessed and transformed in multiple ways by the declared program entities, and it is necessary to protect information which is of personal or confidential or sensitive nature. Furthermore, protection of the information involves two aspects: information security and information privacy, and we have developed different static analysis methods for these notions. With respect to information security measures: access control and information flow control are used to present authorized access to confidential information. With respect to information privacy measures: role, purpose, consent and access rights are used to ensure an authorized use of information.

Systems that process personal information needs to ensure that information access is in accordance with application privacy expectations that usually comes from regulations such as the GDPR, HIPPAA etc. However, much of the compliance efforts rely mostly on manual auditing and review. For large systems the challenges of checking and demonstrating that it is compliant with privacy requirements, remains unresolved. Incorporating the privacy specification in the

system design, allows the analysis to identify potential erroneous accesses that might be lurking in it. In addition, it provides an evidence, i.e., the algorithmic check of compliance for *accountability* and *transperancy* requirements.

We will now briefly discuss the contributions of this dissertation with respect to the individual research goals (cf. also Chapter 4):

1. *RQ1:* **How to formalize and enforce *confidentiality* properties and policies?**

   The methodology proposed in this thesis offers an approach to express *confidentiality* and *integrity* as the security requirements in system design as follows: The syntax and semantics of the core language [61] is extended with awareness of secrecy levels (*high* and *low*) and secrecy type information, as explained in *Chapter* 6. *Chapter* 6 defines notion of *interaction non-interference* tailored to the non-deterministic nature of objects the language setting. This property is enforced by a combination of two kinds of static analysis: a secrecy type system and the trace analysis system, in inter-object and network level communication, respectively. The approach is illustrated with several versions of an example from a news subscription service, in *Chapter* 6.

2. *RQ2:* **How to formalize privacy requirements specifications?** The methodology proposed in this thesis offers the possibility to express regulatory privacy requirements into the system design specifications as follows: *Chapter* 7 presents a formal policy specification language for specifying purpose, access rights and policy, and explains how these privacy specifications are integrated into the syntax and semantics of the language [61]. (Here, the specification deals with static notions.) *Chapter* 7 demonstrates how to specify these privacy requirements on personal data and program constructs. Furthermore, it explains how to operationalize the privacy by design and privacy by default principles. *Chapter* 8 introduces an additional access right *self*, which allows one to statically specify an access right on information about self, i.e., the data subject. *Chapter* 8, also introduces an additional interface *Subject*, in order to handle static awareness of data subject, as subject entities are not known at compile time. In *Chapter* 7 and *Chapter* 8, data subject's consent is expressed by *presence of policies*. In order to deal with dynamic changes in consent by the data subject, *Chapter* 9 introduces constructs to specify a positive consent (to reflect allowed access), a negative consent (to reflect forbidden access), and a *consent* list (i.e., a list of positive and negative consent(s) for a given data subject). In addition, *Chapter* 9 introduces a set of interfaces that forms the basis for interaction with external users, a set of classes that is used in interaction with the runtime system, and presents an approach to demonstrate how to employ them to ensure that the personal information is accessed in accordance with the current privacy settings.

3. *RQ3:* **How to define notions of *static* and *run-time* privacy compliance?**

   The methodology proposed in this thesis formalizes two notions compliance: static privacy compliance and dynamic privacy compliance. For static privacy compliance, the definitions needed to express compliance such as policy compliance, compliance of policy sets, join and meet over policy sets and implication on policy set, are given in *Chapter* 7 and *Chapter* 8. Formalization of static compliance that discuss application of the compliance definitions on program entities, can be found in *Chapter* 8. In addition, *Chapter* 8 presents the runtime formalization of compliance that is based on policy tags (on data values). This runtime formalization in *Chapter* 8 is adapted for checking runtime compliance on consented policies in *Chapter* 9. Note that the definition of *policy compliance* is different for static checking and dynamic checking. *Chapter* 9 presents a runtime formalization of compliance where compliance is defined with respect to a list of consented policies (i.e., list of positive and negative consent), for a given data subject.

4. *RQ4:* **How to enforce and check enforcement of static privacy compliance?**

   The methodology proposed in this thesis allows enforcement and checking of static compliance as follows: The syntax and semantics of the considered core language is extended with privacy specifications and policy compliance, static privacy compliance is established by static analysis, as explained in *Chapter* 7 and *Chapter* 8. *Chapter* 7 describes annotating policies on *data types* and *methods* of interfaces and classes, and explains how these policies puts restrictions on how they (types) are used and on actions they (methods) perform. In particular, it briefly describes how to create sensitive types and methods. In addition, *Chapter* 7 presents static policy checking which is defined by a set of syntax-directed rules, i.e., the type and effect system that checks that the policies are respected when the sensitive information is accessed. This integration of privacy specification and policy compliance in the language, and application of the static analysis is demonstrated on a small case study. A more in-depth description on the formalization presented in *Chapter* 7 can be found in *Chapter* 8. Furthermore, the policy rules from *Chapter* 8 are modified and extended in an improved compliance check, with awareness of *self access*. With the proposed type and effect system, it is not possible to check *self access*. To tackle this situation, *Chapter* 8 proposes an approach which allows the policy on a method to comply with the policy on the sensitive type, if the method uses only *self* access. *Chapter* 8 demonstrates the static analysis on a larger case study, which builds upon the one used in *Chapter* 7.

5. *RQ5:* **How to enforce and check a policy compliant access and support dynamic consent changes at runtime?**

The methodology proposed in this thesis allows enforcement and checking of runtime compliance, supporting dynamic consent changes, as follows: The solution, proposed in *Chapter* 9, consists of a privacy and *consent specification language*, a set of interfaces that forms the basis for interaction with external users together with class implementations and a set of classes that is used in interaction with the runtime system, and runtime checks for all access to personal data to check compliance with the current privacy settings. *Chapter* 9 presents this approach of dynamic compliance checking, in which the personal data is tagged with (subject, purpose) pairs, and methods accessing personal data are annotated with the purpose of processing. The tags are generated automatically be the run-time system. In addition, separate runtime checks are formulated for accessing personal data at runtime, which uses information from the tags on data and the calling context, and compare the information with consented policies of the subject to determine if the access is compliant with respect to the current consent specifications.

5.2 Limitations and future work

In this section we reflect on the limitations of our work and identify some potential extensions and future work.

In order to keep the formalizations and analysis techniques simple and presentable, we have focused on some key privacy concepts of the GDPR. Completeness of GDPR requirements has not been a goal. We have tried to identify those requirements that are susceptible to formalization, in particular, by static (compile-time) analysis using a type and effect system, and we have briefly looked at runtime analysis. The interplay of these two kinds of analysis is clearly useful. Mainly we focus on the intersection of four Articles of the GDPR: 5, 6, 15 and 25. Since we base our work only on a small subset of privacy policy requirements, the proposed policy specification language is limited in terms of what can be expressed and checked.

The specification language can be extended to include privacy notions, such as: *data controller* and *data processor* to identify data controller and data processor in various stages of processing in distributed projects; *temporal validity* to express data retention requirements and address storage limitation requirement; *exceptions* to model restrictions within a given policy; *distributed enforcement* to express multiple applicable regulatory requirements. Since this work deals with tags on data values, it is more practical to include information about applicable regulations or sectoral laws in tags in order to check compliance. For now, this seems complicated because they are extensive and may have conflicting expectations. Furthermore, *creator* and *data owner* for identification of creator and owner of information, can easily be added to the tags. Perhaps the personal data could have both a data subject and a data owner (like national tax office, national healthcare services), which will allow to model conditions such as the data subject may not remove the data alone since the data is owned by other

entities as well (for legitimate purposes, such as archiving, national interest, etc.).

The work presented in thesis can be extended to allow specification for more than one policy set (one for each sector/jurisdiction) and also adapt the policy compliance to include checks for additional information in tags. For static compliance checking, we are limited to one policy on a method. This can be generalized, but we believe that the idea of one purpose for each method (working on personal information) is clarifying and prohibits inflation of privacy-correct programs. An extension to multiple purposes for each method should discuss these aspects.

We have used the concept of *cointerface* in the language that allow type-correct callbacks to external caller objects. This concept gives the possibility of stating minimal security and privacy requirements to callers of methods of an interface. This potential can be explored further, for example to check more fine-grained attributes on callers such as owner or creator.

The work presented in the thesis can be extended to accommodate other legal bases (including the overriding ones, i.e., exceptions such as public interest, vital interest, emergencies etc.) by having separate policy lists for each legal basis, and a disjunction to chose from these bases depending on the context. Although we have the implicit policy that allows the data subject to have read access to self data, there is a downside to it: We have not taken the following cases into account (in *Chapter* 9). The initial policy is $(S, all, rincr)$ to allow self-access for a subject S. Should this be added explicitly in the consented list or would it be better to be baked in as an invariant? Should a user be allowed to remove self-access? These are interesting questions to look into. Since we consider only the *consented* policy list (as we focus on processing based on consent in *Chapter* 9), then baking it in is the best choice. However, when other legal bases are also included at runtime then keeping it as a policy is probably a better design choice. There are some exceptional situations where a user may be declined self-access, for instance in criminal proceedings or situations where other legal basis takes over and inhibits subject‘s *right to access*. Removing self-access in such cases would require means to disable self access. In our approach, this would require removing the $(S, all, rincr)$ policy from the *consented* list. However, in our current approach since $(S, all, rincr)$ is implicitly added to the consented list of S, its removal would require adding a $neg(S, all, rincr)$ to the consented by some other means (this not discussed in the paper).

It is necessary that the policy terminology used towards the data subject is simple but with a formal connection to the underlying programming elements. This is hinted briefly in the papers, more research is needed to identify and develop frameworks and techniques.

Often artificial intelligence technologies are developed using huge amounts of personal data, which is then used for analysis and predictions about data subjects. It would be very nteresting to investigate if development of AI systems is in accordance with PbD principle and gaining insight into accountability and compliance in such systems. It would be interesting to explore if the approach presented in this paper is applicable or can be adapted for AI technologies.

Papers

Chapter 6

A secrecy-preserving language for distributed and object-oriented systems

Toktam Ramezanifarkhani, Olaf Owe, Shukun Tokas

Published in *The Journal of Logic and Algebraic Programming*, October 2018, volume 99, pp. 1–25. DOI: 10.1016/j.jlamp.2018.04.001.

Abstract

In modern systems it is often necessary to distinguish between confidential (low-level) and non-confidential (high-level) information. Confidential information should be protected and not communicated or shared with low-level users. The *non-interference* policy is an information flow policy stipulating that low-level viewers should not be able to observe a difference between any two executions with the same low-level inputs. Only high-level viewers may observe confidential output. This is a non-trivial challenge when considering modern distributed systems involving concurrency and communication.

The present paper addresses this challenge, by choosing language mechanisms that are both useful for programming of distributed systems and allow modular system analysis. We consider a general concurrency model for distributed systems, based on concurrent objects communicating by asynchronous methods. This model is suitable for modeling of modern service-oriented systems, and gives rise to efficient interaction avoiding active waiting and low-level synchronization primitives such as explicit signaling and lock operations. This concurrency model has a simple semantics and allows us to focus on information flow at a high level of abstraction, and allows realistic analysis by avoiding unnecessary restrictions on information flow between confidential and non-confidential data.

Due to the non-deterministic nature of concurrent and distributed systems, we define a notion of *interaction non-interference* policy tailored to this setting. We provide two kinds of static analysis: a secrecy-type system and a trace analysis system, to capture inter-object and network level communication, respectively. We prove that interaction non-interference is satisfied by the combination of these analysis techniques.

The authors were partially supported by IoT-Sec (NRC) (https://its-wiki.no/wiki/IoTSec:Home) and by the project SCOTT (www.scott-project.eu).

Thus any deviation from the policy caused by implicit information leakage visible through observation of network communication patterns, can be detected. The contribution of the paper lies in the definition of the notion of *interaction non-interference*, and in the formalization of a secrecy type system and a static trace analysis that together ensure interaction non-interference. We also provide several versions of a main example (a news subscription service) to demonstrate network leakage.

6.1 Introduction

Programming languages can provide fine-grained control for security issues because they allow accurate and flexible security information analysis of program components [56]. In particular, to specify and enforce information-flow policies, the effectiveness of language-based techniques has been established. Information-flow policies are essentially specified based on a mapping from the set of logical information holders to a lattice of security classes representing levels of information sensitivity. Moreover, these policies usually dictate that no execution of the program should lead to an information-flow from more sensitive to less sensitive information holders [55], otherwise the information-flow is called "illegal".

Since information-flow policies are hyper-properties [25], i.e., are characterized as sets of trace properties, their specification, enforcement, and reasoning are difficult, especially in complicated systems. Therefore, giving a precise definition of the policy regarding legal and illegal information flows is challenging and highly dependent on the model of the system, the attackers, and their capabilities. Secure information flows are often expressed by semantic models of program execution in the form of a *non-interference* policy. Non-Interference stipulates that manipulation and modification of confidential data should be allowed in programs, as long as their visible outputs do not improperly reveal information about the confidential data. In addition, attackers are typically assumed to be able to view "low" information. The usual method for showing that non-interference holds is to demonstrate that the attacker cannot observe any difference between two executions that differ only in their confidential input [47]. In other words, if two possible input states of a program share the same low values, then the observable behaviors of the program execution on these states should be indistinguishable by the attacker [56]. Although the observable behavior is defined by the program output, there is no limitation in specifying program behaviors, and there is no fixed limitation on what is observable by the attackers. For example, when programs have runtime interactions with the environment, attackers may also see intermediate outputs [7]. In addition, the attacker may observe the progress of the program, e.g., absence or presence of the next observable value, which leads to the concept of progress-sensitive non-interference [7].

In the setting of distributed concurrent objects communicating by asynchronous methods calls, variables are encapsulated by objects and are not directly observable when forbidding remote variable access. Thus illegal ex-

plicit flows in the sense of assignment of confidential (or high) variables to non-confidential (or low) variables inside objects are not critical. In this setting, method calls and replies are represented by messages sent over a network, and thus network traffic could be observable to attackers. Therefore, patterns of network messages reflecting calls and replies can be informative to attackers and may reveal high-level information. For example, consider the following code in which for a specific user role the program's privileges are temporary raised to allow the creation of a new user folder[1]:

```
try:
        if (URole):
                o.raisePrivileges()
                os.mkdir('/home/' + username)
                o.lowerPrivileges()
except OSError:
        print('Unable to create new user directory')
        return False
return True
```

where the syntax $o.m(e)$ denotes a remote method call to o. An attacker may deduce confidential information about the user role based on the observation of method calls because in the then-branch there is a call and in the else-branch, which is empty in this case, there is no method call. This is a case of implicit information flow, which may appear when the observable program behavior includes observation of method calls. In addition, such information leakage can result in other successful critical attacks such as arbitrary code execution in injection attacks, which have been among the Top Ten critical attacks for years [87]. Here the essential information that makes the attack successful, is related to when and how to attack the program. For instance in the above example, the attacker knows that in some executions the program only raises the privilege level for a short while before lowering it again. After observation of the call in an execution, he can find a useful step toward a successful attack, for instance by throwing an exception, which leads away from the call to *lowerPrivileges*(). As a result, the program is indefinitely operating in a raised privilege state, possibly allowing further exploitation to occur by the attacker such as calling privileged functions [23], executing the attacker's own code and launching an injection attack.

To prevent information leakages in distributed concurrent object systems, we enrich the notation of non-interference by considering observability of interactions among objects by attackers. So, we introduce a notion of *Interaction Non-Interference*, which stipulates program executions to be equivalent in the view of attackers observing method call events. In addition, we are considering prevention of *inter-object leakages* when an object improperly sends secret information to another object, that might be non-observable from the network view. We also

[1] This vulnerability has been exploited in SplitVT, which is a program for splitting terminals into two shells, and allowing arbitrary code execution by attackers CVE-2008-0162.

consider some special cases of network leakage based on sophisticated mechanisms including suspension and non-blocking calls, which increase difficulties in specification and enforcement of non-interference. For example, attackers might be able to distinguish between executions with the same sequence of method calls that only differ in blocking or suspension behavior. However, we do not impose unnecessary restrictions on information flows from more sensitive to less sensitive variables inside objects, which make our approach more realistic than other pessimistic approaches based on static analysis, and thus significantly reduces the rate of false positives.

Our setting To formalize our approach we consider a high-level core language based on the chosen concurrency model, namely the paradigm of so-called *active objects* using asynchronous method calls as the only interaction mechanism, thereby combining the Actor model and object-orientation. This language is derived from *Creol* [60, 63]. Shared variables as well as thread-based notification are avoided. Synchronization control is achieved by cooperative scheduling: A local suspension mechanism allows an object to perform other tasks while waiting for a condition to become true or for a method result to appear. In Creol, an object can be seen as a black box in the sense that its content such as its fields are not observable from outside the object, and the main observation of objects is through their interaction by means of method calls. In a network this is observed through messages corresponding to invocations and completions of remote method calls, and dynamic object creation. Underlying network protocols may ensure that an attacker may observe but not alter the content of a message [55], and message content may be considered non-observable due to encryption techniques [123]. We consider the case that the destination and source of a message is considered observable, and in addition, we assume that an observer may be able to deduce the method name and whether it reflects a method invocation or a method completion. In addition, confidential message content communicated through the network to untrusted objects is also a source of secrecy leakage. Our approach covers these kinds of information leakages, and is relevant in Actor-based systems since these are based on message interaction. The notion of observable events may be further refined, for instance by considering certain parts of the network secure, say locally created objects.

We present an extension of Creol called *SeCreol*, in which Creol is extended with awareness of secrecy levels as well as secrecy type information. We show that programs respecting certain static restrictions, including secrecy typing, satisfy interaction non-interference. Moreover, to ensure that a system preserves secrecy of information, we use a combination of access control and information flow control of communicated information capturing direct access, and tracking communication patterns to capture indirect accesses.

Contribution The following are the main contributions of this paper: (i) Introducing interaction non-interference for non-deterministic distributed systems communicating by message passing. (ii) Extending the core Creol language by providing a security-type system and developing a static analysis approach

for detection of network leakage. (iii) Provably enforce the interaction non-interference property in programs of the SeCreol language by static analysis.

There is evidence, for example in [23], showing that the interaction between components and objects, e.g., the sequence of system and method calls, are valuable for attackers and can cause information leakage. Therefore, interaction non-interference is a critical property in a variety of secure systems, and thus its satisfaction and application are not limited to object-oriented systems.

Paper outline Section 6.2 formalizes the observable behavior of object-oriented distributed systems by explaining their execution model, and discusses security and attack models, as well as system assumptions, leading to the notion of interaction non-interference in Section 6.3. Section 6.4 introduces the SeCreol core language and a subscription example, to demonstrate our approach. A type system for secrecy levels is given in Section 6.5, while network leakage is considered in Section 6.6. Section 6.7 shows soundness of the network analysis, using the operational semantics of SeCreol given in Section 6.8. Section 6.9 discusses related work, and Section 9.6 concludes the paper points, and suggests possible future work.

6.2 Behavior of object-oriented distributed systems

We consider concurrent, distributed objects where each object has its own execution thread. An object does not have access to the internal state variables of other objects. Object communication is only by method calls, allowing asynchronous communication, implemented by means of asynchronous message

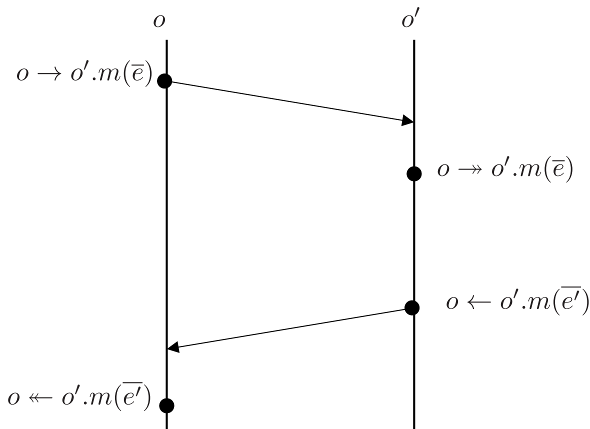

Figure 6.1: Illustration of method interaction: Object o calls a method m on object o' with arguments $\overline{e}$. The long arrows indicate message passing, and the bullets indicate generation of events. The corresponding event is written next to each bullet.

passing. In order to avoid undesirable waiting in the distributed setting, we allow mechanisms for non-blocking method calls. By means of a suspension mechanism, unfinished method invocations in an object may be placed on the object's process queue, for instance while waiting for a response from another object. The process will be enabled when the object receives the response. This allows flexible interleaving of incoming calls and (enabled) suspended processes. Internally in an object, there is at most one process executing at any time. Objects reflect concurrent system components, while data structure inside an object is defined by data types using functional programming.

The execution of a distributed system can be represented by the sequence of communication events that has appeared between the system components. This sequence is called the *communication history* (or *trace*) [19, 29, 58], which in our case consists of invocation and completion events of the called methods. At any point in time, the communication history abstractly captures the system state [28]. And we represent the set of executions of a distributed system by its possible communication histories, letting infinite histories represent non-terminating executions. The formalization of interaction non-interference, which we define later, is given as a property over the communication history.

Definition 1. ***(Communication events)*** *We consider the following events* Ev *of a system, where* o, o' *are objects,* m *is a method, and* $\overline{e}$ *is a list of expressions:*

- *the set of invocation events* $o \rightarrow o'.m(\overline{e})$*,*
- *the set of invocation reaction events* $o \twoheadrightarrow o'.m(\overline{e})$*,*
- *the set of completion events* $o \leftarrow o'.m(\overline{e})$*,*
- *the set of completion reaction events* $o \twoheadleftarrow o'.m(\overline{e})$*,*
- *the set of object creation events* $o \leftrightarrow o'.\textbf{new}\, C(\overline{e})$

The arrows reflect the direction of the message sending, and a two-way arrow indicates a synchronization event.

In our model, a method call to m is reflected by the four communication events

$$o \rightarrow o'.m(\overline{e}); o \twoheadrightarrow o'.m(\overline{e}); o \leftarrow o'.m(\overline{e}'); o \twoheadleftarrow o'.m(\overline{e}')$$

as graphically illustrated in Fig. 6.1. The figure shows the time-line of the caller object o and the callee object o'. An invocation message is sent from o to o' when a method m is called, which is reflected by the invocation event $o \rightarrow o'.m(\overline{e})$ where $\overline{e}$ is the list of actual parameters. The invocation reaction event $o \twoheadrightarrow o'.m(\overline{e})$ reflects that o' starts execution of the method, and the completion event $o \leftarrow o'.m(\overline{e}')$ reflects method termination, where $\overline{e}'$ is the list of returned values. Reading the reply in object o is reflected by the completion reaction event $o \twoheadleftarrow o'.m(\overline{e}')$. Other events may be interleaved with these four events, and in particular there might be an arbitrary delay between message receiving and reaction (due to message queuing).

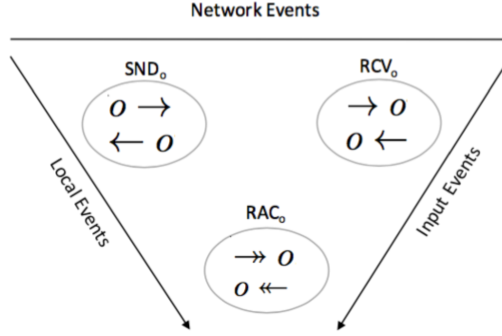

Figure 6.2: Illustration of the categories of method-related events for an object o. Observable network events for o consist of events from o (SND$_o$) and events to o (RCV$_o$), while non-observable o events are the internal reaction events of o (RAC$_o$).

Object creation is similar to method interaction. The event $o \leftrightarrow o'.\mathbf{new}\,C(\overline{e})$ can be understood as the sequence $o \rightarrow o'.C(\overline{e}); o \twoheadrightarrow o'.C(\overline{e}); o \leftarrow o'.C(void); o \twoheadleftarrow o'.C(void)$ where C represents the class constructor.

Fig. 6.2 categorizes communication events between objects. Messages sent from an object o are denoted by SND_o and messages received by an object o are denoted by RCV_o, while RAC_o denotes reactions events of the object o, which are internal o events. For a given object o, these three event sets are disjoint.

$$\begin{array}{lll} SND_o & \equiv \{o \rightarrow, \leftarrow o\} & \text{send events of o} \\ RCV_o & \equiv \{\rightarrow o, o \leftarrow\} & \text{receive events of o} \\ RAC_o & \equiv \{\twoheadrightarrow o, o \twoheadleftarrow\} & \text{reaction events of o} \end{array}$$

where $o \rightarrow$ denotes the set of invocation events *from* o and $\leftarrow o$ the set of completion events *from* o, while $\rightarrow o$ denotes the set of invocation events *to* o and $o \leftarrow$ the set of completions events *to* o, and so on. The set SND_o represents output from o, while RCV_o and RAC_o represent (external and internal) input to o. The union of SND_o and RCV_o represents events visible over the network, while RAC_o represents internal events not visible over the network, as illustrated in Fig. 6.2.

Next, we define *communication histories* as a sequence of events.

Definition 2. ***(Communication histories)*** *The communication history for a system at a given point in an execution is a finite sequence of* Ev *events.*

In our static analysis, we will consider finite traces (histories), representing executions up to a given program point, or segments of such executions. The empty sequence is denoted *empty* and we let semicolon denote sequence append (adding an event to the end of a sequence). Thus any sequence is either empty or can be seen as an appended sequence. We let t/S denote the

projection of trace t by a set of events S defined by $empty/S = empty$ and $(t;x)/S = \textbf{if } x \in S \textbf{ then } (t/S);x \textbf{ else } t/S$, and we overload the notation by letting t/o denote the projection of a trace t by the set of events that has the object o as sender or receiver, i.e., the subsequence of events that involve o.

6.2.1 Attack model

We consider two levels of attacks: i) *Inter-object leakage*, where attackers appear as objects and improperly obtain secret information from other objects. ii) *Network leakage*, where attackers derive secret information by observing the network traffic.

We consider the case that network attackers may know the whole system including program code and the distribution, but can only observe observable runtime events at the network level. Based on the explained behavioral model of object-oriented distributed systems, these events are passed as messages between objects. Therefore, even when assuming encryption of message content, the source and the destination object of messages are (implicitly) visible for attackers. Knowledge of the program code may allow an attacker to sometimes deduce the methods name in an event and whether it is an invocation or a completion message. By overestimating the attackers' capabilities, we assume that the method name might be known to an attacker, and therefore we assume that all aspects of a message except the parameters are observable. However, reaction events are not observable by attackers since they are internal to an object.

Hence, possible leakage of information includes *network leakages* and *inter-object leakages*. Leakage of network traffic is caused by observing the patterns of network traffic, while leakage to other objects occurs when an object improperly sends secret information to the other object, something that might not be observable from the network view, but from the other object.

Self-calls will not be observable over the network and do not cause network leakage. Similarly, communication to and from internal objects (for instance object generated by this at the same location) could also be considered non-observable at the network level (as well as local communication over internal sub-nets, assumed to be safe). However, for simplicity, we do not here include location awareness and treat all objects as if they were in different locations. (Location awareness could easily be added.)

In a distributed object system, the relative execution speed of the objects is not known. This is reflected in our model by letting the queue of incoming messages to an object be unspecified. Two messages sent from one object to another may be handled in the reverse order. In general, the ordering of RAC_o events in an execution may differ from the ordering of RCV_o events in the same execution. This gives a certain degree of non-determinism at the object level. At the network level one might consider message overtaking, loss, and duplication. However this gives an even higher degree of non-determinism, and therefore a weaker notion of leakage. We therefore ignore message overtaking, message loss, and message resubmission at the network level, since sequence information represents the upper limit of what is observable at the network level, assuming

reasonable reliable and efficient networks. The order of messages in a sequence is then observable and can be informative for attackers, and may cause network leakage.

This notion of network observability can be formalized as an equivalence relation over histories, called observable network equivalence ($\approx_{net}$), and will be used to compare the behaviors of two different execution histories (σ and σ') in order to detect leakage.

Definition 3. ***(Observable network equivalence)***

$$\sigma \approx_{net} \sigma' \equiv (obs(\sigma) = obs(\sigma'))$$

where obs expresses observable trace information*, defined inductively over finite histories σ by:*

$$\begin{array}{lcll} obs(empty) & = & empty & \\ obs(\sigma; (o \to o'.m(\overline{e}))) & = & obs(\sigma); (o \to o'.m) & \textbf{if } o \neq o' \\ obs(\sigma; (o' \leftarrow o.m(\overline{e}))) & = & obs(\sigma); (o' \leftarrow o.m) & \textbf{if } o \neq o' \\ obs(\sigma; (o \leftrightarrow o'.\textbf{new}\, C(\overline{e}))) & = & obs(\sigma); (o \to o'.C)(o \leftarrow o'.C) & \\ obs(\sigma; x) & = & obs(\sigma) & \textbf{otherwise} \end{array}$$

The *otherwise* case is taken when no other equation applies (in this case when x is a reaction event). Similarly, observable network equivalence *relative to a particular object o* is defined by

$$obs(\sigma/o) = obs(\sigma'/o)$$

The latter will be used when we do class-wise analysis, focusing on the object represented by this.

6.3 Interaction non-interference

As mentioned, non-interference ensures that an attacker should not be able to obtain confidential information by observing the low input and output of an executions of a system. We therefore need to capture the possible observations at a given point in an execution, represented by the communication history at that point, and define a notion of low equivalence between histories. Intuitively, two histories are low equivalent if they have the same low information, i.e., when ignoring non-observable events and arguments that are not low.

Definition 4. ***(Low equality)*** *$\sigma =_L \sigma'$ is defined by $low(\sigma) = low(\sigma')$, defining the low projection over histories and expressions as follows*

$$\begin{array}{lcll} low(empty) & = & empty & \\ low(\sigma; o \to o'.m(\overline{e})) & = & low(\sigma); o \to o'.m(low_m(\overline{e})) & \\ low(\sigma; o \leftarrow o'.m(\overline{e})) & = & low(\sigma); o \leftarrow o'.m(low_m(\overline{e})) & \\ low(\sigma; o \leftrightarrow o'.\textbf{new}\, C(\overline{e})) & = & low(\sigma); o \to o'.C(low_C(\overline{e})); o \leftarrow o'.C(void) & \\ low(\sigma; x) & = & low(\sigma) & \textbf{otherwise} \end{array}$$

where $low_m(\overline{e})$ is defined by the sublist of those parameters e_i for which the ith parameter is declared as Low *according to the method declaration. Similarly,*

*$\textsf{low}_C(\overline{e})$ is the sublist of actual class parameters e_i for which the ith class parameter is declared as **Low**.*

Observation It follows directly from the definitions above that low equality is a stronger relation than observable equality, i.e. $\sigma =_L \sigma'$ implies $\sigma \approx_{net} \sigma'$.

Let Σ be the set of (finite or infinite) traces of events for all possible completed executions of a system, letting finite traces represent terminating executions. Below σ and σ' will range over Σ. Non-interference of an object o expresses that if two executions involving o are low equal up to a certain time i, then also the low output will be the same, including the next step (after the given time), if it is an output (i.e., a SND_o event). If the object is deterministic with respect to its input, this could be expressed by

$$\forall \sigma, \sigma', i \,.\ (\sigma/o)|i =_L (\sigma'/o)|i \wedge (\sigma/o)[i{+}1] \in SND_o \Rightarrow\ (\sigma/o)|\,i{+}1 \approx_{net} (\sigma'/o)|\,i{+}1$$

where i represents the time relative to o (the number of o steps), $\sigma[i]$ denotes the ith element of σ ($i \in Nat$), and $\sigma|i$ is the sequence prefix $\sigma[1..i]$ (or σ if the length of σ is less than i).

Since our objects are non-deterministic, due to non-deterministic queues of incoming messages and of internal process queues, which in turn reflect non-deterministic network speed and object processing speeds, this definition cannot be used. It would be too strong, since it essentially expresses that the next low output (if any) from a given object o at a given time is deterministic. For instance if one possible execution at a given time starts with a low output event x from o and another with an observably different low output event x' from o, interference would not be satisfied, since the execution that starts with x has next a low output that is not observably the same as in all other executions.

In order to deal with non-determinism, we need to consider the set of possible executions, such that observable network equivalence holds for the set of possible next steps, i.e., considering all possible continuations of σ after i compared to the *set of* all possible continuations of σ' after i. In general, equivalence of two sets can be expressed by using existential quantification. We therefore express our notion of non-interference using an existential quantifier, as follows.

Definition 5. ***(Interaction non-interference)*** *We define* interaction non-interference *(INI_o) for an object (or a group of objects) o:*

$$\begin{aligned}&\forall \sigma, \sigma', i \,.\, (\sigma/o)|i =_L (\sigma'/o)|i \wedge (\sigma/o)[i+1] \in SND_o\\ &\Rightarrow\ \exists \sigma'' \,.\, (\sigma'/o)|i \leq (\sigma''/o) \wedge (\sigma/o)|i+1 \approx_{net} (\sigma''/o)|i+1\end{aligned}$$

where $\leq$ expresses the sequence prefix relation.

Here $\sigma, \sigma', \sigma''$ range over sequences of events reflecting possible completed executions (Σ). Thus the definition implies liveness (progress sensitivity). Class-wise analysis implies that we are interested in a given object o. The existential quantifier reflects possible non-determinism, and σ'' allows to choose the non-deterministic extension of σ' that follows σ for the given object o. The definition says that if an execution ($sigma$) has an output from o at time $i + 1$ then

Pr	$::=$	$In^*\ Cl^*$	program
$\mathcal{L}$	$::=$	Low | High | ...	secrecy levels
T	$::=$	I | Int | Any | Bool | String | Void | List[T] | ...	types
U	$::=$	$T \mid T : \mathcal{L}$	type/secrecy level
In	$::=$	**interface** I [**extends** I^+] [**with** I]$\{D^*\}$	interface declaration
Cl	$::=$	**class** C $([U\ cp]^*)$ [**implements** I^+]	
		$\{[U\ w\ [:= e]]^*\ [B]\ $ [[**with** I] $M]^*\}$	class definition
M	$::=$	$D\ B$	method definition
D	$::=$	$U\ m([U\ y]^*)$	method signature
B	$::=$	$\{[T\ x\ [:= e];]^*\ [s;]$ **return** $e\}$	method blocks
v	$::=$	$x \mid y \mid w$	variables (local/field)
e	$::=$	null | void | this | caller | $cp \mid v \mid f(\overline{e}) \mid e \sqsubseteq e$	pure expressions
rhs	$::=$	e | **new** $C(\overline{e})[: \mathcal{L}] \mid e.m(\overline{e})$	right-hand-sides
s	$::=$	skip | $s; s \mid v := rhs$	assignment
		$\mid e!m(\overline{e})$	asynchronous call
		await $v := e.m(\overline{e})$ | **await** e	suspension
		| **if** e **then** s [**else** s] **fi** | **while** e **do** s **od**	if and while

Figure 6.3: SeCreol BNF language syntax, with C denoting class name, I interface name, m method name, cp formal class parameter, w fields, y method parameter, x local variable. The brackets in $[T]$ and $[\overline{T}]$ are ground symbols. We let $[\ ]^*$, $[\ ]^+$ and $[\ ]$ denote repeated, repeated at least once, and optional parts, respectively.

any other execution with the same low o events up to time i has a possible extension (σ'') after time i with the observably same output event. Thus the set of observable output events for a given object at a given time must be deterministic relative to the low inputs before this time.

This definition of interaction non-interference is sufficient to avoid leakage by network attackers.

Our goal is to statically detect the INI_o property by means of two kinds of static analysis: i) a deductive system for secrecy typing ii) a deductive system for trace analysis of network events, such that both analyses are class-wise. In order to show this in some detail we will consider a high-level core language for the chosen concurrency model.

6.4 The SeCreol language

We define a minimal high-level language illustrating the concurrency model of concurrent objects communicating with asynchronous methods. The language, called SeCreol, builds on the concurrency model of Creol [60], extended with

secrecy constructs, including declaration of static secrecy levels for variables and parameters, and testing of runtime secrecy levels of objects.

The syntax of SeCreol is given in Fig. 6.3. A program consists of a number of interfaces and classes. We let the last class declared in the program be taken as the main class, which is instantiated automatically and its body will start to execute. An interface may have a number of super-interfaces and method declarations. A method of an interface or class may have a *cointerface*, which gives the (minimal) interface of the caller objects. (For simplicity an interface may only use one common cointerface for all its methods.) This allows type-correct call-backs [60]. A class C takes a list of class parameters cp, defines fields w, and has an optional method body for initialization B (also called the class constructor), followed by method definitions, with the corresponding cointerfaces as declared in the interfaces. Class parameters cp are like fields apart from being initialized through the **new** statement. Class parameters, the implicit class parameter this and the implicit method parameter caller are read-only. A class may implement a number of interfaces, and for each method of an interface of the class it is required that the class defines the method such that the cointerface and types of each method parameter and return value are respected. Additional methods may be defined in a class as well, but these may not be called from outside the class. For simplicity we omit class inheritance.

All variables and parameters are typed by data types or interfaces and for simplicity the syntax of the data type language is omitted here. Classes are not allowed as types, which means that an object can only be seen though an interface, and therefore, remote access to fields nor methods that are not exported through an interface is not allowed. This limits the possible interactions between the concurrent objects, regardless of where they are located, and in particular, shared variables concurrency is avoided. With respect to security analysis, it has the advantage that no field is observable from outside of an object. Thus observable behavior is limited to interaction by means of method-oriented communication.

Expressions e and functions f are side-effect free, and $\overline{e}$ is a (possibly empty) expression list, comma-separated. Statements include standard constructs for assignment, skip, if, while, object generation, and sequential composition. The *simple call* statement $e!m(\overline{e})$ is like message passing; a message is sent to the object expressed by e (the callee) indicating that it should execute method m (when the callee is free to do this) with a list of actual parameters $\overline{e}$. Thus the current object is not blocked, and will not receive the return value. If the return value is desired by the calling object, it may use the *blocking call* statement $v := e.m(\overline{e})$ or the *non-blocking call* statement **await** $v := e.m(\overline{e})$. The latter call statement forces the caller object to suspend the current process, allowing it to continue with any enabled suspended process in its process queue or perform an incoming call. Similarly, the conditional await statement **await** e suspends, placing the current process on the process queue. This process is enabled when the Boolean condition e is satisfied. The considered core language allows high-level and yet efficient method-based interaction between concurrent objects, supporting both passive and active waiting. The operational semantics

of the language is given in Section 6.8.

The language is strongly typed, and a typing system can be given in the style of [62]. We use a standard notion of subtyping (subsuming subinterfacing). If T' is a subtype/subinterface of T, we say that T' is better than T, and a method declaration D' is better than D if they have the same method name and number of parameters, the return type of D' is better than that of D, and each formal parameter of D is better than the corresponding one of D' (i.e., contravariance). The type system will ensure that a class properly defines all the methods of its declared interfaces (and superinterfaces), or better ones, and it ensures that each method call will be bound to a method declaration. The self-call this.$m(\overline{e})$ will be bound to the enclosing class (which must have a type-correct declaration of m). When e is of interface I, the method call $e.m(\overline{e})$ will be bound to I (or the closest superinterface of I with a (type-correct) declaration of m). For simplicity, we do not allow interfaces nor classes to declare several methods with the same name. Interface Any is the most general interface, supported by any object.

A variable is typed either by an interface or by a data type, called *object variable* or *data variable*, respectively. The runtime value of an object variable is an object identity (or null), and that of a data variable is a data value. Data variables are passed by value and object variables are passed by reference (i.e., the object identity is passed by value). Note that all object expressions are typed by an interface, except this, which is typed by the enclosing class. In a well-typed program, we may assume that each call is annotated by the interface/class of the callee, as in $o.m_I(\overline{e})$ where I will contain a declaration of m.

Secrecy Levels We enrich the typing system with *secrecy levels*. Secrecy levels range over $\mathcal{L}$ of basic secrecy descriptions with ordering $\sqsubseteq$, such that $(\mathcal{L}, \sqsubseteq)$ is a lattice, i.e., a partially ordered set with meets ($\sqcap$), joins ($\sqcup$), a top element $\top$ and bottom element $\bot$. Higher in the lattice means more secure; and thus the top element is the most secure. For example, a simple multi-level secrecy system might have secrecy descriptions *low*, *medium*, and *high*, with ordering $low \sqsubseteq medium \sqsubseteq high$, where $low = \bot$ and $high = \top$. A more expressive lattice could have several medium elements, indexed by object identities, or sets of object identities, for controlling access rights. This is essential at runtime for controlling secrecy with respect to objects; however, in our static analysis we will not use levels indexed by identities, since in general there is limited static knowledge about object identities.

In the syntax, all fields, formal parameters, and return values are given a secrecy level, with level *low* as default (if none is specified). Local variables do not have a declared secrecy level; their level starts as Low but may change after each statement. At runtime, objects are assigned a secrecy level that protects against unauthorized changes. Such a protected part is typical in policy enforcement research [38]. The statically assigned level of a formal data parameter represents the maximal level of any actual parameter. The declared secrecy level of an object variable expresses the secrecy of the object identity, which is typically *low*, reflecting that object *identities* (as such) are considered non-secret, whereas the

runtime secrecy level of an object gives more detailed information, for instance about the access rights of the object. The static analysis is class-based, and therefore the analysis is based on the (statically) declared levels, and not the runtime object levels. However, the language allows specification of restrictions on the secrecy level of a new object (as in x:=new C():Low) which determines the initial runtime secrecy level of the generated object. At runtime an object generated by the statement x:=new C():l will get the level $l \sqcap l_{\mathsf{this}}$ where l_{this} is the level of the parent object. Note that $l \sqcap l_{\mathsf{this}} \sqsubseteq l_{\mathsf{this}}$, which ensures that the secrecy level of the generated object will not exceed that of the parent object. As an object encapsulates local data and fields, these are not accessible from outside of the object, and we do not need static control of write access to fields of an object. At runtime the secrecy level of an object can be tested using the $\sqsubseteq$ operation in the program.

In the static analysis, we consider all statically assigned levels, and all possibilities for levels that can be assigned at runtime. This allows us to detect a maximal secrecy level for each program variables at any given point in a program (see Sec. 6.5). The next subsection describes an example of a network leakage, as well as a non-leaking version.

A subscription example

A simple subscription example illustrates the different language mechanisms, including simple, blocking and non-blocking method calls, and suspension mechanisms. Note the use of cointerfaces in Fig. 6.4, which implies that the caller of subscr is of type Client, which in turn allows the field *users* in class SUBSCLIENT to be typed as a list of Clients, thereby allowing the call users[i]!notify(n) in the class to be type correct since users[i] then is of interface type Client. Here List is a predefined generic data type with generators empty and insert, and with functions length and delete. All program variables in the example are declared as Low (by default) except the parameters to notify and publish, allowing high level News information to be passed to clients through these methods. The data type News may be defined as a String or a more complex data structure. To control and limit the notification of high level news, the test $n \sqsubseteq first(user)$ is made before notification. Thereby notification is restricted to client objects with a high enough runtime secrecy level.

We use the convention that class names are written in upper-case, interfaces and types are capitalized, while variable, method, and function names are in lower-case characters. We omit return void at the end of a class body. The non-blocking call await v := c.notify(n) makes a NEWSPROVIDER object send notifications at a speed adjusted to the Client c, without blocking itself from responding to other calls. In contrast, the *notify* method in class SUBSCLIENT uses a simple call, users[i]! notify(n), to notify each client, without suspending nor waiting for each one to receive the call. Also, the main program (i.e., the constructor of the main class) uses simple calls, in order to set up the initial system structure without waiting for replies. The suspending call in make_subscr allows the SUBSCLIENT object to continue with notifications

```
interface Client {
        Void notify(News:High n)
        Bool make_subscr(s:Subscriber) }
interface Subscriber with Client {
        Bool subscr()
        Bool unsubscr() }
interface SubsClient extends Subscriber, Client {}
interface Publisher {Void publish(News:High n)}
class SUBSCLIENT() implements SubsClient{
        List[Client] users := empty; // to store subscribers
        Void notify(News:High n) {Nat i:=1;
                while (i ≤ length(users)) do
                  if n ⊑ users[i] // checking runtime sec.levels
                  then users[i]!notify(n) fi; // simple call
                  i:=i+1 od;
                return void}
        Bool make_subscr(s:Subscriber){Bool ok;
                await ok:= s.subscr(); return ok}
     with Client
        Bool subscr(){
                if caller ≠ this
                then users := insert(users,caller) fi;
                return caller ≠ this }
        Bool unsubscr(){
                users := delete(users,caller);
                return true} }
class NEWSPROVIDER(Client c) implements Publisher {Void v;
            Void publish(News:High n){v := await c.notify(n); return v}}
class MAIN() { // main program
                SubsClient s1 := new SUBSCLIENT():High;
                SubsClient s2 := new SUBSCLIENT():High;
                Client a := new SUBSCLIENT();
                Client b := new SUBSCLIENT():High;
                Client c := new SUBSCLIENT();
                Publisher n1 := new NEWSPROVIDER(s1):High;
                Publisher n2 := new NEWSPROVIDER(s1);
                s2!make_subscr(s1);
                a!make_subscr(s2);
                b!make_subscr(s2);
                c!make_subscr(s1)}
```

Figure 6.4: A simple subscription example (with occurrences of *High* emphasized).

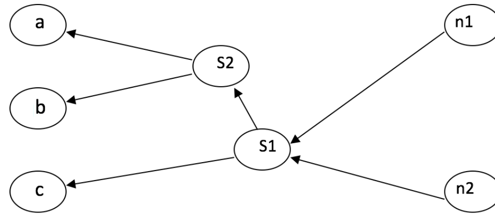

Figure 6.5: Illustration of the flow of news for the subscribers, clients, and news providers in the main program.

```
class SUBSCLIENT() implements SubsClient {
        ... // as before

        Void notify(News:High n) {Nat i:=1;
                while (i ≤ length(users)) do
                  if n ⊑ users[i] // checking runtime sec.levels
                  then users[i]!notify(n)
                  else users[i]!notify(empty) fi; // async. call
                  i:=i+1 od;
                return void }
}
```

Figure 6.6: Secure Class Server, second version – more secure, but generates dummy calls.

and other requests while waiting for Subscriber s to respond. The main program declares a subscriber $s1$ getting news from $n1$ and $n2$, and $s1$ notifies c and $s2$, while $s2$ further notifies a and b, as illustrated in Fig. 6.5. Note that $s1$ and $s2$ play a dual role, that of a client and that of a subscriber, and must therefore be of interface SubsClient. This makes the program well-typed.

The first version of the example poses a possible network leakage because a network observer may detect which subscribed client objects are High, by comparing the network traffic from a given subscriber object over time. The second version uses a dummy call in the else branch of notify to confuse a network observer (which cannot see the news content). The third version reduces the need for dummy calls due to non-deterministic background (self) activity that makes dummy calls when the object is not busy with other things. Here a somewhat different version of *notify* is given.

It is noticeable that in our approach we are considering the worst case scenario. For instance, in the subscription example, we consider that the set of subscribed and unsubscribed clients are known to the attacker (i.e., by tracking

```
class SUBSCLIENT() implements SubsClient { List[Client] users := empty;
       { this!mask() } //initialization, starting internal
background activity by a self-call
       ...
       Void notify(News:High n) { Nat i:=1;
           if improper(n) then skip
           else while (i ≤ length(users)) do
               users[i]!notify(n);
               i:=i+1 od fi;
               return void}
       // send to all or none

       Void mask(){Nat i:=1;
               while (i ≤ length(users)) do
                 await true;
                 users[i].notify(empty) ; // suspending call
                 i:=i+1 od;
               this!mask();
               return void}
}
```

Figure 6.7: Secure Class Server, third version

the execution communication), otherwise an observer may not be sure that the lack of notification to a client is due to unsubscription or the presence of high-level news.

6.5 Secrecy-type system

Prevention of information flow from one information holder to another one with a lower level have been considered in the literature. In our setting of active objects, characteristics such as information hiding and encapsulation imply that there is no external access to class fields [60]. Instead, we need to prevent information flow from one object to another, which we specify by means of static rules for acceptable information flow. We enforce this policy in our secrecy-type system for any well-typed program. Hence, our static analysis does not need to imply any restrictions inside a class such as limitations on information flows from high-level variables to low-level ones (e.g. $v_{High} := v_{Low}$), unless high information can be revealed in communication among objects or during suspension. Therefore, despite the fact that static analysis usually appears as a rather pessimistic and restrictive technique implying a high rate of false positives [66], we are able to be less restrictive. In order to make our static analysis as precise as possible, we allow the secrecy levels of program variables differ with the different program points. This makes our analysis flexible. However, we rely on level information

about fields formulated in a way similar to a class invariant, to be respected upon leaving a method invocation.

Our analysis is done class-wise. This is possible in our setting since remote access to fields is forbidden and since all object interaction is done by methods declared in an interface. The secrecy analysis of a class only depends on that class declaration, related interfaces, and the class parameter declaration of instantiated classes (through the **new** construct). We assume a well-typed program and assume each method call $e.m(\overline{e})$ is augmented by annotating the method name m by the interface of the callee e (as in $e.m_I(\overline{e})$), or the enclosing class when e is this. The secrecy-type system for classes and methods are shown in Fig. 6.8. The confidentiality of a class definition Cl is formalized by judgments of the form

$$\vdash \ Cl \ \ \textbf{ok}$$

expressing that the class definition obeys the confidentiality rules. The confidentiality of a method definition M is formalized by judgments of the form

$$C \vdash M \ \textbf{ok}$$

where C is the enclosing class. The confidentiality of a *statement* s is formulated by considering judgments of the form

$$C \vdash [\Gamma, pc] \ s \ [\Gamma', pc']$$

where C is the enclosing class, Γ is a mapping binding variable names to confidentiality levels for a given program point, and pc is the confidentiality level of the current program point. As Γ and pc depends on the program point, we let the "pre-binding" $[\Gamma, pc]$ denote the bindings in the pre-state of s and the "post-binding" $[\Gamma', pc']$ those in the post-state of s. Finally, the confidentiality of expressions and right-hand-sides rhs, given in Fig. 6.9, are formulated by judgments of the form

$$C \vdash [\Gamma, pc] \ rhs :: l$$

where l is the resulting confidentiality level of rhs. Post-bindings are not needed here as our expressions and right-hand-sides are side-effect free.

For simplicity, we let the mapping Γ_C (corresponding to table look-up) represent the *declared* secrecy levels of fields and class parameters for a class C as given in the class definition. If the secrecy level of a field w is declared as l, the binding $w \mapsto l$ is included in Γ_C. In contrast, Γ expresses confidentiality information depending on a particular program point. Since Γ-levels of class fields can increase and decrease, the type rules insist that at the end of each method (and at each ion point) their resulting levels should not exceed the declared secrecy level (or equal). This allows us to assume the declared level at each method start and after suspension. Furthermore, the notation $\Lambda[I, m, i]$ denotes the level of the ith parameter of the method as *declared* in interface I. And similarly for classes. For a class C, we let C also be the name of the class constructor (initialization code).

$$\frac{C \vdash\ M_i\ \mathtt{ok}, \quad \text{for each } M_i \in \overline{M}}{\vdash \mathtt{class}\ C(\overline{cp} : \overline{U})\{\overline{w} : \overline{U}';\ \overline{M}\}\ \mathtt{ok}} \text{(S-CLASS)}$$

$$\frac{\begin{array}{c} C \vdash [\Gamma_C[\overline{y} \mapsto \mathcal{L}(\overline{U}), \overline{x} \mapsto \overline{\mathsf{Low}}], \mathsf{Low}]\ s\ [\Gamma, pc] \\ C \vdash [\Gamma, pc]\ e :: l' \\ l' \sqsubseteq l \\ \Gamma[\overline{w}] \sqsubseteq \Gamma_C[\overline{w}] \end{array}}{C \vdash T : l\ m(\overline{y} : \overline{U})\{\mathtt{var}\ \overline{x} : \overline{T}; s; \mathtt{return}\ e\}\ \mathtt{ok}} \text{(S-METHOD)}$$

Figure 6.8: SeCreol confidentiality type system for classes and methods where Γ_C denotes the declared secrecy levels for class parameters and fields, in class C and Γ expresses confidentiality information depending on a particular program point.

Map notation A mapping M is given by a set of bindings $z_i \mapsto value_i$ for a finite set of disjoint identifiers z_i, the *domain*. The empty map is denoted $\emptyset$. Map look-up is written $M[z]$ where z is an identifier. A map update, written $M[z \mapsto d]$, is the map M updated by binding z to d, regardless of any previous bindings of z. Similarly $M[S]$ denotes M updated with a set S of (disjoint) bindings.

According to Rule S-CLASS in Fig. 6.8, confidentiality of each class is satisfied, or simply is *ok*, if the confidentiality of each method is satisfied. The confidentiality of a method (see Rule S-METHOD) is satisfied if its body satisfies confidentiality starting with the declared level bindings (for fields and class parameters, method parameters, and local variables) and with Low as the starting pc level, and resulting in some binding $[\Gamma, pc]$ such that Γ respects the declared field and class parameter bindings levels (i.e., $\Gamma[z] \sqsubseteq \Gamma_C[z]$ for each field/class-parameter z) and such that the returned value respects the declared output level of the method. As stated before, we check $\Gamma[z] \sqsubseteq \Gamma_C[z]$ because the secrecy level of program variables is allowed to be changed in different program points. This is not unlike previous approaches such as [123], except that we make no distinction between confidential and non-confidential variables as long as they do not affect the communication behavior. However, a complication for object oriented programs is that the order of method calls (and suspended processes) is not statically given, therefore the declared level of fields (and class parameters) must be respected at each method return and suspension point. This allows us to assume the declared level at method start and after suspension. This implies that the level of a field may temporarily be higher than the declared level. As we will explain with more details in secrecy-type system, it also implies that a simple fix-point calculation is required to be used when dealing with while-loops.

The SeCreol secrecy-type system for expressions and statements are shown in Fig. 6.9 and Fig. 6.10, respectively. These figures present a collection of

typing rules describing which secrecy type is assigned to each occurrence of an expression and program variable. In general, based on these rules, the level of an occurrence of an expression is determined using Γ and pc. The rules check that each occurrence of an actual parameter (or return value) respects the declared level of the corresponding formal parameter (or method return level), and that fields and class parameters respect the corresponding declared levels at suspension points and at method returns. In our formalization this is checked by premises in the rules; thus if these premises cannot be derived, the program will not be accepted as a program satisfying the secrecy rules. Note that each statement may adjust Γ, but only **if** and **while** statements may affect pc. Thus the level of variables and pc may differ at different program points, which for example means a call that is acceptable at one program point, might be unacceptable at another point. Furthermore, the rules ensure that parameters of calls made in a branch with a high condition will be high and may therefore not leak information.

Rule S-EXP states that the confidentiality of an expression e is achieved by $\Gamma[e] \sqcup pc$. We include pc since it represents the context level of the current program branch. Thus a low level expression occurring in a program branch with level pc, gets pc as its level, since it may reveal context information. We define $\Gamma[e]$ as follows: For a constant c (including null, this, *void*, and caller) $\Gamma[c]$ is Low (i.e., $\bot$), $\Gamma[e \sqsubseteq e']$ is High (i.e., $\top$), and for other kinds of expressions (including function applications) $\Gamma[e]$ is defined as $\sqcup_{v \in e} \Gamma[v]$, where v ranges over the variables textually occurring in e, and $\Gamma[v]$ is its level recorded in Γ. For simplicity, we here ignore so-called sanitizer functions, which are special functions resulting in a lower level than some of its inputs. Moreover, Object identities are not confidential, thus object variables are typically declared with a Low level. However, the level of such variables in Γ is affected by the branch level pc as other program variables. Thus the resulting level of object creation is pc as object identities as such are considered Low. For the right-hand-side of a call or new corresponding other rules in Fig. 6.9, each actual parameter is required to have a level not exceeding the declared level of the corresponding formal parameter. The resulting level of the call right-hand-side is the declared return level of the method, joined with the current context level pc. We observe that

$$C \vdash [\Gamma, pc]\, rhs :: l \;\Rightarrow\; pc \sqsubseteq l$$

which means the *rhs* level is always at least as high as pc. This fact can easily be proved by looking at each case of an expression or right-hand-side *rhs* according to the SeCreol syntax (Fig. 6.3).

According to the secrecy-type system for statements in Fig. 6.10, *skip* does not change anything. Similarly, a simple call does not change Γ nor pc, but the actual parameter levels must respect the declared levels of the corresponding formal parameters (as above). For an assignment, object creation statement, or call, $v := rhs$, with level l for rhs, the level of v in Γ is changed to l, which could imply a downgrade or an upgrade (or no change) of level. The pc is not modified since such a statement is considered efficiently terminating without any

$$\frac{}{C \vdash [\Gamma, pc]\; e :: \Gamma[e] \sqcup pc}\;\text{(S-Exp)}$$

$$\frac{C \vdash [\Gamma, pc]\; e_i :: l_i \quad l_i \sqsubseteq \Gamma_{C'}[cp_i]}{C \vdash [\Gamma, pc]\; (\textbf{new}\; C'(\overline{e}) : l) :: pc}\;\text{(S-New)}$$

$$\frac{C \vdash [\Gamma, pc]\; e_i :: l_i \quad l_i \sqsubseteq \Lambda[I, m, i]}{C \vdash [\Gamma, pc]\; e.m_I(\overline{e}) :: \Lambda[I, m] \sqcup pc}\;\text{(S-Call)}$$

$$\frac{C \vdash [\Gamma, pc]\; e_i :: l_i \quad l_i \sqsubseteq \Lambda[C, m, i]}{C \vdash [\Gamma, pc]\; \mathsf{this}.m(\overline{e}) :: \Lambda[C, m] \sqcup pc}\;\text{(S-SelfCall)}$$

Figure 6.9: SeCreol secrecy-type system for expressions and right-hand-sides.

branching.

For an **await** statement we must ensure that the declared levels of all fields and class parameters are respected, since the suspension may cause other processes to continue, for which we assume these declared levels. As mentioned, the declared level of fields must be respected at the end/suspension of each process. Levels of local variables will remain after an **await** statement since local variables are not affected by other processes. We therefore use map composition (+) in the post-state of an await to overwrite the levels of fields and class parameters by the declared levels (Γ_C). In the case of a call, the effect of the assignment part is added after the map composition since this assignment happens after suspension. A high await condition may cause implicit leakage, since the presence of high information may be leaked through a low output, for instance await <leaving house>; x.report(true) where report takes low input. Therefore the pc level resulting from an await is that of the await condition/right-hand-side (which is at least as high as the former pc level).

With respect to typing of security levels, a blocking call $v := e.m(\overline{e})$ can be seen as the sequence e!m($\overline{e}$); v:= <returned value>, and the non-blocking call **await** $v := e.m(\overline{e})$ can be seen as the sequence e!m($\overline{e}$); await <long enough>; v:= <returned value>. The rules for blocking and non-blocking calls can be derived from this understanding. Note that an await-statement may affect the pc. A high await condition may reveal secret information that may be leaked. An example could be await <leaving home>; athome:= false. where leaving one's home is considered secret. By raising the level after the high await condition, the athome variable becomes high, and leakage through data values is avoided. Other indirect leakage of an await statement is considered in the next section.

As mentioned an **if** statement may cause implicit leakage of high information, i.e., an **if** statement with a high test may reveal secret information through branches with different low level values communicated to other objects. To avoid this, Rule S-IF lifts the pc level of each branch by the level of the test. This will make all expressions occurring in both branches at least as high as the if-test. Thereby this kind of implicit leakage is avoided. (Note that l is at least as high as pc in the rule.) Since the static analysis does not know which branch is taken at runtime, the resulting value of Γ for each variable is calculated as the highest level of each branch. An **if** statement without an else-branch is like an **if** statement with skip in the else-branch.

$$\frac{\text{(S-SKIP)}}{C \vdash [\Gamma, pc]\ skip\ [\Gamma, pc]}$$

(S-COMPOSITION)

$$\frac{C \vdash [\Gamma, pc]\ s_1\ [\Gamma_1, pc_1] \quad C \vdash [\Gamma_1, pc_1]\ s_2\ [\Gamma_2, pc_2]}{C \vdash [\Gamma, pc]\ s_1; s_2\ [\Gamma_2, pc_2]}$$

(S-SIMPLE-CALL)

$$\frac{C \vdash [\Gamma, pc]\ e.m_I(\overline{e}) :: l}{C \vdash [\Gamma, pc]\ \ e!m_I(\overline{e})\ [\Gamma, pc]}$$

(S-RHS)

$$\frac{C \vdash [\Gamma, pc]\ rhs :: l}{C \vdash [\Gamma, pc]\ \ v := rhs\ [\Gamma[v \mapsto l], pc]}$$

(S-AWAIT)

$$\frac{C \vdash [\Gamma, pc]\ e :: l \qquad \Gamma[\overline{w}] \sqsubseteq \Gamma_C[\overline{w}]}{C \vdash [\Gamma, pc]\ \textbf{await}\ e\ [\Gamma + \Gamma_C, l]}$$

(S-AWAIT-CALL)

$$\frac{C \vdash [\Gamma, pc]\ rhs :: l \qquad \Gamma[\overline{w}] \sqsubseteq \Gamma_C[\overline{w}]}{C \vdash [\Gamma, pc]\ \textbf{await}\ v := rhs\ [(\Gamma + \Gamma_C)[v \mapsto l], l]}$$

(S-IF)

$$\frac{C \vdash [\Gamma, pc]\ e :: l \qquad C \vdash [\Gamma, l]\ s_1\ [\Gamma_1, pc_1] \qquad C \vdash [\Gamma, l]\ s_2\ [\Gamma_2, pc_2]}{C \vdash [\Gamma, pc]\ \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{fi}\ [\Gamma_1 \sqcup \Gamma_2, pc]}$$

(S-WHILE)

$$\frac{C \vdash [\Gamma_i, pc_i]\ e :: l_i \qquad C \vdash [\Gamma_i, l_i]\ s\ [\Gamma_i', pc_i'] \qquad \Gamma_{i+1} = \Gamma_i \sqcup \Gamma_i', \quad pc_{i+1} = pc_i \sqcup pc_i'}{C \vdash [\Gamma_1, pc_1]\ \textbf{while}\ e\ \textbf{do}\ s\ \textbf{od}\ [FIX_i(\Gamma_i), pc_1]} \qquad i = 1, 2, \ldots$$

Figure 6.10: SeCreol secrecy-type system for statements.

The treatment of **while** is similar to an **if** statement without an else-branch, except that the static analysis cannot predict how many times the branch is iterated. Each iteration may lift the levels in Γ or pc. However, a loop will have a finite number of program variables and since there is a finite number of levels, there is a minimal fixpoint reachable in a finite number of approximations (typically i equal to one or two). Rule S-WHILE reflects this fixpoint calculation.

The secrecy typing ensures that there is no flow from high values to low values, and that values evaluated in an if-branch with a high test are high (since they may depend on the test), and similarly for values evaluated after an await

with a high test or inside a while-loop with a high test. Thus the values of low variables in any program state do not depend on high inputs. Furthermore, this ensures that for any event generated by o, the values of parameters declared as low do not depend on high inputs. A proof of this based on a semantics that includes runtime secrecy levels, is given in [90], which proves the soundness of the secrecy rules presented here. [2]

The subscription example The subscription example in Fig. 6.4 has a straight forward secrecy typing. The secrecy analysis of the while loop needs no iteration to reach the fixpoint. The notify call in class newsprovider (as well as that in SUBSCLIENT) has a high actual parameter (n), which is acceptable since the notify method in Client is declared with a high (formal) parameter. However, the if-statement has a high test, and at the network level the pattern of notify calls could cause network leakage, which we consider in the next section. The High annotations on the objects created in the main class concern the runtime level of these objects, and not the variables declared in the main program. The second and third versions of the example have no additional secrecy challenges, the argument empty is low which is always acceptable.

Another example A (quasi) example is given in Fig. 6.11 to illustrate possible changes in the levels of fields (xh and xl) and local variables (x). Level changes are written to the right of each line, not repeating unchanged information. The program satisfies the rules for confidentiality, i.e., the program does not leak information in its explicit output and respects field levels at return/await statements. Note that the lowering of xh was needed to make the *check* call allowed, that the higher level of the x was maintained over the await (since x is local), that the higher level of x was acceptable in the *passw* call, and that the high level of the local variable x is allowed at the return point (after which x is de-allocated). If the await condition had been high, pc would be raised to high after the await, and the call to check would not be secrecy-type correct since xh would then be high.

6.6 Network level leakage

We here consider enforcement of network-level non-interference (INI_o) for SeCreol programs by means of *static trace analysis*. We assume a given program that is secrecy-type correct, i.e., has passed the secrecy-type analysis of Section 6.5.

[2] Alternatively, one could add a level to methods, letting this level be used as the starting pc level of the method body, and require that methods called with high pc must be high. In the subscription example, the local variable u would then be high, the notify call would be accepted, and the secrecy analysis of the while loop would need no iteration (as before). However, this would mean that a high method is not observable and therefore do not cause leakage. All methods called by a high method must also be high, which gives some limitations in what is allowed. If we believe that different notify patterns represent an observable leakage, we can consider the generated pattern and check for leakage. This approach is explained below in Sec. 6.6.

```
interface Passw{
 Int:Low passw(Int:High x)// store password, return a ref number
 Int:High check(Int:Low x)// check validity of password given ref
}

class TEST(Passw o){
 Int:High xh;
 Int:Low xl;

 Int:High test(Int x){  xh ↦ High . Note: all others are Low
      xl := x;          xl ↦ Low
      x := xh;          x ↦ High
      xh := xl;         xh ↦ Low . Note: suspension is ok even with x high
      await <low cond.>; xh ↦ High, x ↦ High . Note: all others are Low
      xh := o.passw(x); xh ↦ Low . Note: the call is ok with x high
      x := o.check(xh); x ↦ High . Note: the call is ok since xh now is Low
      return x      Note: return is ok with x High, since xh ⊑ High ∧ xl ⊑ Low
}}
```

Figure 6.11: An example showing level changes in fields and local variables (indicated to the right in each line).

Moreover, we assume that await-, if-, and while-tests are decorated with the levels resulting from the secrecy-type analysis, using the notation e_l, and we assume the interface of a callee is known from the type analysis.

The static analysis is class-wise and we check that a class is not leaking network information, according to INI_{this}. The analysis is based on *trace expressions*, Δ, detected by means of static analysis applied to *method bodies*. The trace expression of a method reflects the possible traces of an execution of that method including calls generated and consumed by the method. Each trace expression may contain "high" subtraces, caused by high if- and while-conditions. Therefore we check whether any high (sub)trace can be reduced to a low (sub)trace (given the context of the class), as formalized below by *INIcheck*. In the CLASS rule of Fig. 6.12, the premise $INIcheck(\Delta_{m_i})$ checks the INI property for the trace expression of each method m_i of the class. It must be checked that each trace expression reveals no high information, as detailed further below. We let **isOK** denote that a class declaration satisfies interaction non-interference, and we let M **reveals** Δ denote that a method declaration M reveals the trace set Δ. In the METHOD rule, $default_T$ denotes the default initial value for variables of type T. The substitution of default values for the local variables makes the initial values explicit. In this analysis, we ignore the initial reaction event since it is implicit for the given method.

For statements s we consider judgments of the form $\vdash [\Delta'] \; s \; [\Delta]$. Due to non-determinism caused by suspension and process queues, we cannot estimate

$$\frac{\begin{array}{c}(\text{CLASS})\\ \vdash m_i(y)\{\texttt{var}\ x;s;\texttt{return}\ e\}\ \texttt{reveals}\ \Delta_{m_i}\ ,\quad \text{for each } m_i \in \overline{M}\\ INIcheck(\Delta_{m_i})\ ,\quad \text{for each } i\end{array}}{\vdash \texttt{class}\ C(cp)\{w;\overline{M}\}\ \texttt{isOK}}$$

$$\frac{\begin{array}{c}(\text{METHOD})\\ \vdash [\Delta]\ s\ [\mathsf{caller} \leftarrow \mathsf{this}.m]\end{array}}{\vdash m(y)\{\texttt{var}\ T\ x;s;\texttt{return}\ e\}\ \texttt{reveals}\ \Delta[default_T/x]}$$

Figure 6.12: Network level rules for classes and methods.

the exact history of a method body as a single trace, but may estimate it as an (extended) regular expression, using $(\ldots|\ldots)$ for choice, semicolon for sequential composition, and superscript $*i$ for repetition, and in addition $\bullet$ for any (finite) sequence. The latter represents unknown activities of the object during suspension. Thus a trace expression defines a set of possible traces. Input events will not occur in the trace expressions, since they cannot be detected from the code. But we include reaction events of the form $\mathsf{this} \twoheadleftarrow o.m$ because they can be detected from the code and give implicit information about the corresponding input events ($\mathsf{this} \leftarrow o.m$). So even though reaction events are not directly observable in the class code; they implicitly restrict the time where the corresponding observable input event $\mathsf{this} \leftarrow o.m$ may occur. For instance, a blocking call to m on an external object o gives the trace $\mathsf{this} \rightarrow o.m; \mathsf{this} \twoheadleftarrow o.m$ while a simple call to m gives $\mathsf{this} \rightarrow o.m$. In the first case there will not be any output from the object between the observed events $\mathsf{this} \rightarrow o.m$ and $\mathsf{this} \leftarrow o.m$, as opposed to the second case which gives no restriction for how late the completion event may appear. So an observer may distinguish a difference. Without the reaction events in the traces, the two cases will have the same trace expressions and our system would be unsound since observable differences are not captured. Simple, blocking, and suspending calls are observably different and are represented differently in the traces.

Furthermore, self-calls pose some non-trivial challenges. For instance, consider the code

```
if e_High then v:=this.m1() else v:=this.m2() fi
```

If $m1$ makes the call o!n(true) and $m2$ makes the call o!n(false), where the parameter is Low, the outcome of the high if-test is leaked to object o. In order to handle such cases we include self-calls in the traces even though they are not observable. The example will then not pass the INIcheck test. And in the example if $\mathsf{e_{High}}$ then this.m(true) else this.m(false) fi, the outcome of the if-test may seem to be leaked if $m(x)$ makes the call $o.n(x)$ where o is an external object. However, in this example the argument to m must be high in order to pass the secrecy-typing requirements, which means that there is no leakage.

Another challenge related to self-calls is that a call $o.m(\overline{e})$ may be a self-call if o equals this, unless the interface of o is not supported by the enclosing class (for instance when m is not implemented in the class). This means that calls can be categorized as self-calls (calls to this), external calls (calls through interfaces not supported by the enclosing class), and calls for which we do not know at static time if they are self-calls or not. The analysis must deal with all these categories. The call events of external calls are visible to an observer, but not for self-calls. However, for a self-call the activity caused by called method might be important, but not for external calls since the output of such an invocation is not an output of this object. Thus the static treatment is non-trivial.

In the analysis of statements, we employ backward trace analysis for detection of generated observable events, by triples $[\Delta']\ s\ [\Delta]$ similar to Hoare triples, where Δ denotes a trace expression. Intuitively, it means that the statement s generates the trace Δ' (the *pre-trace*) when Δ is the trace generated after s has terminated (the *post-trace*). The notation $\Delta[e/x]$ denotes the trace set expression Δ with all occurrences of the variable x replaced by the expression e. The example if high then o:=o'; o!m(...) else o'!m(...) fi, motivates that the effects of assignments should be considered in the analysis (in order to detect non-leakage in this case). As we will explain later in this section, the above code satisfies the policy. Therefore, the reason for doing a backward analysis is that the generated trace expressions contain program variables, and therefore the effect of assignments must be considered. The handling of assignments can then be done by backward substitution as in Hoare logic. Thus an assignment $x := e$ causes the replacement of e for x in the pre-trace, letting $[e/x]$ denote the substitution. Similar substitutions are caused by object creation, blocking, and non-blocking calls with assignment part $x := rhs$, except that here the value assigned to x is not statically given, and is reflected by a fresh value.

In the axioms of Fig. 6.13 for basic statements, the pre-trace Δ' is expressed by means of the post-trace, given by a symbol Δ, consistent with left-constructive analysis. Based on these rules, *skip* does not have any effect on a trace while in case of an assignment, the pre-trace is determined by replacement of x with e in the post-trace. Moreover, in case of a simple call, a call event is added to the pre-trace (even if it is a self-call). In the rules for **await**, the symbol $\bullet$ represents arbitrary traces caused by suspension. Since a high test in a conditional await-statement may depend on high variables, its enabledness may reveal secret information. For instance, an await statement with a condition testing *raised privileges* gives a high (sub)trace. Therefore the $\bullet$ is considered high in this case, which affects the INIcheck. For instance, a program path going through a conditional await testing *raised privileges* gives a high trace expression, which cannot be used to match any low trace. The notation of $\Delta[\mathit{fresh}/x]$ in these rules denotes that all occurrences of the variable x is replaced by a fresh constant.

Rules for trace analysis are shown in Fig. 6.14. The rule for sequential composition resembles that of Hoare Logic. We may for instance derive $\vdash [\Delta']\ \mathit{skip}; s\ [\Delta]$ from $\vdash [\Delta']\ s\ [\Delta]$. In Rule IF-ELSE we encode the traces of the branches, Δ_1 and Δ_2, into a branching expression, $(\Delta_1 \mid \Delta_2)_l$ where l is the level of the if-expression (as obtained by the secrecy-typing). The rule for

$\vdash [\Delta]\ skip\ [\Delta]$

$\vdash [\Delta[e/x]]\ x := e\ [\Delta]$

$\vdash [(\mathsf{this} \to o.m); \Delta]\ o!m(\overline{e})\ [\Delta]$

$\vdash [(\mathsf{this} \to o.m); (\mathsf{this} \twoheadleftarrow o.m); \Delta[\mathit{fresh}/x]]\ x := o.m(\overline{e})\ [\Delta]$

$\vdash [((\mathsf{this} \to x.C); (\mathsf{this} \twoheadleftarrow x.C); \Delta)[\mathit{fresh}/x]]\ x := \mathbf{new}\, C(\overline{e})\ [\Delta]$

$\vdash [(\bullet)_l; \Delta]\ \mathbf{await}\ e_l\ [\Delta]$

$\vdash [(\mathsf{this} \to o.m); \bullet; (\mathsf{this} \twoheadleftarrow o.m); \Delta[\mathit{fresh}/x]]\ \mathbf{await}\, x := o.m(\overline{e})\ [\Delta]$

Figure 6.13: Trace axioms for basic statements. Here *fresh* denotes a fresh symbol.

$$\textsc{(seq-comp)}\quad \frac{\vdash [\Delta'']\ s\ [\Delta'] \qquad \vdash [\Delta']\ s'\ [\Delta]}{\vdash [\Delta'']\ s; s'\ [\Delta]}$$

$$\textsc{(if-else)}\quad \frac{\vdash [\Delta_1]\ s_1\ [\varepsilon] \qquad \vdash [\Delta_2]\ s_2\ [\varepsilon]}{\vdash [(\Delta_1|\Delta_2)_l; \Delta]\ \mathbf{if}\, e_l\, \mathbf{then}\, s_1\, \mathbf{else}\, s_2\ [\Delta]}$$

$$\textsc{(while)}\quad \frac{\vdash [\Delta]\ s\ [\varepsilon]}{\vdash [((\Delta[\overline{fresh/w}])_l)^{*i}; \Delta']\ e_l\, \mathbf{do}\, s\, \mathbf{od}\ [\Delta']}$$

Figure 6.14: Rules for trace analysis. In Δ^{*i} each fresh constant c is replaced by c^i, making freshness depend on the iteration, and $\overline{w}$ is the list of program variables used in a right hand side inside an iteration.

while is similar, using superscript $*i$ for repetition where the *iteration index* i allows us to refer to each iteration. In particular, this allows freshness to be dependent on an iteration i, as captured by Δ^{*i}. Remark that the nesting of if- and while-statements determine the inner secrecy labels in a regular expression. For a loop with a counter variable j starting on 1 and such that $j := j+1$ occurs in the loop body as the only update of j, we simply use j as the iteration index, ignoring the statement $j := j+1$ in the further analysis, and avoiding replacing j with a fresh constant, and thereby allowing more simplifications.

Subscription example Consider the original notify method defined in Fig. 6.4. We need to find Δ' such that $[\Delta']\ body\ [\mathsf{caller} \leftarrow \mathsf{this}.notify]$ for the *body*. Using the rules (in a left to right manner) we determine Δ' as

$$((\mathsf{this} \to users[i].notify \mid \varepsilon)_{\mathsf{High}})^{*i}; \mathsf{caller} \leftarrow \mathsf{this}.notify$$

$(\Delta\|\Delta)_l$	$\longrightarrow$	Δ	
$\bullet;\bullet$	$\longrightarrow$	$\bullet$	
$(\varepsilon)_l$	$\longrightarrow$	ε	
$\mathsf{this} \rightarrow \mathsf{this}.m;\Delta$	$\longrightarrow$	Δ	for a simple, recursive call to m outside a branch, if Δ does not start with $[\bullet;]\mathsf{this} \twoheadleftarrow \mathsf{this}.m$

Figure 6.15: Simplification rules for the trace set of a method m. A self-call $\mathsf{this} \rightarrow \mathsf{this}.m$ is detected as simple if it is not followed by $\mathsf{this} \twoheadleftarrow \mathsf{this}.m$ nor $\bullet;\mathsf{this} \twoheadleftarrow \mathsf{this}.m$. And a self-call is recursive if it is to the enclosing method m (as in method *mask*).

which contains high subtraces. As explained below, it will not satisfy the INIcheck.

For the second version of notify, we get the trace expression

$$((\mathsf{this} \rightarrow users[i].notify \mid \mathsf{this} \rightarrow users[i].notify)_{\mathsf{High}})^{*i};\mathsf{caller} \leftarrow \mathsf{this}.notify$$

As explained below it simplifies to $(\mathsf{this} \rightarrow users[i].notify)^{*i};\mathsf{caller} \leftarrow \mathsf{this}.notify$, which does not contain high subtraces, and satisfies the INI property.

For method make_subscr, we need to solve

$$[\Delta']\ body\ [\mathsf{caller} \leftarrow \mathsf{this}.make_subscr]$$

for the method *body*. We determine Δ' as

$$\mathsf{this} \rightarrow s.subscr;\bullet;\mathsf{this} \twoheadleftarrow s.subscr;\mathsf{caller} \leftarrow \mathsf{this}.make_subscr$$

which has no high subtrace, and is therefore not causing network leakage.

INIcheck The $INIcheck$ test of a class is done by checking $INIcheck(\Delta_m)$ for each method m of the class (including the constructor) where Δ_m is the trace expression generated for the body of m. The test is passed if Δ_m can be reduced to a trace expression without high subtraces, using the simplification rules defined in Fig. 6.15. If the simplified Δ_m has high subtraces, we flatten Δ_m to a set of trace expressions t_i, by flatting the branches, where each t_i is defined as high if it goes through a path of Δ_m with a high branch (or subtrace), and otherwise low. For example, a trace such as $\Delta_1;(\Delta|\Delta')_l;\Delta_2$ is flattened to $t_1 = \Delta_1;\Delta_l;\Delta_2$ and $t_2 = \Delta_1;\Delta'_l;\Delta_2$. For each flattened high trace t_i we must then check if it can be recreated from the set of flattened low traces of the same method, considering also possible other activities during suspension. This check is denoted

$$t_i \ \mathbf{matches}\ S$$

where S is the union of the set of *low traces of the same method* and the set of *low traces of any background self activity* without the final non-observable completion

events. The background self activity is captured by the set of flattened traces of recursive methods, with a simple or non-blocking recursive self-call, and where the method is called by a simple self-call from the class constructor (directly or indirectly). The rules for detecting recreation is defined in Fig. 6.16. Thus a high trace t_i passes the check if t_i **matches** S following from the rules, where as mentioned, S is the union of all low traces of the same method and low traces representing self behavior. The final non-observable completion event is omitted in a trace representing self behavior since this event is non-observable. Clearly, in order to match the final completion event of t_i one must involve a low trace of the same method, while self behavior may appear at suspension points.

The simplification rules in Fig. 6.15 are used to reduce an INIcheck. The rules are confluent and terminating. The first rule says that a branch expression with two identical branches can be simplified, removing the level of the branch expression. The second rule says that suspending twice in a row is equivalent to suspending once (since any number of enabled suspended processes may be taken during each suspension). The last rule says that a simple recursive call (after the last suspension point of a trace) can be ignored since the event is non-observable, the recursive invocation will be done later during suspension, and since an irreducible trace must be revealed in a single invocation. Thus a leakage in a recursive method can be found in the body without the recursive call. A call is detected as a self-call if it has the form $\mathsf{this} \rightarrow \mathsf{this}.m$ and as recursive if m is to the enclosing method. An event $\mathsf{this} \rightarrow \mathsf{this}.m$ in a flattened trace is detected as a simple self-call if it is not followed by $\mathsf{this} \twoheadleftarrow \mathsf{this}.m$ nor $\bullet; \mathsf{this} \twoheadleftarrow \mathsf{this}.m$. In the rules we let $sscalls(t)$ denote the low traces of the methods called by simple self-calls in trace t, omitting the final completion event ($\mathsf{this} \leftarrow \mathsf{this}$), which is non-observable. In order to limit the amount of false positives, one may add further rules, such as replacing $(\Delta|\Delta')_l$ by $(\Delta'|\Delta)_l$ according to some ordering over trace expressions. And one may add that an initial $\bullet$ in a branch can be removed, if the branch expression is preceded by a $\bullet$, and similarly for a final $\bullet$ in a branch. However, a more detailed study is beyond the scope of this paper.

The rules in Fig. 6.16 incorporate suspended behavior S of an object, by starting with a (low and flattened) trace t in S and either stopping at a suspension point ($\bullet$), adding the remaining part of t to S, and continuing with another trace in S. We may add a suspension point at the front or at the end of trace in S, since each such trace is starting from and ending in a suspension (explaining rule 5 and 6 of Fig. 6.16). And an iteration may be included or skipped, and a bullet in an iteration may be ignored. Thus S is in general infinite; however, in the context of checking whether a given flattened trace expression t_i is in S, we may expand S while matching t_i from left to right. This can be done in finite time letting each application of a rule match a lager part of t_i.

The formalization of matching depends on the *duration of network observations*. If we assume the observations are made over a short term, it is plausible that any method of an interface of the class has been called before the observations starts, and its low traces should therefore be included in S. If the observation is long term which we take as the default, this assumption is not appropriate, and the low traces of the active self behavior may safely be included

ε	**matches**	S		
t	**matches**	S	**if**	$t \in S$
$t; t'$	**matches**	S	**if**	t **matches** $S \wedge t'$ **matches** $S \cup sscalls(t)$
$t; t''$	**matches**	S	**if**	$t; \bullet; t'$ **matches** $S \wedge t''$ **matches** $S \cup \{t'\} \cup sscalls(t)$
$\bullet; t$	**matches**	S	**if**	t **matches** S
$t; \bullet$	**matches**	S	**if**	t **matches** S
$t; t''$	**matches**	S	**if**	$t; (t')^{*i}; t''$ **matches** S
$t; t'; t''$	**matches**	S	**if**	$t; (t')^{*i}; t''$ **matches** S
$(t; t')^{*i}$	**matches**	S	**if**	$(t; \bullet; t')^{*i}$ **matches** S

Figure 6.16: Rules to determine if a (high) trace matches a set of low traces, while adding any new suspended processes to S. t is a branch-free trace expression, and S a set of such expressions. Here $sscalls(t)$ denotes the set of low trace expressions, without the final non-observable completion event, for each method with a simple self-call in t. (Exemplified in Fig. 6.17 and Fig. 6.18).

in S. Thus we consider here the worst case in this respect, but the same approach can be used for the case of observations with short duration.

Examples Fig. 6.17 and Fig. 6.18 explains the application of rules in INIcheck with some synthetic examples to cover different possibilities. In the first example, the if statement has the trace $(this \rightarrow o'.m\ ; this \twoheadleftarrow o'.m \mid this \rightarrow o'.m; this \twoheadleftarrow o'.m)_{\mathsf{High}}$ simplified to $this \rightarrow o'.m; this \twoheadleftarrow o'.m$ using the simplification rules, which has no high subtrace, and therefore there is no network leakage. $Exp2$ will not pass the INIcheck because $m1$ is textually different from $m2$. Remark that if $o = \mathsf{this}$ and if $m2$ and $m1$ make the same calls, we have a false positive. In the third example, there is no high subtrace, and therefore no leakage. The examples $Exp4$ to $Exp7$ do not satisfy the INIcheck because they have a pair of high traces (and here no background self activity set S is given). In $Exp7$ the two self-calls may indirectly cause an observable difference, if m has observable output, since the call in the then-branch makes this output happen before the current method is finished (with a visible $\leftarrow \mathsf{this}$ event) while this need not to be the case for the call in the else-branch. $Exp8$ has a possibility of leakage, which is detected. This happens when more than one call happens at runtime, in which case the second call reveals information about the high test. Otherwise, an attacker cannot distinguish between the high or low if-tests. Moreover, as explained in the details in the figure, $Exp9$ satisfies the INIcheck because the high trace can be matched by low traces, and thus the execution traces are indistinguishable from the attacker's point of view. The last non-trivial example, $Exp10$, also satisfies the INIcheck. It is because the only high flattened subtrace in mtd_1, i.e., t_1 can be also recreated with the combination of the low trace of the same method, i.e., t_2 and the trace of simple self-call to method n in mtd_1, i.e., t_n. In other words, the observer cannot distinguish t_1 from an execution that includes t_2 such that t_n happens at the suspension point. This example shows

(Exp1)

$$\frac{\textbf{if } e \textbf{ then } o := o';\ v := o.m \textbf{ else } v := o'.m \textbf{ fi}}{\checkmark : \text{Trace: } (\mathit{this} \to o'.m\ ; \mathit{this} \twoheadleftarrow o'.m}$$

$$\mid \mathit{this} \to o'.m; \mathit{this} \twoheadleftarrow o'.m)_{\mathsf{High}}$$

Simplified to $\mathit{this} \to o'.m; \mathit{this} \twoheadleftarrow o'.m$

No high subtrace.

(Exp2)

$$\frac{\textbf{if } e \textbf{ then } v := o.m_1 \textbf{ else } v := o.m_2 \textbf{ fi}}{\times : \text{Trace: } (\overbrace{\mathit{this} \to o.m_1; \mathit{this} \twoheadleftarrow o.m_1}^{t_1}}$$

$$\mid \overbrace{\mathit{this} \to o.m_2; \mathit{this} \twoheadleftarrow o.m_2}^{t_2})_{\mathsf{High}}$$

Both t_1, t_2 are high.

(Exp3)

$$\frac{v := o.m_1;\ \textbf{await } e_{\mathsf{Low}}; v := o.m_2}{\checkmark : \mathit{Trace} : \mathit{this} \to o.m_1; \mathit{this} \twoheadleftarrow o.m_1;}$$

$$\bullet; \mathit{this} \to o.m_2; \mathit{this} \twoheadleftarrow o.m_2$$

No high subtrace.

(Exp4)

$$\frac{\textbf{if } e \textbf{ then } v := o.m \textbf{ else await } v := o.m \textbf{ fi}}{\times : \mathit{Trace} : (\overbrace{\mathit{this} \to o.m; \mathit{this} \twoheadleftarrow o.m}^{t_1}}$$

$$\mid \overbrace{\mathit{this} \to o.m; \bullet; \mathit{this} \twoheadleftarrow o.m}^{t_2})_{\mathsf{High}}$$

No simplification, and t_1, t_2 high.

(Exp5)

$$\frac{\textbf{if } e \textbf{ then } v := o.m \textbf{ else } o!m \textbf{ fi}}{\times : \mathit{Trace} : (\overbrace{\mathit{this} \to o.m; \mathit{this} \twoheadleftarrow o.m}^{t_1}}$$

$$\mid \overbrace{\mathit{this} \to o.m}^{t_2})_{\mathsf{High}},\ t_1, t_2 \text{ both high.}$$

(Exp6)

$$\frac{\textbf{if } e\ \textbf{ then await } v := o.m \textbf{ else } o!m \textbf{ fi}}{\times : \text{Trace: } (\mathit{this} \to o.m; \bullet; \mathit{this} \twoheadleftarrow o.m}$$

$$\mid \mathit{this} \to o.m)_{\mathsf{High}}$$

Figure 6.17: Examples of network level rule applications for INIcheck (Part I). Here e is high and ✓ indicates success and × failure.

the application of $sscalls(t)$ as the set of low trace expressions for a method with a simple call in the matching rules in Fig. 6.16.

(Exp7)

$$\frac{\begin{array}{c}\textbf{if } e \ \textbf{ then await } v := this.m \\ \textbf{else } this!m \textbf{ fi}\end{array}}{\begin{array}{c}\times : Trace : (this \rightarrow this.m; \bullet; this \twoheadleftarrow this.m \\ \mid this \rightarrow this.m)_{\mathsf{High}}\end{array}}$$

(Exp8)

$$\frac{\begin{array}{c} mtd_1()\{ \\ \textbf{if } (e)_{Low} \textbf{ then } v := o.m_1 \textbf{ fi}; \\ \textbf{if } (e')_{Low} \textbf{ then } v := o.m_2 \textbf{ fi}; \\ \textbf{if } (e)_{High} \textbf{ then } v := o.m_1 \\ \textbf{else } v := o.m_2 \textbf{ fi}\}\end{array}}{\times : \text{Traces similar to } EXP2.}$$

(Exp9)

$$\frac{\begin{array}{c} constructor()\{this!n()\} \\ n()\{v := o.m1; this!n\} \\ mtd_1()\{\textbf{if } e \textbf{ then } v := o.m_1 \textbf{ else } skip \textbf{ fi}\}\end{array}}{\begin{array}{c}\checkmark : \text{Trace of } mtd_1 : (this \rightarrow o.m_1; this \twoheadleftarrow o.m_1 \mid \varepsilon)_{\mathsf{High}}; caller \leftarrow this.mtd_1 \\ \text{Flattened} : t_1 = (this \rightarrow o.m_1; this \twoheadleftarrow o.m_1; caller \leftarrow this.mtd_1)_{\mathsf{High}} \text{ and } t_2 = caller \leftarrow this.mtd_1 \\ \text{Simplified trace of method } n : \ t_n = this \rightarrow o.m_1; this \twoheadleftarrow o.m_1 \\ \text{Here } t_1 = t_n; t_2 \text{ and } t_2 \text{ is low. So } t_1 \textbf{ matches } S \text{ since both } t_1, t_n \text{ are in } S.\end{array}}$$

(Exp10)

$$\frac{\begin{array}{c} n()\{v := o.m1\} \\ mtd_1()\{this!n(); \textbf{await } e_{\mathsf{Low}}; \textbf{if } e \textbf{ then } v := o.m_1 \textbf{ else } skip \textbf{ fi}\}\end{array}}{\begin{array}{c}\checkmark : \text{Trace of } mtd_1 : (this \rightarrow this.n; \bullet; (this \rightarrow o.m_1; this \twoheadleftarrow o.m_1 \mid \varepsilon)_{\mathsf{High}}; caller \leftarrow this.mtd_1) \\ \text{Flattened} : t_1 = this \rightarrow this.n; \bullet; (this \rightarrow o.m_1; this \twoheadleftarrow o.m_1)_{\mathsf{High}}; caller \leftarrow this.mtd_1 \\ \text{and } t_2 = this \rightarrow this.n; \bullet; caller \leftarrow this.mtd_1 \\ \text{Trace of } n : \ t_n = this \rightarrow o.m_1; this \twoheadleftarrow o.m_1; caller \leftarrow \mathsf{this}.n \\ \text{Thus } sscalls(this \rightarrow this.n) = t'_n = this \rightarrow o.m_1; this \twoheadleftarrow o.m_1 \\ \text{Then } this \rightarrow this.n; t'_n; caller \leftarrow this.mtd_1 \textbf{ matches } S \text{ since } t_2 \in S \text{ and } t'_n \text{ is in the extended } S.\end{array}}$$

Figure 6.18: Examples of network level rule applications for INIcheck (Part II). Here e is high and ✓ indicates success and × failure.

The subscription example revisited The original notify method reveals

$$(\mathsf{this} \rightarrow users[i].notify \mid \varepsilon)^{*i}_{\mathsf{High}}; caller \leftarrow \mathsf{this}.notify$$

where i is the iteration index. In order to remove the first high call we need to look at any other background activity in this object. In the first version we do not have any such activity, so based on our simplification rules, the call in the high branch cannot be removed, and not satisfaction on INIcheck is detected. However, for the second version, the redefined notify method reveals the following trace expression

$$(\mathsf{this} \rightarrow users[i].notify \mid \mathsf{this} \rightarrow users[i].notify)^{*i}_{\mathsf{High}}; caller \leftarrow \mathsf{this}.notify$$

which is simplified to

$$(\mathsf{this} \to \mathit{users}[i].\mathit{notify})^{*i}; \mathit{caller} \leftarrow \mathsf{this}.\mathit{notify}$$

which means there is no leaking because due to textually equivalence of the two branches the high subscript is removed. Therefore, this method passes the INIcheck test.

For the last version of the subscription example, the redefined version of *notify* reveals the trace expression

$$(\varepsilon \mid (\mathsf{this} \to \mathit{users}[i].\mathit{notify})^{*i})_{\mathsf{High}}; \mathit{caller} \leftarrow \mathsf{this}.\mathit{notify}$$

Flattening gives the low trace $\mathit{caller} \leftarrow \mathsf{this}.\mathit{notify}$, denoted t_1, and the high trace t_2 given by:

$$((\mathsf{this} \to \mathit{users}[i].\mathit{notify})^{*i}; \mathit{caller} \leftarrow \mathsf{this}.\mathit{notify})_{\mathsf{High}}$$

We then need to show that t_2 **matches** $\{t_1\} \cup S$ where S is the low traces of the background self activity. Since t_1 is without suspension, t_2 must end with t_1, and we need to show $(\mathsf{this} \to \mathit{users}[i].\mathit{notify})^{*i}$ **matches** S. The constructor of class SUBSCLIENT has a simple self-call to *mask* and *mask* has a simple recursive self-call. Thus S contains the trace expression of *mask* (ignoring the final non-observable completion event)

$$(\mathsf{this} \to \mathit{users}[i].\mathit{notify}; \bullet)^{*i}_{\mathsf{Low}}; \mathsf{this} \to \mathsf{this}.\mathit{mask}$$

where the simple recursive self-call can be removed based on the simplification rules. Thus we have

$$(\mathsf{this} \to \mathit{users}[i].\mathit{notify}; \bullet)^{*i} \ \mathbf{matches}\ S$$

Clearly, we also have $(\mathsf{this} \to \mathit{users}[i].\mathit{notify})^{*i}$ **matches** S by the last rule of Fig. 6.16. Therefore also this example passes the INIcheck test! In contrast, the original notify method would need a masking method that selects a subset of the users for notification for instance by using non-deterministic choice (if added to the language).

6.7 Theoretical results

In this section, we prove that by applying the proposed network trace analysis in Section 6.6, any possible deviation from the INI policy defined in Section 6.3 will be detected. The possible execution traces σ for our language are defined by the operational semantics in Section 6.8.

For an execution trace σ, the subtrace involving a given object o, denoted σ/o, consists of output events SND_o, input events RCV_o, and internal (input) events RAC_o. The rules for generating trace expressions Δ of an object o talk about the events generated by o, namely SND_o and RAC_o, as well as $\bullet$. Non-observable self calls are included in the traces σ and trace expressions Δ.

Lemma 1 (Traces correspond to the operational semantics). *Consider a trace expression generated by the trace analysis of a given method m. The trace expression will cover all traces possible by an execution of m according to the operational semantics (Section 6.8) in the sense that each execution of m gives a trace that is an instantiation of one of the flattened trace expressions, when restricted to events generated by the method execution.*

Proof outline Consider a given class C and method m with method body s. We show that the trace expression Δ generated for m, based on the trace axioms (TAx) of Fig. 6.13 and the trace analysis rules (TSt) of Fig. 6.14, is according to the operational semantics, in the sense that the events generated at runtime by any invocation of m is an instantiation of one of the flattened traces of Δ (instantiating the variables in Δ). To obtain the subtrace generated by an invocation of m we let the events generated by the rules of the operational semantics be tagged by the unique identity of the invocation, given by the value of $\delta[callId]$ where δ is the state of the object in the left-hand-side of the rule. In the trace generated by an execution we may then extract the subtrace with a given *callId* tag, called an invocation trace, which will consist of output and reaction events generated by the object (i.e., SND_o and RAC_o events) with the completion event of the method as the last event.

Consider an arbitrary invocation trace of an arbitrary execution of the given method m. We may then prove that the invocation trace is equal to an instantiation of a flattened trace expression. Each triple $[\Delta]\, s\, [\Delta']$ in the trace analysis can be understood as the Hoare triple $[h \in \{\Delta\}]\, s\, [h \in \{\Delta'\}]$ where h is the local communication history (trace), using for instance a reasoning system similar to [35] (without futures). The soundness of the axioms and rules of Figs. 6.13 and 6.14 follows from the soundness of the reasoning system for histories in [35], using this translation to Hoare logic. In particular, the events generated in the pre-traces in the axioms of Fig. 6.13 correspond exactly to the events generated in the operational semantics, and the substitutions in the assignment-like statements in Fig. 6.13 correspond to those of Hoare logic.□

Theorem 1 (INI Deviation Detection). *The interaction non-interference policy is satisfied for a set of objects if the corresponding INIcheck is satisfied by the corresponding classes of those objects. For each object o of a class C we have*

$$INIcheck_C \Rightarrow INI_o$$

Proof outline Let us prove $INIcheck_C \Rightarrow INI_o$ by contradiction, i.e., $INIcheck_C \wedge \neg INI_o$. We assume that there is a class C that based on the rules in Fig. 6.12 satisfies **isOK** and that there is an object o of that class that does not satisfy INI, i.e., $\neg INI_o$. This means that for each method m in that class with s as the method body, Δ is calculated in the form of $[\Delta] s [caller \leftarrow \mathsf{this}.m]$ based on Fig. 6.13 and 6.14. According to Lemma 1, Δ reflects the communication traces obtained from the operational semantics. Due to the satisfaction of INIcheck,

we know that each high trace t obtained after simplification and flattening of Δ (using the simplification rules in Fig. 6.15) can be recreated by the set S of low traces of m and background self activity (i.e., t **matches** S), using the rules in Fig. 6.16.

And, based on $\neg INI$, there are sequences of events σ and σ', and some i, such that:

$$(\sigma/o)|i =_L (\sigma'/o)|i \wedge (\sigma/o)[i+1] \in SND_o \wedge \\ \nexists \sigma'' \,.\, (\sigma'/o)|i \leq \sigma'' \wedge (\sigma/o)|i+1 \approx_{net} (\sigma''/o)|i+1$$

where σ, σ' and σ'' range over possible execution traces. Therefore, based on $\neg INI$ there is not any execution trace σ'' which contradicts that the flattened t is matched by the set S of suspension behaviors. The set of trace expressions representing background self activity provides an underestimation of the process queue of o when an invocation of m is made. Thus there is an execution that continues with an instance of t after $(\sigma'/o)|i$.

It suffices to consider σ and σ' such that the inputs to o from other objects come as late as possible, i.e., a call $o' \rightarrow o$ comes just before $o' \twoheadrightarrow o$ and such that a completion $o \leftarrow o'$ comes just before $o \twoheadleftarrow o'$ (for o' different from o). This can be made possible by considering executions where the objects generating calls or completions to o are slowed down. This has no effect on the behavior of o, and it allows us to derive the missing RCV_o events from a flattened trace expression.

It is also clear that if for each object o_i of class C_i the trace analysis of C_i is OK, and thus INI_{o_i} is satisfied, then for a set of such objects O, INI_O is satisfied as well. Since any receive event for an object must happen after the corresponding send event, we consider the subset of executions where the inner send events match the corresponding receive events. We may assume σ is in this set. Formally, consider any σ, σ' (in this set), and i. We may assume the left side of the implication, i.e. $(\sigma/O)|i =_L (\sigma'/O)|i \wedge (\sigma/O)[i+1] \in SND_O$ and need to prove $\exists \sigma'' \,.\, (\sigma'/O)|i \leq (\sigma''/O) \wedge (\sigma/O)|i+1 \approx_{net} (\sigma''/O)|i+1$. We have that $(\sigma/O)[i+1] \in SND_O$ and thus there must be an object o in O such that $(\sigma/o)[i+1] \in SND_o$. This means that we can apply INI_o since σ/o and σ'/o are determined by σ/O and σ'/O and we may choose σ'' such that o is scheduled in the next step (as in σ). $\square$

6.8 Operational semantics

We here present the operational semantics of the core language. The main purpose of this semantics is to understand the communication traces appearing at runtime. We therefore omit the complication of assigning runtime secrecy levels to objects and values of program variables. An operational semantics with secrecy levels is presented in [90]. The semantics formalizes the notion of process queue (PQ), idleness, and generation of events (as labels on the transition relation). Thus a sequence of execution steps gives rise to a sequence of events, capturing

the history. Generation of identities for objects and method calls is handled by underlying semantics functions and implicit attributes. The operational semantics uses an additional construct **get** to deal with (the completion of) call statements, letting **get** u appear as a right-hand-side, where u denotes a method call identity. The *query* v:=get u will block while waiting for completion of u and v:=await get u will suspend. We use the notation explained above for mappings, and a denotes an object expression, b denotes a Boolean expression, o denotes an object identity, u denotes a method identity, and d denotes a value (a data value or an object identity).

For simplicity we omit rules for while. While can be handled by expanding a while-statement to be executed to an if-statement with an inner while upon execution of the the while-statement. The semantics of the while-statement while b do s od is equivalent to that of if b then s; while b do s od fi. The semantics of an if-statement without else-part, if b then s; while b do s od fi, is equivalent to if b then s else skip fi.

The operational semantics of the core language is given in Fig. 6.19 and Fig. 6.20. A runtime configuration of a system is seen as a multiset of objects and messages (using blank-space as a binary multiset constructor). Each rule in the operational semantics deals with only one object o, and possibly messages, reflecting that we deal with concurrent distributed systems communicating asynchronously. When a subconfiguration c can be rewritten to a c', this means that the whole configuration $\ldots c \ldots$ can be rewritten to $\ldots c' \ldots$, reflecting interleaving semantics. Each object o is responsible for executing all method calls to o as well as self-calls. An object has at most one active process, reflecting a method execution, and a sequence of suspended processes organized in a process queue PQ. Remote calls and replies are handled by messages. Objects have the form

$$o : \mathbf{ob}(\delta, \overline{s})$$

where o is the object identity, δ is the current object state, and $\overline{s}$ is a sequence of statements ending with a **return**, representing the remaining part of the active process, or **idle** when no active process. A message have the form

$$\texttt{msg}\ o \rightarrow o'.m(\overline{e})$$

representing a call event, where o is caller, o' callee, m the method and $\overline{e}$ the actual parameter values, or

$$\texttt{msg}\ o \leftarrow o'.(u, d)$$

representing a completion event where d is the returned value and u the identity of the call.

In the operational semantics rules, pc is the confidentiality level of the object that is going to execute an instruction at the current program point. Moreover, the operational semantics uses some additional variables, like PQ for holding the process queue and nextId and nextOb for generating unique identities for calls and objects. These appear as fields in the operational semantics. Furthermore, this is handled as an implicit class parameter, while callId and caller appear

SKIP: $o : \mathbf{ob}(\delta, \texttt{skip}; \overline{s}) \xrightarrow{empty} o : \mathbf{ob}(\delta, \overline{s})$

ASSIGN : $o : \mathbf{ob}(\delta, v := e; \overline{s}) \xrightarrow{empty} o : \mathbf{ob}(\delta[v := e], \overline{s})$

IF-TRUE : $o : \mathbf{ob}(\delta, \underline{\texttt{if}}\ b\ \texttt{then}\ \overline{s1}\ \texttt{else}\ \overline{s2}\ \texttt{fi}; \overline{s}) \xrightarrow{empty} o : \mathbf{ob}(\delta, \overline{s1}; \overline{s})$
$\texttt{if}\ \delta[b] = true$

IF-FALSE : $o : \mathbf{ob}(\delta, \underline{\texttt{if}}\ b\ \texttt{then}\ \overline{s1}\ \texttt{else}\ \overline{s2}\ \texttt{fi}; \overline{s}) \xrightarrow{empty} o : \mathbf{ob}(\delta, \overline{s2}; \overline{s})$
$\texttt{if}\ \delta[b] = false$

NEW : $o : \mathbf{ob}(\delta, v := \texttt{new}\ C(\overline{e}); \overline{s}) \xrightarrow{o\ \leftrightarrow\ \delta[\mathsf{nextOb}].C(\delta[\overline{e}])} o : \mathbf{ob}(\delta[v := \mathsf{nextOb}, \mathsf{nextOb} := next(\mathsf{nextOb})], \overline{s})$
$\delta[\mathsf{nextOb}] : \mathbf{ob}(\delta_C[\mathsf{this} \mapsto \delta[\mathsf{nextOb}], \overline{cp} \mapsto \delta[\overline{e}]], init_C)$

SIMPLE CALL : $o : \mathbf{ob}(\delta, a!m(\overline{e}); \overline{s}) \xrightarrow{o \to \delta[a].m(\delta[\mathsf{nextId}, \overline{e}])} o : \mathbf{ob}(\delta[\mathsf{nextId} := next(\mathsf{nextId})], \overline{s})$
$\texttt{msg}\ o \to \delta[a].m(\delta[\mathsf{nextId}, \overline{e}])$

CALL : $o : \mathbf{ob}(\delta, [\texttt{await}]\ v := a.m(\overline{e}); \overline{s}) \xrightarrow{o \to \delta[a].m(\delta[\mathsf{nextId}, \overline{e}])} o : \mathbf{ob}(\delta, [\texttt{await}]\ v := \texttt{get}\ \delta[\mathsf{nextId}]; \overline{s})$
$\texttt{msg}\ o \to \delta[a].m(\delta[\mathsf{nextId}, \overline{e}])$

START : $\texttt{msg}\ o' \to o.m(u, \overline{d})$
$o : \mathbf{ob}((\alpha|\beta'), \mathbf{idle}) \xrightarrow{o' \twoheadrightarrow o.m(u, \overline{d})} o : \mathbf{ob}((\alpha|(\beta[\mathsf{caller} \mapsto o', \mathsf{callId} \mapsto u, \overline{y} \mapsto \overline{d}])), \overline{s})$
$\texttt{where}$ m is statically bound to $(m, \overline{y}, \beta, \overline{s})$ in the class of this

Figure 6.19: Operational rules (Part I) for small-step semantics (without secrecy levels).

as implicit method parameters, holding the identity of a call and its caller, respectively. The operational semantics uses an additional *query* statement, [**await**] **get** u, for dealing with the termination of call/await call statements. The syntax [**await**] denotes an optional **await**. The query **get** u is blocking while waiting for the method response with identity u, and **await get** u is a suspending query.

The state of an object is given by a twin mapping, written $(\alpha|\beta)$, where α is the state of the field variables (including PQ, nextId, nextOb) and class parameters $\overline{cp}$ (including this), and β is the state of the local variables and formal

$$\textsc{return}: \quad o : \mathbf{ob}(\delta, \texttt{return}\ e) \xrightarrow{\delta[\mathsf{caller}] \leftarrow \delta[\mathsf{this}].(\delta[\mathsf{callId}], \delta[e])} o : \mathbf{ob}(\delta, \mathbf{idle})\ \ \texttt{msg}\ \delta[\mathsf{caller}] \leftarrow \delta[\mathsf{this}].(\delta[\mathsf{callId}], \delta[e])$$

$$\textsc{query}: \quad \texttt{msg}\ o \leftarrow o'.(u,d)\ \ o : \mathbf{ob}(\ldots[\texttt{await}]\ v := \texttt{get}\ u\ \ldots) \xrightarrow{o \twoheadleftarrow o'.(u,d)} o : \mathbf{ob}(\ldots v := d \ldots)$$

$$\textsc{await}: \quad o : \mathbf{ob}(\delta, \texttt{await}\ b; \overline{s}) \xrightarrow{empty} o : \mathbf{ob}(\delta, \overline{s})\ \ \texttt{if}\ \delta[b] = true$$

$$\textsc{continue}: \quad o : \mathbf{ob}((\alpha|\beta'), \mathbf{idle}) \xrightarrow{empty} o : \mathbf{ob}((\alpha[PQ \mapsto rest])|\beta), \overline{s})\ \ \texttt{if}\ deq(\alpha[PQ], \alpha) = ((\beta, \overline{s}); rest)$$

$$\textsc{suspend}: \quad o : \mathbf{ob}((\alpha|\beta), \overline{s}) \xrightarrow{empty} o : \mathbf{ob}((\alpha[PQ \mapsto enq(\alpha[PQ], (\beta, \overline{s}))], \varepsilon), \mathbf{idle})\ \ \texttt{if}\ \overline{s}\ \text{starts with}\ \texttt{await}$$

Figure 6.20: Operational rules (Part II) for small-step semantics (without secrecy levels).

parameters (including callId and caller) of the current process. Look-up in a twin mapping, $(\alpha|\beta)[z]$, is simply given by $(\alpha+\beta)[z]$. For an expression e, the notation $\alpha[z := e]$ abbreviates $\alpha[z \mapsto alpha[e]]$, and the notation $(\alpha|\beta)[v := e]$ abbreviates **if** v *in* β **then** $(\alpha \mid \beta[v \mapsto (\alpha|\beta)[e]])$ **else** $(\alpha[v \mapsto (\alpha|\beta)[e]] \mid \beta)$, where *in* is used for testing domain membership.

The *process queue* PQ is the queue of suspended processes, of form $(\beta, \overline{s}))$. The operations $enq(PQ, p)$ and $deq(PQ, \alpha)$ are used to add a process p to the queue, and to select an *enabled* process (if any) from the queue, respectively. The latter results in the sequence $(p; PQ')$ of the selected enabled process p and the remainder of the queue PQ' (depending on the specific scheduling policy), or the empty sequence *empty* if no process is enabled. A process $(\beta, \overline{s})$ is *enabled* if it starts with an enabled statement. A conditional **await** statement is enabled if the condition evaluates to true (in state $\alpha|\beta$), and an **await** call statement is not enabled (unless reduced by the QUERY rule). All other statements are enabled.

Asynchronous (simple) method invocation is captured by the rule SIMPLE CALL/CALL. The generated call identity is locally unique (and globally unique in combination with the parent object). The call identity generated by this rule is passed through an invocation message, which is to be consumed by the callee object by the rule START. When an object has no active process, denoted ***idle***, a suspended process may be continued (by rule CONTINUE), given that

the process is enabled, or a method call is selected for execution by rule START. The invocation message is removed from the configuration by this rule, and the identity of the call is assigned to the implicit parameter callId. With rule RETURN, a return value is generated upon method termination and passed in a completion message together with the call identity stored in callId. The return value is fetched by rule QUERY. Note that a query statement blocks until the corresponding future value is generated by rule RETURN.

The QUERY rule says that an occurrence of **await** $v :=$ **get** u, or $v :=$ **get** u, in object o is replaced by the assignment $v := d$ when the completion **msg** $o \leftarrow o'.(u, d)$ appears. The keyword **await** is removed when in front of such a query statement. Note that rule QUERY removes the completion message from the configuration, which is possible since any corresponding **get** will be found in the object when the completion message appears There is at most one such occurrence (in the first statement of either the active statement list or a process in PQ). If object o does not contain **get** u then the completion message is removed without any effect on o. This happens when the corresponding call was a simple call. In Rule START, we assume that m is bound to a method with local state β (including default values) and code $\overline{s}$. Note that bindings for the parameters $\overline{y}$ and the implicit parameter nextId are added to the local state.

Object creation is captured by the rule NEW. The generated object identity is locally unique, and also globally unique since the object identity is given by a generator term embedding the parent object. The generated object gets this identity. Here $init_C$ denotes the initialization statements (the constructor) of class C, and δ_C denotes the initial state of class C with default/initial values for the fields. The binding of class parameters and this is added explicitly (by this $\mapsto \delta[\text{nextOb}]$ and $\overline{cp} \mapsto \delta[\overline{e}]$). We obtain an active object by letting *init* initiate internal activity, using simple self-calls to allow the object to interleave continued internal activity with reaction to external calls. The initialization statements of a program will typically create the other initial objects.

In the case that an await statement is not enabled, the current process is placed on the process queue and the object becomes *idle*, as described by rule SUSPEND. An idle object may next start a new process (according to rule START) or continue with an enabled process from the process queue (according to rule CONTINUE). This choice depends on the underlying scheduling inside an object. The given language fragment may be extended with constructs for local (stack-based) method calls, e.g., by using the approach of [36]. As we focus on inter-object communication, this is omitted here. For simplicity we omit runtime secrecy levels and therefore the result of evaluating a secrecy comparison (by $\sqsubseteq$) is not defined in the operational semantics.

For a given program (and starting object) the operational semantics defines a set of *executions*, each given by a sequence of global states (configurations). The state of an execution E at time t is the state given by $E[t]$. A sequence of execution steps $E[i] \xrightarrow{e_i} E[i+1] \xrightarrow{e_{i+1}} E[i+2] \xrightarrow{e_{i+2}} \ldots$ generates the trace $e_i; e_{i+1}; e_{i+2} \ldots$. Even if an execution E may be infinite, our analysis will deal with finite segments. In our concurrency model the objects compute

independently at their own speed (when not blocked), and we assume that one object is not unboundedly delayed (unless blocked). Thus for our concurrency model we may assume *inter-object fairness*.

6.9 Related work

As stated in the introduction, programming languages can provide fine-grained control for security issues, and a large amount of work is based on Denning's paper on information flow security [33]. Substantial contributions have been made to prevent disclosure of confidential information based on static, dynamic, or hybrid *information flow control* approaches.

Statically checking information flow to protect confidentiality and integrity is a promising technique as it provides increased precision [33] and low runtime overhead of dynamic security classes [80]. To enforce information flow control policies using static program analysis, program elements are annotated with necessary information. Volpano et al [116] were the first ones to formulate Denning's approach [33] based on program certification, as a type system and proved soundness of a version of non-interference theorem for a core deterministic language. To track information flow in Java, Myer [80] extended the type system of Java and proposed a *decentralized label model*. The extended type system was later implemented in Jif compiler. In another adaptation, type system of functional language OCaml, was extended to Flow Caml [104] by annotating ML types with security labels. Major challenges pertaining to static approaches are that they usually require complicated type annotations and often result in a significant degree of false positives [115]. Although, theorem proving techniques are used in [31, 64] to improve precision of static program analysis.

In contrast, dynamic mechanisms such as [8, 34] are more permissive, imposing high overhead and may require changes to the runtime systems, e.g. special schedulers. Additionally, in a majority of real time systems, security policies vary dynamically [122] and cannot be determined at compile time. In [122] Zheng et al expanded the scope of information flow control by providing mechanism to update label values of program elements during run time. Tse and Zdancewic facilitate dynamic flow control by proving non-interference for a security-typed lambda calculus with runtime principals and enable more expressive security polices [113]. Sabelfeld and Russo [97] compare and contrast static with dynamic program analysis, and deduce (using simple imperative language) that both techniques assure the same level of termination-insensitive non-interference.

Exploration of both static and dynamic approaches are made in [95], and hybrid mechanisms such as [14, 20, 99], are provided to enhance the information flow capability and increasing permissiveness, by realizing static analysis by security type systems and realizing dynamic analysis by monitors. This hybrid approach was also employed in the development of a new system and language, Fabric [73], which is used to build secure distributed information systems. Fusion of static and dynamic mechanisms of analysis for concurrent programs has been proposed by Guernic [48], using an automaton to monitor the information flow

for a single execution of a concurrent program. Most of the work in programming language research that provides information flow control is based on the principle of *non-interference*.

M. Miller [78] explores language-based capabilities in the context of the object capability model in his Ph.D. thesis. This model is useful for investigating object communication and computation aspects. However, it focuses on robustness issues rather than security issues. Additionally, Hammer and Snelting [51] propose information flow control based on program dependency graphs, and demonstrates significant reduction in annotation overhead and improved program analysis precision [50].

Our proposed approach falls in the category of static analysis. However, in this approach we have prevented a high false positive rate since, due to hiding and encapsulation in our distributed object-oriented setting, we do not impose unnecessary restrictions on information flow inside objects. In addition to a new security type- and effect-system for the considered language, we propose a new kind of class-wise trace analysis to restrict the flow of control among objects communicating by asynchronous methods, to avoid indirect leakage observable at the network level. Moreover, while most of the related work aims at preventing traditional progress-insensitive non-interference, we are considering progress-sensitive non-interference, where an attacker can indirectly observe the progress of an object, caused by e.g. process termination or suspension. To the best of our knowledge, there is no prior work considering a concept similar to interaction non-interference, which stipulates indistinguishability of interactions between distributed objects for a network viewer observing the messages exchanged through method calls on the communication channels in the network, given that the communicated low input values are the same.

In general, techniques that come with different goals might also have some similarities. For example, model-based verification has a significant different goal than our work. For instance, the main difference between our work and [46] is that in our case there is no exploitable bug in the program. Instead, we consider legal program behavior that might be informative to attackers. The attack model and assumptions make remarkable differences as well. For example, in our setting, there is no direct interaction with the malicious agent, the interaction is indirect though asynchronous method calls. With respect to non-interference, we are focusing on security leakages where the attacker is capable of observing the interactions between agents, which for example is totally different from works where attackers have interactions with the system, e.g. [117]. However, by our trace analysis, we are also looking at runtime state changes and try to prevent reaching states that may result in information leakage according to interaction non-interference. This has similarities to to model-based verification techniques such as Hoare logic, model checking, or state-based analysis approaches. In contrast to Hoare logic we avoid verification conditions requiring (non-trivial) theorem proving, and in contrast to model checking we transform a program to a trace expression used for further analysis.

6.10 Conclusion

We have studied non-interference for concurrent distributed object systems communicating by means of asynchronous method calls. The concurrent objects may communicate confidential and non-confidential information, restricting confidential information to method parameters/returns declared as safe channels for confidential information. Due to the non-deterministic nature of such systems and due to the non-trivial implicit information flow leakage related to observation of communication patterns, standard definitions of non-interference are not suitable. We have defined a notion of *interaction non-interference* and have shown how to enforce it by static analysis, using a type and effect system for secrecy levels and using analysis of communication traces addressing indirect network leakage. The analysis is modular and is done class-wise, and we have outlined a proof for soundness. We have considered an object-oriented language centered around the chosen concurrency model. The setting of concurrent objects and object-orientation gives some benefits as well as some challenges, compared to other settings. The benefits include:

- *protection of state.* Each object encapsulates its state in the sense that remote access is not allowed. This means that we do not restrict information flow between confidential and non-confidential variables as long as they do not affect communication behavior to cause a leakage. All fields are private and their secrecy level may vary dynamically with the static knowledge of their values and with the implicit context of high level if- and while-tests.
- *concurrency control.* The high level await mechanism allows cooperative scheduling with explicit control of process suspension and resumption without low level mechanism such as locking or signaling mechanisms. This enables a compositional analysis.
- *message-oriented communication.* The underlying message passing mechanism for method interaction defines one-way as well as two-way interactions. Based on our analysis, the implicit leakage at the network level can then be addressed by expressing communication traces for each method.
- *inflation of high levels.* Our approach does not lead to inflation of high levels, since method calls in a high context do not require methods to be high, and since fields and program variables may go from a high level to a low level.

However, this setting implied some challenges, which we have addressed:

- *secrecy level invariants.* The presence of suspension points imply that secrecy levels of fields may change during suspension. We use an approach similar to class invariants for controlling the level of fields during suspension. Secrecy levels of fields must be maintained upon suspension and method completion. In contrast, the levels of local variables may change freely since their values are not modified during suspension.

- *modularity.* For a given object, the precise timing of observable input events, reflecting method invocation and completion to the object, cannot be detected statically since these events cannot be determined from the program code. This makes the trace analysis less direct. We solve this by considering trace expressions that include reaction events. These are by definition non-observable, but give partial information about the timing of the corresponding input events, which makes the analysis less direct.

- *implicit self-calls.* Self-calls pose non-trivial challenges for the modular analysis, since the static analysis cannot in general detect if a call $o.m(\ldots)$ is a self-call or not, and since a self-call may indirectly have observable effects (when the called method calls external objects). We solve this by including self-calls in the trace expressions, making special considerations for calls detected as self-calls (i.e., calls to this).

The considered language is small, but includes mechanisms for process control, which often is defined by the underlying operating system. With a dedicated virtual machine this makes it possible to limit attacks from within the underlying operating system.

Future work As future work, we will consider other language features such as inheritance, which was not considered here, enrich our static analysis, and providing a hybrid approach to satisfy the interactive non-interference policy combining runtime and static analysis. The latter point requires an operational semantics assigning runtime level to objects as well as to values of program variables.

Inheritance and late binding will complicate the analysis in that the binding of a method call is not in general static. As our approach depends on static binding to be able to compute the traces, it cannot be extended to deal with inheritance in a straight forward manner. However, the approach for partial correctness reasoning used in [88, 89] allows modular reasoning for each (sub)class with static binding of self calls, based on the assumption that the runtime class of the considered object is the same as the considered class. Calls other than (explicit) self-calls are statically controlled by interfaces (as in SeCreol). This means that trace sets of inherited methods need to be recalculated in a subclass (due to possible renewed bindings of Self-calls). This would create problems in the frameworks such as behavioral subtyping or lazy behavioral subtyping [37]. Using the approach of [89], we may extend the current class-wise static analysis of non-interference to deal with inheritance and late binding.

The concept of cointerface was used in the language to allow type-correct callbacks to external caller objects. This concept gives the possibility of stating minimal security requirements to callers of methods of an interface. We would like to explore this possibility in future work.

To enrich the static analysis, we aim to use the program dependency graphs (PDGs) by considering the effects of program control flow and data flow [94] on interactive communication among objects with security levels which is also

proven at least as powerful as security type systems in detecting potential information flows [74] while it can decrease false positives even more in progress-sensitive approaches. In addition, to improve the enforcement, we consider dynamic labeling and decentralized label model (DLM) [73] to provide a hybrid enforcement mechanism as future work.

Acknowledgements. This work was partially supported by the project IoTSec - Security in IoT for Smart Grids, with number 248113/O70 part of the IKTPLUSS program funded by the Norwegian Research Council, and by the project SCOTT (www.scott-project.eu) funded by the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No 737422. This Joint Undertaking receives support from the European Unions Horizon 2020 research and innovation programme of Austria, Spain, Finland, Ireland, Sweden, Germany, Poland, Portugal, Netherlands, Belgium, and Norway.

Authors' addresses

Toktam Ramezanifarkhani University of Oslo, Postboks 1337 Blindern, 0316 Oslo, Norway, toktamr@uio.no

Olaf Owe University of Oslo, Postboks 1337 Blindern, 0316 Oslo, Norway, olaf@uio.no

Shukun Tokas University of Oslo, Postboks 1337 Blindern, 0316 Oslo, Norway, shukunt@uio.no

Chapter 7

Language-based mechanisms for privacy by design

Shukun Tokas, Olaf Owe, Toktam Ramezanifarkhani

Published in *Proceedings of the 14th IFIP International Summer School on Privacy and Identity Management*, June 2019, volume 576, pp. 142–158. DOI: 10.1007/978-3-030-42504-3_10.

II

Abstract

The privacy by design principle has been applied in system engineering. In this paper, we follow this principle, by integrating necessary safeguards into the program system design. These safeguards are then used in the processing of personal information. In particular, we use a formal language-based approach with static analysis to enforce privacy requirements. To make a general solution, we consider a high-level modeling language for distributed service-oriented systems, building on the paradigm of active objects. The language is then extended to support specification of policies on program constructs and policy enforcement. For this we develop i) language constructs to formally specify privacy restrictions, thereby obtaining a policy definition language, ii) a formal notion of policy compliance, and iii) a type and effect system for enforcing and analyzing a program's compliance with the stated polices.

7.1 Introduction

Advances in information technologies have often led to concerns about privacy. With the adoption of information and communication technology in our daily lives, the gathering and processing of personal information fundamentally increases the potential for privacy threats. In particular, privacy and data protection features are often ignored by conventional engineering approaches [30] or accommodated as an afterthought. Aligning the software ecosystem with the privacy-related requirements is an essential step towards better data protection. In order to endorse privacy as a first-class requirement and promote privacy compliance from the outset of product development, the *privacy by design* (PbD) requirement has been formally embedded in the GDPR regulations (Article 25 [41]). Article

The authors were partially supported by IoT-Sec (NRC) (https://its-wiki.no/wiki/IoTSec:Home) and SCOTT (EU)(www.scott-project.eu).

25 [41] obliges the controllers to design and develop products with a built-in ability to demonstrate compliance towards the data protection obligations.

The main idea of privacy by design is to make privacy a key consideration in development of systems. Privacy by design is a framework consisting of seven foundational principles: *i)* proactive not reactive; preventive not remedial, *ii)* privacy as default setting, *iii)* privacy embedded into design, *iv)* full functionality - positive-sum, not zero-sum, *v)* end-to-end security - full lifecycle protection, *vi)* visibility and transparency, and *vii)* respect for user privacy - keep it user-centric. We focus on the *privacy embedded-into-design* principle, due to its potential connection with language mechanisms. We explore the idea of adding privacy requirements into programming/specification languages and use static analysis for enforcing such privacy requirements.

In this paper, we follow the privacy by design principle, by integrating necessary safeguards into the processing of personal information, using a language-based approach. In particular, we explore how to formalize fundamental privacy principles and to provide built-in abilities to fulfill data protection obligations. As a step towards this goal we develop a policy specification language that provide constructs for specifying privacy requirements on sensitive (personal) data In particular, a policy is given by a set of triples that put restrictions on the *principals* that may access the information for certain *purposes* and the permitted *access rights*. Such policy statements are then linked with language constructs of a high-level modeling language oriented towards distributed and service-oriented systems. Policies are annotated with the *data types* and *methods*.

Certain aspects of privacy restrictions can be expressed by means of static concepts, while others can only be expressed at runtime, such as *data subject*, *consent*, and other user-defined changes. In this paper, we focus on statically declared policies and implicit consent (at compile time presence of policy implies consent). Changes in consent and policies are handled at runtime through predefined functionalities, which is beyond the scope of this article. In addition to *read* and *write* access, we consider *incremental* access (*incr*), allowing addition of sensitive information without read access and without modifying existing information. For instance, in a healthcare setting, a lab assistant may have incr access to treatment data, while a nurse may have both read and incremental access ($read \sqcup incr$), and a doctor may have full access ($read \sqcup write$). We formalize a notion of policy compliance, to develop a scheme of policy inheritance. Finally, to enforce policy compliance, we define a set of rules, i.e., the type and effect system that checks that the policies are respected when the sensitive information is accessed. The theory of the current work is presented in more details in [**pricreol19**].

In summary, the main idea is to provide language constructs that express privacy policy specifications capturing static aspects of privacy and use these to statically analyze a program's compliance with the policy specifications. We make the following contributions: i) propose a policy language for specifying purpose, access and policy requirements (see Figure 9.1), ii) formalize a notion of policy compliance, iii) show how the policy language can be used with an underlying object-oriented language, and iv) develop a mechanic type and effect

A	$::=$	$read \mid incr \mid write \mid self$	basic access rights
		$\mid no \mid full \mid rincr \mid wincr$	abbreviated access rights
		$\mid A \sqcap A \mid A \sqcup A$	combined access rights
$\mathcal{P}$	$::=$	(I, R, A)	policy
$\mathcal{P}s$	$::=$	$\{\mathcal{P}^*\} \mid \mathcal{P}s \sqcap \mathcal{P}s \mid \mathcal{P}s \sqcup \mathcal{P}s$	policy set
$\mathcal{RD}$	$::=$	**purpose** R^+	
		[**where** Rel [**and** Rel]*]	purpose declaration
Rel	$::=$	$R^+ < R^+$	sub-purpose declaration

Figure 7.1: BNF syntax definition of the policy language. I ranges over interface names and R over purpose names. The operators $\sqcup$ and $\sqcap$ denote join and meet, respectively.

system for analyzing a program's compliance with the annotated privacy policies. *Paper outline.* The rest of the paper is structured as follows. Section 7.2 presents the formalization of privacy policies, including a policy definition language and a formalization of policy compliance. Section 7.3 introduces the core language, with support for the specification of privacy principles. Section 7.4 presents the type and effect system. Section 7.5 demonstrates the analysis on a small case study. Section 8.8 discusses related work, and Section 8.9 concludes the paper.

7.2 Language constructs for policy specification

Privacy policies are often described in natural language statements. To verify formally that the program satisfies the privacy specification, the desired notions of privacy need to be expressed explicitly. To formalize such policies, we define a policy specification language. Furthermore, to establish a link between policies and programming language constructs, we extend the syntax and semantics of a small core language (see Section 7.3). In our setting, a privacy policy is a statement that expresses permitted use of the sensitive information by the declared program entities. To support privacy-by-design, we define policies at the design level, and associate policies to data types and methods of interfaces and classes, such that the policies of a method in a class must comply with the corresponding policy in an interface of the class. In particular, a policy is given by a set of triples that put restrictions on: What *principals* may access the sensitive data, which *purposes* are allowed, and which *access-rights* are permitted. That being the case, a policy $\mathcal{P}$ is given by a triple (I, R, A), where *i)* I ranges over interfaces, which are organized in an open-ended inheritance hierarchy, *ii)* R ranges over purposes, which are organized in a hierarchy (reflecting specialization), and *iii)* A ranges over access rights, which are organized in a lattice. Thus principals are expressed by the Interfaces, while new language constructs are added to represent purposes, access rights, and policies.

The language syntax for policies is summarized in Figure 9.1, where [ ] is used

as meta-parenthesis, and superscripts * and $^+$ denote general and non-empty repetition, respectively. Here we briefly discuss the specification constructs.

Principal describes the roles that can access sensitive information and is given by an interface. For instance for a call $x := o.m(\bar{e})$, where o is typed by an interface with policy (I, R, A), the caller object must support interface I. Interfaces are organized in an open-ended inheritance hierarchy, letting $I < J$ denote that I is a subinterface of J. For example,

$$Specialist < Doctor < HealthWorker$$

Any is predefined as the least specialized interface, i.e., the superinterface of all interfaces. We let $\leq$ denote the transitive and reflexive extension of $<$.

Purpose names are used to restrict usage of sensitive data to specific purposes. Such purpose names can be organized in a hierarchical structure, reflecting a *purpose hierarchy* [53]. We let purposes be organized in a directed acyclic graph reflecting specialization. Purpose names are defined by the keyword **purpose**. For instance, the declaration

$$\textbf{purpose}\ spl_treatm, treatm\ \textbf{where}\ spl_treatm < treatm$$

makes *spl_treatm* more specialized purpose than *treatm*. If data is collected for the purpose of *spl_treatm* then it cannot be used for *treatm*. However, if it is collected for the purpose of *treatm* then it can be used for *spl_treatm*. We let $\leq$ denote the transitive and reflexive extension of $<$.

Access-right describes permitted operations on sensitive data. Access rights are given by a lattice, with meet and join operations (see Figure 8.3): *read* gives read access, *write* gives write access (without including read access), *incr* allows addition of new information but neither read nor write is included. The combination of *read* and *incr*, i.e., $read \sqcup incr$ is abbreviated *rincr* gives read and incremental access. Similarly, $write \sqcup incr$ is abbreviated as *wincr*, which gives write and incremental access. Full access is given by a combination of *read* and *write* (which includes incremental access), i.e., *full* is the same as $read \sqcup write$. These general access rights can be combined with access rights on self, i.e., access rights when the principal is the subject herself. (details are omitted). For instance, a nurse should be able to see treatment data of a patient and add new data, and needs *rincr* access, while a lab assistant may add lab data and needs only *incr* access. A patient should see data about herself, which requires $self \sqcap read$.

A single policy ($\mathcal{P}$) is given by a triple (I, R, A), and a policy set ($\mathcal{P}_s$) is given by a set of policy triples (with meet and join operation defined). For our purposes, we annotate methods with single policies while data types are annotated with policy sets reflecting the permitted usage by different principals.

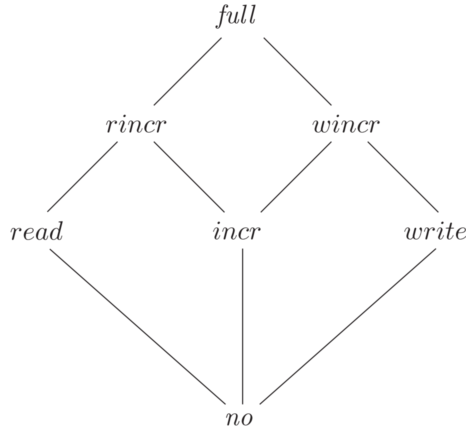

Figure 7.2: The lattice for general access rights (without *self*). Note that *rincr* is the same as $read \sqcup incr$, *wincr* is the same as $write \sqcup incr$, and *full* is the same as $read \sqcup write$.

Example. The example in Figure 8.5 gives an illustration for declaring policies, and annotating methods and types with policies. The policy $(Doctor, treatm, rincr)$ restricts access to objects typed by the *Doctor* interface, for only *treatm* (treatment) purposes, and with *rincr* data access. This is checked by a type and effect system in section 7.4. The policy set

$$\{(Doctor, treatm, full), (Doctor, treatm, rincr), (Nurse, treatm, read)\}$$

restricts access by these three policies. Here, the policy $(Doctor, treatm, rincr)$ is redundant since $(Doctor, treatm, rincr) \sqsubseteq (Doctor, treatm, full)$, and is colored grey to indicate that. Method *makePresc* has policy $(Doctor, treatm, rincr)$, meaning that this method must be called by a Doctor object (or a more specialized object), for purposes of treatment and with read and incremental access (but not write access). Thus a doctor can add new prescription, but not change or remove old ones. Method *getPresc* has policy $(Nurse, treatm, read)$, meaning that this method must be called by a Nurse object (or a more specialized object such as a Doctor object), for purposes of treatment, and with read-only access. These two methods, with associated policies, are inherited in interface *PatientData*.

7.2.1 Policy compliance definition

Here, we briefly present a few definitions needed to express policy compliance.

Definition 6 (Policy Compliance)**.** *The sub-policy relation* $\sqsubseteq$*, expressing* policy compliance*, is defined by*

$$(I', R', A') \sqsubseteq (I, R, A) \equiv I \leq I' \wedge R' \leq R \wedge A' \sqsubseteq A$$

```
purpose basic_treatm, treatm where basic_treatm < treatm

policy P_Doc = (Any, treatm, full)
policy P_AddPresc = (Doctor, treatm, rincr)
policy P_GetPresc = (Nurse, treatm, read)
policy P_Presc = {P_GetPresc, P_AddPresc, P_Doc}

type Presc == Patient * String :: P_Presc

interface Patient extends Subject {Void getSelfData() :: P_SelfPresc}
interface AddPresc {Void makePresc(Presc newp):: P_AddPresc}
interface GetPresc {Presc getPresc(Patient p) :: P_GetPresc}
interface PatientData extends AddPresc, GetPresc {}
interface Nurse extends Principal { Presc nurseTask() :: P_GetPresc}
interface Doctor extends Nurse{ Void doctorTask(Patient p) :: P_Doc}

class PATIENTDATA() implements PatientData {
  type PData = List[Presc] :: P_Presc
  PData pd = empty();
  Presc getPresc(Patient p){return last(pd/p)} :: P_GetPresc
  Void makePresc(Presc newp) {
     if newp ≠emptyString() then pd:+ newp fi } :: P_AddPresc }

class DOCTOR() extends NURSE implements Doctor{//inherits pd
  Void doctorTask(Patient p){
    Presc oldp = pdb.getPresc(p);
    String text = ...; //new presc using symptoms info and oldp
    Presc newp = (p, text); // here, new sensitive data is created!
    pdb!makePresc(newp)}:: P_Doc }
```

Figure 7.3: Interface, class, type, and policy definitions for the Prescription Example. Grey policy specifications are implicit while underlined ones need to be explicitly stated. A class implementation of Nurse is omitted. The projection pd/p is the list of strings associated to patient p, and the function *last* gives the last element.

(where the last $\sqsubseteq$ operation is on access rights) with $\bullet$ as bottom element*, representing non-sensitive information. It follows that $\sqsubseteq$ is a partial order.*

A policy $\mathcal{P}'$ complies with $\mathcal{P}$ if it has the same or larger interface, the same or more specialized purpose, and if the access rights of $\mathcal{P}'$ are the same or weaker than that of $\mathcal{P}$. In particular, the policy of the implementation of a method should comply with that of the interface. Note that $\bullet \sqsubseteq \mathcal{P}$ expresses that an implementation without access to sensitive information complies with any policy.

Moreover, the use of *self* in the access part allows us to distinguish between

different kinds of self access for different purposes, such as $(Patient, all, read \sqcap self)$ and $(Patient, private_settings, self)$. The latter gives full access to data about *self* for purposes of *private settings*, while the first gives read access to data about *self* for all purposes.

We define a lattice over sets of policies with meet and join operations, and generalize the definition of compliance to sets of policies:

Definition 7 (Compliance of Policy Sets)**.**

$$\{\mathcal{P}'_i\} \sqsubseteq \{\mathcal{P}_j\} \equiv \forall i \,.\, \exists j \,.\, \mathcal{P}'_i \sqsubseteq \mathcal{P}_j$$

This expresses that a policy set S' complies with a policy set S if each policy in S' complies with some policy in S. We define meet and join operations over policy sets by set union and a kind of intersection, respectively, adding implicitly derivable policies:

Definition 8 (Join and Meet over Policy Sets)**.**

$$S \sqcup S' \equiv closure(S \cup S')$$

$$S \sqcap S' \equiv closure(\{P \mid P \sqsubseteq S \wedge P \sqsubseteq S'\})$$

where the closure operation *is defined by*

$$closure(S) \equiv S \cup \{(I, R, A \sqcup A') \mid (I, R, A) \sqsubseteq S \wedge (I, R, A') \sqsubseteq S\}$$

We have a lattice with $\emptyset$ as the bottom element. The closure operation adds implicitly derivable policies, and ensures that $\{(I, R, A \sqcup A')\} \sqsubseteq \{(I, R, A)\} \sqcup \{(I, R, A')\}$. For instance, $\{(Doctor, treatm, read)\} \sqcup \{(Doctor, treatm, write)\}$ is the same as $\{(Doctor, treatm, full)\}$. These constructs are useful in specification of constraints and in capturing access to sensitive information with declared privacy policies. The meet operation typically reflects worst-case analysis.

Definition 9 (Implication on Policy Set)**.** *We define the notation $\mathcal{P}s' \Longrightarrow \mathcal{P}s$ ($\mathcal{P}s'$ implies $\mathcal{P}s$) by $\{\bullet\} \Longrightarrow \mathcal{P}s$ and $\mathcal{P}s \sqsubseteq \mathcal{P}s'$ for $\mathcal{P}s'$ other than $\{\bullet\}$.*

Implication is used to check policy compliance of an actual parameter with respect to a formal parameter. If $\{\bullet\}$ is the policy on the actual parameter and $\mathcal{P}_{doc}$ the policy on the formal parameter, we will check $\{\bullet\} \Longrightarrow \mathcal{P}_{doc}$.

Policies on methods. Let $\mathcal{P}_{I,m}$ denote the policy of a method m given in an interface I, and $\mathcal{P}_{C,m}$ denote the policy of a method m given in a class C. We will require that the implementation of a method in a class (C) respects the policy stated in the interface (I), i.e., $\mathcal{P}_{C,m} \sqsubseteq \mathcal{P}_{I,m}$. And we also require that a method redefined in an interface (I) respects the policy of that method in a superinterface (J), i.e., $\mathcal{P}_{I,m} \sqsubseteq \mathcal{P}_{J,m}$. By transitivity of $\sqsubseteq$, a method implementation in a class that respects the policy given in an interface also respects the policy of the method given in a superinterface, i.e., $\mathcal{P}_{C,m} \sqsubseteq \mathcal{P}_{I,m}$ and $\mathcal{P}_{I,m} \sqsubseteq \mathcal{P}_{J,m}$

implies $\mathcal{P}_{C,m} \sqsubseteq \mathcal{P}_{J,m}$. For instance, consider an interface *GetPresc* with a method *getPresc*() with policy $(Nurse, treatm, read)$. An implementation of this method in a class must have a policy that complies with it, such as $(Any, treatm, read)$, $(Nurse, treatm, self \sqcap read)$, or $(Nurse, basic_treatm, read)$. In contrast, the implementation cannot have policy $(Doctor, treatm, read)$, as this would not allow a $Nurse$ as the caller object, and also not $(Nurse, all, rincr)$, because this violates purpose and access restrictions.

Policies on types. We let the policy of a type T, denoted $\mathcal{P}_T$, be a policy set. Let the policy set $\{(Doctor, treatm, rincr), (Nurse, treatm, read)\}$ be the policy set on type *Presc*. This allows the data of type *Presc* to be accessed based on these two policies, depending on the calling context. For instance, if the caller is a *Doctor* object and the purpose is *treatm* then *read* as well as *incr* access is allowed on data of type *Presc*. The policy set of an actual variable must imply the policy set of the type of the corresponding formal variable. Together, the policies on methods and types provide sufficient abstractions to control access to sensitive data.

In the next section we consider a high-level imperative language for service-oriented systems where policy specifications are integrated.

7.3 Embedding policy with program constructs

We target object-oriented, distributed systems (OODS) and consider the active object programming paradigm [85], which is based on the actor model [57] and gives a high-level view of communication aspects in OODS. In the active object model, objects are autonomous and execute in parallel, communicating by so-called asynchronous method invocations. We assume interface abstraction, i.e., an object can only be accessed through an interface and remote field access is illegal. This allows us to focus on major challenges of modern architectures, without the complications of low-level language constructs related to the shared-variable concurrency model.

We propose a small core language, based on Creol [61], centered around a few basic statements. It has a compositional semantics which is beneficial to analysis [61, 93]. The language is imperative and strongly typed, with data types for data structure locally inside a class. The data type sublanguage is side-effect-free. The motivation is that the language gives high-level descriptions of distributed systems and synchronous and asynchronous interaction based on methods, thereby avoiding shared variable access, and avoiding explicit signaling and notification. The BNF syntax of the language is summarized in Figure 7.4. As before, optional parts are written in brackets (except for type parameters, as in List[T], where the brackets are ground symbols). Class parameters (z), method parameters (y) the implicit class parameter this and the implicit method parameter caller are read-only. A class may implement a number of interfaces, and for each method of an interface (of the class) it is required that the class defines the method such that policy of each method parameter and return value

Pr	$::=$	$[\mathcal{T} \mid \mathcal{RD} \mid In \mid Cl]^*$	program
$\mathcal{T}$	$::=$	$\textbf{type}\, N\, [\overline{T}] =<type_expression> [::\mathcal{P}s]$	type definition
T	$::=$	$I \mid \mathsf{Int} \mid \mathsf{Any} \mid \mathsf{Bool} \mid \mathsf{String} \mid \mathsf{Void} \mid \mathsf{List}[T] \mid N$	types
In	$::=$	$\textbf{interface}\ I\ [\textbf{extends}\ I^+]\ \{D^*\}$	interface declaration
Cl	$::=$	$\textbf{class}\ C\ ([T\ z[::\mathcal{P}]]^*)$	class definition
		$[\textbf{implements}\ I^+]\ [\textbf{extends}\ C]$	support, inheritance
		$\{[T\ w\ [:= ini]\ [::\mathcal{P}]]^*$	fields
		$[B\ [::\mathcal{P}]]$	class constructor
		$[[\textbf{with}\ I]\ M]^*\}$	methods
D	$::=$	$T\ m([T\ y\,[::\mathcal{P}]]^*)\ [::\mathcal{P}]$	method signature
M	$::=$	$T\ m([T\ y\,[::\mathcal{P}]]^*)\ [\{s\}]\ [::\mathcal{P}]$	method definition
B	$::=$	$\{[T\ x\ [:= rhs];]^*\ [s;]\ \textbf{return}\ rhs\}$	method blocks
v	$::=$	$w \mid x$	assignable variable
e	$::=$	$v \mid y \mid z \mid \mathsf{this} \mid \mathsf{caller} \mid \mathsf{void} \mid f(\overline{e})$	pure expressions
ini	$::=$	$e \mid \textbf{new}\ C(\overline{e})$	initial value of field
rhs	$::=$	$ini \mid e.m(\overline{e})$	right-hand sides
s	$::=$	$\mathsf{skip} \mid s; s$	sequence
		$\mid v := rhs \mid v :+ rhs \mid e!m(\overline{e}) \mid I!m(\overline{e})$	assignment and call
		$\mid \textbf{if}\ e\ \textbf{then}\ s\ [\textbf{else}\ s]\ \textbf{fi}$	if statement
		$\mid \textbf{while}\ e\ \textbf{do}\ s\ \textbf{od}$	while statement

Figure 7.4: BNF syntax of the core language. A field variable is denoted w, a local variable x, a method parameter y, a class parameter z, and list append is denoted $+$. The brackets in $[T]$ and $[\overline{T}]$ are ground symbols.

are respected. Additional methods may be defined in a class, but these may not be called from outside the class. The language supports single class inheritance and multiple interface inheritance (using the keyword **extends**). Below, we give BNF syntax for method and type declarations.

Definition 10 (Method Declaration Syntax)**.**

$$T\ m([Y\ \ y]^*)\ [::\mathcal{P}]$$

where T is the result type and Y is the type of parameter y.

An inherited method m inherits the policy of m from the superinterface, unless the interface declares its own policy for m. However, the redefined policy of m (of interface I) cannot be more restrictive than that of the superinterface (J), i.e., $\mathcal{P}_{I,m} \sqsubseteq \mathcal{P}_{J,m}$, ensuring that a class implementation of m satisfying $\mathcal{P}_{I,m}$ also satisfies any declarations of m in a superinterface.

Definition 11 (Data Type Declaration Syntax and Sensitivity)**.**

$$\textbf{type}\ N\ [TypeParameters] =<\ type_definition\ >\ \ [::\mathcal{P}s]$$

where the type parameters are optional. The predefined basic types (Nat, Int,

String, Bool, Void) are non-sensitive*. A user-defined type is* sensitive *if a policy set is specified in the type definition.*

For example, a sensitive *String* type restricted by a policy $\mathcal{P}_s$ can be defined by

$$\texttt{type}\ Info\ = String :: \mathcal{P}_s$$

and encryption could go from Info to String, and decryption the other way.

We consider next *sensitive* functions, which create new sensitive data, for instance a product of individually non-sensitive data may be sensitive. Generator functions (here called constructors) are considered *sensitive* if they i) combine information about a subject with non-sensitive or sensitive information or ii) use sensitive information. We assume that sensitive generators produce sensitive types (with some exceptions, such as constructors of encrypted data). Defined functions are *sensitive* if their type is sensitive and the definition directly or indirectly contains a sensitive application of a constructor. For instance we may (recursively) define a parameterized list type by $List[T] = empty()|append(List[T]* T)$ meaning that lists have the form $empty()$ or $append(l, x)$, where l is a list and x a value of type T. (We let the notation $l + x$ abbreviate $append(l, x)$.) The list is sensitive if T is sensitive, but the append constructor function is not sensitive. A pair product type can be defined by $PatientData = (Patient * String)$ where $Patient$ is a interface representing a data *subject*. This type is sensitive (even though $String$ is not), and the pair ($current_patient$, "*no health problems*") is a sensitive application of the product constructor. These examples suffice for our purposes here. It can be detected statically if a function is sensitive (further details are omitted). Applications of sensitive functions may create new sensitive data, something which require write access. This way the policy control is driven by the declared data types rather than variable declarations. Data types are reusable and therefore their policies are likely to more reliable and appropriate than one-time adhoc specification for program variables.

When the lawful basis of processing of personal information is performance of contract or other valid bases but not the consent, the policies must be formulated in a way that ensures that they are built into the system *by default*, i.e., no measures are required by the data subject in order to maintain his/her privacy. However, when consent is the basis of processing the data subjects, choices in privacy settings are captured at runtime (as outlined in [106]).

We next show how to define static policy checking for our core langauge.

7.4 An effect system for privacy

We propose static policy checking defined by a set of syntax-directed rules, given as a type and effect system [84], but dealing with policies rather than types. We consider two kinds of judgments. For a statement s, the judgment

$$C, m \vdash [\Gamma]\ s\ [\Gamma']$$

expresses that inside a method body m and an enclosing class C, the statement(list) s when started in a state satisfying the environment Γ results

$$\frac{read \sqsubseteq \Gamma[v] \sqcap (\mathcal{P}_{C,m}@(C,m))}{C,m \vdash [\Gamma]\ v ::\ \Gamma[v] \sqcap \Gamma[pc]}\ \text{(P-VAR)}$$

$$\frac{\begin{array}{ll} C,m \vdash [\Gamma]\ e_i ::\ \mathcal{P} & \text{for each argument } e_i \text{ of a sensitive type} \\ write \sqsubseteq \mathcal{P}_T \sqcap (\mathcal{P}_{C,m}@(C,m)) & \text{if } f_T \text{ is a sensitive function} \end{array}}{C,m \vdash [\Gamma]\ f_T(\overline{e}) ::\ \mathcal{P}_T \sqcap \Gamma[pc]}\ \text{(P-FUNC)}$$

$$\frac{\begin{array}{c} \mathcal{P}_{I,n} \sqsubseteq_{Co,R}\ \mathcal{P}_{C,m}@(C,m) \\ C,m \vdash [\Gamma]\ e :: \mathcal{P}' \\ C,m \vdash [\Gamma]\ e_i :: \mathcal{P}_i \quad \mathcal{P}_i \Longrightarrow \mathcal{P}_{par(I,n)_i} \quad \text{for each i} \end{array}}{C,m \vdash [\Gamma]\quad e.n_I(\overline{e}) :: \mathcal{P}_{out(I,n)}}\ \text{(P-CALL)}$$

Figure 7.5: Policy Rules for Expressions and Right-Hand Sides.

in a state satisfying the environment Γ'. Here Γ is a mapping from program variable names to policy sets, such that the policy set of a variable in a given state gives an upper bound of the permitted operations. In order to deal with branches of if- and while-statements where the context policy is influenced by that of the if- and while-tests, Γ uses an additional variable pc (the program context) reflecting the current branching policy (as in [93]). Note that the rules are right-constructive in the sense that Γ' can be constructed from Γ and s.

For an expression or right-hand side e, the judgment

$$C,m \vdash [\Gamma]\ e :: \mathcal{P}s$$

expresses that the evaluation of e in a state satisfying Γ gives a value satisfying the policy set $\mathcal{P}s$, where m is the enclosing method and C the enclosing class.

Figure 8.9 defines the typing rules for expressions and right-hand sides, and Figure 8.10 defines the typing rules for (selected) statements. We let $\mathcal{P}_{I,m}$ denote the policy of method m of interface I, $\mathcal{P}_{C,m}$ denote the policy of method m of class C, and $\mathcal{P}_T$ denote the policy associated with a type T. If no policy is specified for any declaration, we understand that there is no sensitive information, i.e., the policy is $\{\bullet\}$. Data types with sensitive constructors will be considered sensitive. A non-sensitive method would not be able to access or create sensitive data, and a non-sensitive type declaration would not allow assignment of sensitive information to variables of that type.

Rule P-VAR says that the policy of a variable v (a field, parameter, or local variable) is the one recorded in Γ for v, i.e., $\Gamma[v]$, combined with that of the program context pc. The premise states that there must be read access to v, both according to the policy set of the variable and according to the policy set of the enclosing method body. If the policy of the enclosing method m is (I,R,A), the *policy set of the method body* is defined by

$$(I,R,A)@(C,m) \equiv (I,R,A)\ \cup\ (\cup_i\ \{(I_i,R,A)\})$$

where I_i ranges over all the interfaces of C that export m. Thus the policy set of the method body is that of the method and those where this object is the principle (as seen through one of the interfaces exporting m).

Rule P-FUNC says that the policy of a function application $f_T(\overline{e})$ is that of the resulting type T (detected by ordinary typing) combined with that of the program context *pc*. Sensitive arguments must be checked (which ensure read access to the variables occurring in these arguments), and in case f is a sensitive function application, there must be write access according to the policy of T and the policy of the method body. Constants (function without arguments), as well as object creation, have policy set $\{\bullet\}$.

Rule P-CALL says that the policy of a remote call $e.n_I(\overline{e})$ where I is the interface of the method (detected by ordinary typing), is the policy on the return type of the method (as given by the declaration of m in I). The first premise ensures that the policy of the called method complies with policy of the enclosing body. The second premise ensures that the callee expression has a valid policy, and the last premise ensures each actual parameter has a policy set that implies the policy set of the corresponding formal one.

The rule P-SKIP says that the environment is not changed. The rule for sequential composition says that the final environment of s_1 is used as the starting environment for the next statement s_2. The rules P-WRITE and P-LOCAL-WRITE say that the final environment is that of the right-hand side. Writing to a field requires write access, while writing to a local variable is always allowed. An incremental assignment $w : +e$ requires *incr* access, and the final environment is as for the assignment $w := w + e$. The premises for asynchronous call is as for P-CALL, and the resulting environment is unchanged (since no variable is changed).

Note that, if by mistake, no policy is specified due to forgetfulness, the static compliance checking would detect any use of sensitive information and the program would not pass the privacy checks. In particular, data types with constructors associating data to subjects will be considered sensitive. A non-sensitive method would not be able to access or create sensitive data, and a non-sensitive type declaration would not allow assignment of sensitive information to variables of that type.

We next show how to apply the static analysis on the Prescription case study.

7.5 Case study

Consider the example from Figure 8.5 where *Doctor*, *Nurse*, *Patient*, *PatientData*, *AddPresc*, *GetPresc* are interfaces. A *PatientData* object contains data for a number of patients, and can be accessed by doctors and nurses, based on different policies. Policies are declared by the keyword **`policy`**. Patient data *pd* of type $PData$ (list of *Presc*) is labeled with polices: $\{\mathcal{P}_{GetPresc}, \mathcal{P}_{Doc}\}$, and an implicit policy $(Subject, all, \mathit{self} \sqcap \mathit{read})$ is included in every policy set to allow read access when the principal is the data subject. This policy ($\mathcal{P}_{Presc}$) allows *(i)* a patient to access his/her own data, *(ii)* gives *full* (i.e., read, incr, write) access to the *Doctor* for *treatm* purposes, and *(iii)* gives *read-only* access to the *Nurse*

(P-skip)

$$\frac{}{C, m \vdash [\Gamma]\ skip\ [\Gamma]}$$

(P-composition)

$$\frac{C, m \vdash [\Gamma]\ s_1\ [\Gamma_1] \quad C, m \vdash [\Gamma_1]\ s_2\ [\Gamma_2]}{C, m \vdash [\Gamma]\ s_1; s_2\ [\Gamma_2]}$$

(P-write)

$$\frac{\begin{array}{c} C, m \vdash [\Gamma]\ rhs :: \mathcal{P} \\ write \sqsubseteq \Gamma_C[w] \sqcap (\mathcal{P}_{C,m}@(C, m)) \end{array}}{C, m \vdash [\Gamma]\ w := rhs\ [\Gamma[w \mapsto \mathcal{P}]]}$$

(P-local-write)

$$\frac{C, m \vdash [\Gamma]\ rhs :: \mathcal{P}}{C, m \vdash [\Gamma]\ x := rhs\ [\Gamma[x \mapsto \mathcal{P}]]}$$

(P-incr)

$$\frac{\begin{array}{c} C, m \vdash [\Gamma]\ rhs :: \mathcal{P} \\ incr \sqsubseteq \Gamma_C[w] \sqcap (\mathcal{P}_{C,m}@(C, m)) \end{array}}{C, m \vdash [\Gamma]\ w :+ rhs\ [\Gamma[w \mapsto \Gamma[w] \sqcap \mathcal{P}]]}$$

(P-asyncCall)

$$\frac{C, m \vdash [\Gamma] \quad e.n_I(\overline{e}) :: \mathcal{P}_{out(I,n)}}{C, m \vdash [\Gamma] \quad e!n_I(\overline{e})\ [\Gamma]}$$

Figure 7.6: Policy Rules for Statements.

for *treatm* purposes. The purpose *treatm* is declared by the keyword **`purpose`**. The policies need to be declared only once and then the effect system will keep track of the policies in a given program state. For example, the declaration of $makePresc()$ includes the policy $\mathcal{P}_{AddPresc}$. Now we show an application of a few type rules, on the statements in the method $doctorTask()$ from Figure 8.5.

1. $x := rhs$
   $String\ text = rhs$ //Apply P-LocalWrite
   The premise $rhs :: \bullet$ associates $\bullet$ with rhs, since it is a local variable and has no policy.

   $\Gamma[x \mapsto \mathcal{P}] \implies \Gamma[text \mapsto \bullet]$
   Gamma for *text* is updated with $\bullet$.

2. $Presc\ newp = (p, text);$ //Apply P-Func, P-LocalWrite

a) $read \sqsubseteq \Gamma[v] \sqcap (\mathcal{P}_{C,m}@(C,m))$
   $(\Gamma[p] \sqcap \Gamma[text]) \sqcap \mathcal{P}_{Presc}$
   $(\bullet \sqcap \bullet) \sqcap (\mathcal{P}_{C,m}@(C,m))$
   i.e., $(\bullet \sqcap \bullet) \sqcap \mathcal{P}_{Doc}$ since $(\mathcal{P}_{C,m}@(C,m)) = \mathcal{P}_{Doc}$
   which reduces to $\mathcal{P}_{Doc}$
   $read \sqsubseteq \mathcal{P}_{Doc}$ (*i.e.*, $read \sqsubseteq \Gamma[v] \sqcap (\mathcal{P}_{C,m}@(C,m)))$
   which reduces to $read \sqsubseteq full$, using the notation $A \sqsubseteq (I, R, A')$ when $A \sqsubseteq A'$, and $A \sqsubseteq \{(I, R, A')_i\}$ when $A \sqsubseteq (I, R, A')_i$ for some i (i.e., $A \sqsubseteq A'_i$).

b) $write \sqsubseteq \mathcal{P}_T \sqcap (\mathcal{P}_{C,m}@(C,m))$, since the constructor $(_,_)$ is sensitive
   $write \sqsubseteq \mathcal{P}_{Doc} \sqcap \mathcal{P}_{Presc}$
   which reduces to $write \sqsubseteq full$, and the policy of $(p, text)$ is $\mathcal{P}_{Presc}$

c) $\Gamma[newp \mapsto \Gamma[(p, text)]]$
   $\Gamma[newp \mapsto \mathcal{P}_{Presc}]$ //since pc is empty here

In the first statement, the policy set on *text* is $\{\bullet\}$ because it is not yet associated with a subject. But when non-sensitive *text* is combined with a subject identity, this is seen as construction of a sensitive data, and P-FUNC is used to ensure that the information can be read and constructed by the current context. The rest of the example can be checked in a similar way.

7.6 Related work

Language-based mechanisms are techniques based on programming languages that are often used in developing secure applications. In particular, language-based security mechanisms are used in specification and enforcement of security policies. In recent years, various techniques (compilers, automated program analysis, type checking, program rewriting etc.) have been explored from the perspective of their applicability in enforcing security and privacy policies in programs. Privacy by Design (PbD) has been discussed and promoted from several viewpoints such as privacy engineering [30, 49, 86], privacy design patterns [26, 59], and formal approaches [71, 100, 112]. Tschantz and Wing, in [112] and Daniel Métayer, in [71] discuss the significance of formal methods for foundational formalizations of privacy related aspects. In [100], Schneider discusses the main ideas of *Privacy by Design* and summarizes key challenges in achieving Privacy by Construction and probable means to handle these challenges. The paper calls for ways to ensure control of *purpose* integrated in programming languages. It is also indicated that in order to ensure that privacy-compliant code is sound and correct, formal methods would be helpful in proving soundness and completeness (with respect to a set of predefined privacy concepts). Privacy design strategies [59] focus on how to take privacy requirements into account from the beginning and make it a software quality attribute. The engineering aspects of privacy by design is addressed, but there is a lack on how to apply them in practice. In

our work, we adhere to several privacy design strategies such as separating and hiding the data, and encapsulation in an object-oriented context.

Hayati and Abadi [53] describe a language-based approach based on information-flow control, to model and verify aspects of privacy policies in the Jif (Java Information Flow) programming language. In this approach data collected for a specific purpose is annotated with Jif principals and then the methods needed for a specific purpose are also annotated with Jif principals. Explicitly declaring purposes for data and methods ensures that the labeled data will be used only by the methods with connected purposes. Purposes are organized in a hierarchy, with sub-purposes. However, this representation of purpose is not sufficient to guarantee that principals will perform actions compliant with the declared purpose. But this can be checked statically in our approach, because the principal is restricted by a purpose-based access control.

Basin et al. [11] propose an approach that relates a purpose with a business process and use formal models of inter-process communication to demonstrate GDPR compliance. Process collection is modeled as data-flow graphs which depict the data collected and the data used by the processes. Then these processes are associated with a data purpose and are used to algorithmically i) generate data purpose statements, ii) detect violation of data minimization, and iii) demonstrate compliance of some more aspects of GDPR. Since in GDPR, end-users should know the necessary purpose of data collection, some works such as [11] propose to audit logs and detect if a computer system supports a purpose. In a continuation of this work [4], Arfelt et al. show how such an audit can be automated by monitoring. Automatic audits and monitoring can be applied to a system like ours as a complementary step to verify how it complies with the GDPR. Besides, our work is more focussed on integrating such legal instruments during the design phase, using formal language semantics. In [1], Adams and Schupp consider black-box objects that communicate through messages. The approach is centered around algorithms that take as input an architecture and a set of privacy constraints, and output an extension of the original architecture that satisfies the privacy constraints. This work is complementary to ours in that it puts restrictions on the run-time message handling. In contrast to our work, the approach does not concern analysis of program code.

In [43], Ferrara and Spoto discuss the role of static analysis for GDPR compliance. The authors suggest combining taint analyses and backward slicing algorithms to generate reports relevant for the various actors (i.e., data protection officers, chief information officers, project managers, and developers) involved at various stages of GDPR compliance. In particular, taint analysis is performed on each program statement and then the data-flow of sensitive information is reconstructed using backward-slicing. These flows are then abstracted into the information needed by the compliance actors. However, they do not formalize nor check privacy policies (as we do).

In the sense of access control mechanisms such as RBAC that controls and restricts system access to authorized users, there are some common features. In addition to the hierarchies of roles and access rights supported by RBAC, our framework introduces hierarchies of purposes to control role access. However, our

work uses static analysis while RBAC uses runtime analysis. Anthonysamy et al. [3] demonstrate a *semantic-mapping* approach to infer function specifications from semantics of natural language. This technique is useful in compliance verification as it aids in identification of program constructs that implements certain policies. The authors implement this technique in a tool, CASTOR, which takes policy statements (in natural language) and source code as input, and outputs a set of semantic mappings between policies and function specifications (function name, associated class, parameters etc.).

7.7 Conclusion

We have investigated challenges and opportunities in approaching privacy from the *by-design* perspective, i.e., embedding privacy design requirements into a language. We have considered a small core language supporting active objects, and extended it to integrate privacy policies. We chose three primary constituents of a privacy policy, i.e., *principal, purpose*, and *access right*. Policies are declared for methods and data types, and together restrict the usage of sensitive data.

We defined a language for formulating these policies, discussed static privacy polices, and formalized a concept of static privacy policies. We have formulated rules for policy compliance, given by an extended effect system. The problem of checking a program's compliance with privacy policies, reduces to efficient type-checking. The analysis is class-wise, which is a benefit in open object-oriented systems, and for scalability. Needless to mention that much work needs to be done, in terms of defining possibly new constructs and abstractions in order to formalize the essential data protection principles. In the future we would like to *i)* extend the policy definition language, to express a wider range of privacy restrictions, *ii)* work out a larger case study, and *iii)* in particular focus on the dynamic policy and consent management.

Acknowledgements. The authors were partially supported by IoT-Sec (NRC) and SCOTT (EU).

Authors' addresses

Shukun Tokas University of Oslo, Postboks 1337 Blindern, 0316 Oslo, Norway, shukunt@uio.no

Olaf Owe University of Oslo, Postboks 1337 Blindern, 0316 Oslo, Norway, olaf@uio.no

Toktam Ramezanifarkhani University of Oslo, Postboks 1337 Blindern, 0316 Oslo, Norway, toktamr@uio.no

Chapter 8

Static checking of GDPR-related privacy compliance for object-oriented distributed systems

Shukun Tokas, Olaf Owe, Toktam Ramezanifarkhani

Submitted to the Journal of Logic and Algebraic Programming, in first round revision

Abstract

The adoption of information technology in foremost sectors of human activity such as banking, healthcare, education, governance etc., increases the amount of data collected and processed to enable these services. With the convenience the technology offers, it also brings increased challenges pertaining to the privacy. In response to these emerging privacy concerns, the European Union has approved the General Data Protection Regulation (GDPR) to strengthen data protection across the European Union. This regulation requires individuals and organizations that process personal data of EU citizens or provide services in EU, to comply with the privacy requirements in the GDPR. However, the privacy policies stating how personal information will be handled to meet regulations as well as organizational objectives, are given in natural language statements. To demonstrate a program's compliance with privacy policies, ideally, a link should be established between policy statements and the program code, with the support of a formalized analysis.

Based on this vision, we formalize a notion of privacy policies and a notion of compliance for the setting of object-oriented distributed systems. For this we provide explicit constructs to specify constituents of privacy policies (i.e., principal, purpose, access right) on personal data. We present a policy specification language and a formalization of compliance, as well as a high-level language for distributed systems extended with support for policies. We define a type and effect system for static checking of compliance of privacy policies and show soundness of this analysis based on an operational semantics. Finally, we prove a progress property.

The authors were partially supported by IoT-Sec (NRC) (https://its-wiki.no/wiki/IoTSec:Home) and SCOTT (EU)(www.scott-project.eu).

III

8.1 Introduction

With the adoption of information technology in almost all areas of our life, the collection and processing of personal data have intensified. This development depends on trustworthy functioning of information and communication technologies to support the individual privacy rights and democratic values of society [30]. To address the challenges of data protection and privacy of the individuals within the European Union (EU) and the European Economic Area (EEA), the European Union Parliament approved the General Data Protection Regulation (GDPR) [41]. The GDPR is said to be *"The single most important change in data privacy regulation in 20 years"* [77].

To promote data protection from the outset of the product/service development, requirements to *data protection by design* and *data protection by default* have been formally embedded in Article 25 of the GDPR. Article 25 requires the controllers to design and develop products with a built-in ability to demonstrate compliance towards the data protection obligations. Note that the terms *privacy by design* [22] and GDPR's *data protection by design* have similar goals, and are often used interchangeably. The principle of *data protection by default* says that privacy is built into the system, i.e., no measures are required by the *data subject* in order to maintain her privacy.

In this paper we follow the data protection by design and data protection by default principles, by integrating necessary safeguards into the processing of personal information, using a language-based approach. Our ambition is to investigate how to formalize fundamental privacy principles and to provide built-in abilities to fulfill data protection obligations under the GDPR. As a step towards this goal, we develop a policy language that provides constructs for specifying privacy requirements on personal data and then present a type and effect system for analyzing a program's compliance with respect to the stated privacy policies.

A privacy policy in this setting is a statement that expresses permitted use of personal information of the declared program entities, such as data types and methods of interfaces and classes. In particular, we define a notion of privacy policies given by sets of triples that put restrictions on what kind of *principals* may access the personal information, for what *purposes*, and what kind of operations and *access* are permitted on this data, i.e., restricting *who*, *why*, and *what*. A policy on declared program entities puts restrictions on how they are used and on actions they perform. We define a notion of policy compliance, and show how compliance can be checked at compile time by an extended type and effect system for an object-oriented, distributed language centered around asynchronous and synchronous method interaction, extended with policy specifications. The static checking is class-wise and is dealing with privacy policies rather than ordinary types, and is performed on classes that are type-correct with respect to ordinary typing. Information without a policy is non-sensitive, and its access is not restricted. The static type-checker ensures that a non-sensitive method may not access sensitive information and that a variable of a non-sensitive type may not be assigned sensitive information. (We

use the term “sensitive” as a synonym of “personal”.)

Certain aspects of the GDPR can be expressed by means of static concepts, whereas some can only be expressed at runtime, such as *subject* or *consent* changes by external users, whereas others are not easily formalized, such as the economic penalty rules. At compile time we let the statically declared policies provide privacy by default, and then give a framework enabling change of consent at runtime. The static policies serve as initial policies for a program, while changes in consent and policies can be handled at runtime, for instance through predefined functionalities. By annotating declared program entities with privacy policies and developing a scheme of policy inheritance, we may limit the number of policy annotations needed. This should make the approach simple and easy to use in practice, as demonstrated by our case study. At runtime these policies, possibly extended with additional information such as *data subject*, can be attached to the data values and objects. However, only limited information about the subject is available at compile time; for instance the static analysis of a class may be aware of local data where the *caller* object is the subject.

The static compliance checking is done by a static type and effect system based on the kind of privacy policies outlined here. Even though static notions may only cover limited parts of the GDPR, static compliance checking has the advantage of ensuring that all programs passing the checks do comply with the static GDPR policies, thereby providing a strong guarantee before the programs are executed. The rules are syntax-directed, following the legal formation of expressions and function applications as well as statements. The requirements to communication constructs such as remote calls are central.

We target distributed, object-oriented and service-oriented systems. We formalize a static notion of policy declarations in this setting. To demonstrate the analysis of static policy compliance for imperative programs, we develop a type and effect system for checking policy compliance for a high-level language supporting the active object programming paradigm [16, 61, 65, 85], based on the actor model [57]. In this programming model, objects are autonomous and execute in parallel, communicating by so-called asynchronous method invocations. Object-local data structures are defined by data types. We assume interface abstraction, i.e., an object can only be accessed though an interface and remote field access is illegal. This allows us to focus on major challenges of modern architectures, without the complications of low-level language constructs related to the shared-variable concurrency model. Remote field access would make the analysis less precise.

In summary, the main idea is to provide language constructs that express privacy policy specifications capturing static aspects of the GDPR specific privacy principles and use these to statically analyze a program’s compliance with the policy specifications. We make the following contributions:

1. Propose a policy specification language for specifying purpose, access and policy requirements.

2. Formalize a notion of policy compliance.

3. Show how the policy language can be used with an underlying object-oriented language.
4. Develop a mechanic type and effect system for analyzing a program's compliance with the specified privacy policies.
5. Develop a runtime system with policy tags. Prove soundness/progress.

Whereas all of these represent novel research, the overall contribution of the paper is to show how to approach the formalization of GDPR specific data protection requirements from a static point of view.

Paper outline. The remainder of the paper is structured as follows. Our research focus and the relevance to the GDPR are stated in the next section. Section 8.3 presents our formalization of the GDPR policies, including a policy specification language and a formalization of policy compliance, and outlines how it applies to the setting of object-oriented distributed systems (OODS). We discuss the usage of policies and include the first part of a case study. For the second part of the case study, we define an executable, imperative, high-level language for active object systems, extended with policy specifications. Section 8.4 introduces this language. Section 8.5 presents the static compliance checking by means of a type and effect system, and demonstrates the analysis on the case study. Section 8.6 briefly discusses an extension to deal with consent and self access to personal data about a data subject. Section 8.7 presents an operational semantics and proves soundness and progress. Section 8.8 discusses related work, and Section 8.9 concludes the paper.

8.2 Relevance to the GDPR and research focus

The GDPR contains 99 articles covering quite diverse aspects of privacy such as data protection principles, accountability, data protection impact assessment, certification, penalties etc. However, we will focus on the intersection of mainly Article 5, Article 15, and Article 25, due to their potential for establishing links with programming language mechanisms and in particular static analysis. Figure 8.1 illustrates this idea and our focus. Clearly, one may express a larger part of the GDPR concepts by runtime entities than by compile time entities. Furthermore, it is clear that static analysis will in general be less precise than runtime analysis and typically over-approximate the privacy restrictions. Thus static analysis may seem like a less fruitful approach; however, static analysis has the advantage that problems caught during static checking can be solved before runtime and thereby gives rise to more reliable software and fewer runtime errors. Therefore it is interesting to investigate compile time aspects of the GDPR and to define a notion of static compliance of these aspects.

Article 5 lists the *data protection principles* related to personal data processing, which includes the following: lawfulness, fairness, and transparency; purpose limitation; data minimization; accuracy; storage limitation; integrity and confidentiality. Compliance with these principles is intrinsic for better data protection.

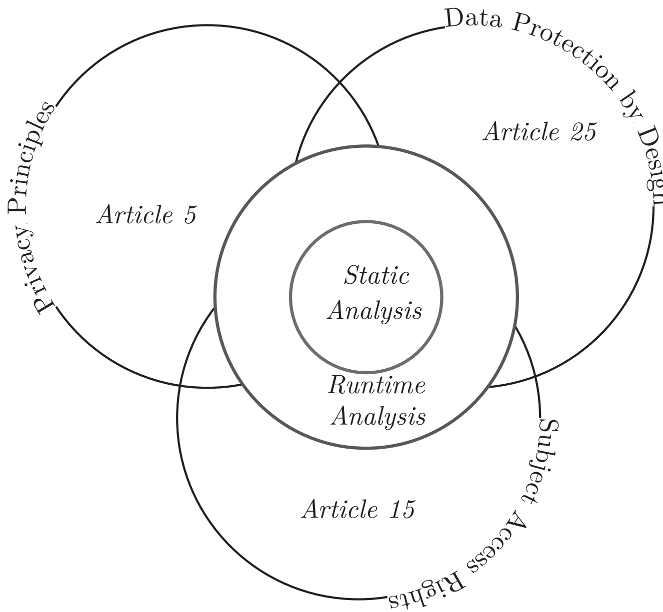

Figure 8.1: Research focus

Article 15 creates a *Right for access by the data subjects* to have access to their personal data that an organization processes and holds about them. The data subject is entitled to obtain, for example, the purposes of data processing; which recipients (such as organizations) is the personal data shared with; how the personal information was collected; existence of right to restrict processing or erasure of personal data. More on subject access rights are discussed in Section 8.3.

Article 25 introduces *data protection by design* and *data protection by default* obligations. It requires the organizations to embed data protection into the design and later stages of product/service development. In addition, it requires that by default, an organization may only process the personal data that is necessary for fulfillment of the stated purposes.

In addition to the articles mentioned above, Article 6 of the GDPR outlines six lawful grounds, such as consent or fulfillment of contractual obligation, for processing of personal data. The regulation treats consent as one of the guiding principles for legitimate processing, and Article 7 sets out the conditions for processing personal data (when relying on consent). We sketched these articles very briefly. For more details, please refer to Articles and Recitals in [41].

In our setting, we specify privacy-by-default policies, which are statically checked. When the lawful basis of processing of personal information is performance of the contract or other valid bases but not the consent, the policies should be formulated in a way that ensures that they are built into the

system *by default*, i.e., no measures are required by the data subject in order to maintain her privacy. However, when consent is the basis of processing, the choices (or privacy settings) of the data subjects are captured at runtime (as outlined in [106]).

To verify formally that a system satisfies its privacy specification, the desired notions of privacy need to be expressed explicitly. However, given these principles and obligations, not all privacy requirements are susceptible to formalization. We study an intersection of these main concepts from the design as well as the legal point of view, with a motivation to establish links between the two views. However, we do not cover all the aspects of the aforementioned articles. For example, the requirements for data minimization, integrity, storage limitation, and accuracy require a different set of tools and methods for assessing compliance.

We illustrate the research focus with an example. In order to provide healthcare services, a clinic collects information related to an individual‘s health. So as to collect and process this information, the clinic first needs to identify the purposes for which this personal information will be used. This is done by statically declaring privacy policy requirements on the methods and data types. These requirements are expressed in a policy specification language, which allows designers to express privacy requirements, contributing towards purpose limitation, transparency, data protection by design, data protection by default, and accountability requirements. In the next section, we discuss the parts of the GDPR that can be formalized, i.e., what can be expressed as policies, and what can be checked. In particular, we focus on static policies and static checking.

8.3 Formalization of static privacy policies and policy compliance

In the object-oriented language setting, an object may assume different views, depending on the interaction context. These views are expressed by specification of the externally observable behaviour of objects, declared through interfaces. We extend this specification of observable behaviour of objects to provide language support for the enforcement of privacy policies.

Clearly, at compile time we are limited to static entities, while at runtime we can deal with runtime entities. Thus, compile time policies must in general be more coarse-grained than runtime policies, for instance the compile-time policy of the value of a variable is based on a worst-case symbolic analysis while at runtime it can be based on the value itself. At compile time, we may express and analyze the GDPR-related notions using static names, either names occurring in the executable program text, or names occurring in specifications capturing GDPR-related aspects. Examples of the former are method names, variable names, type names, class names, and interface names. Examples of the latter are names describing purpose, access rights, and policies. The combination of these two categories of names gives a way of expressing static policies restricting access to the sensitive information. At runtime, it is natural to associate the policies with objects and data values, but these entities are not known at compile

A	::=	$read \mid incr \mid write \mid self$	basic access rights
		$\mid no \mid full \mid rincr \mid wincr$	abbreviated access rights
		$\mid A \sqcap A \mid A \sqcup A$	combined access rights
$\mathcal{P}$	::=	(I, R, A)	policy
$\mathcal{P}s$	::=	$\{\mathcal{P}^*\} \mid \mathcal{P}s \sqcap \mathcal{P}s \mid \mathcal{P}s \sqcup \mathcal{P}s$	policy set
$\mathcal{RD}$	::=	**purpose** R^+	
		[**where** Rel [**and** Rel] *]	purpose declaration
Rel	::=	$R^+ < R^+$	sub-purpose declaration

Figure 8.2: BNF syntax definition of the policy language. I ranges over interface names and R over purpose names.

time. At compile time, policies on data values can be approximated by policies on the corresponding data types. Static policies serve a double purpose: They should have an abstract view meaningful to external users, so that they may understand and reconsider their privacy settings, and at the same time should be meaningful to analysis in terms of program technical concepts at compile time.

8.3.1 Policies

We consider *three* vital constituents of the GDPR privacy policies, namely *principal*, *purpose*, and *access right*, specified by triples (I, R, A) where I, R, and A denote the three constituents, respectively. The main emphasis of the policy specification language is on the specification of privacy restrictions at the language level. It would be appropriate to link an external user's policy view with the system's policy view. For example, a policy $(Doctor, treatm, rincr)$ on a data subject's health information in the system is expressed in natural language to an external user as: A *Doctor* can process your personal health information for *treatment* purposes, but is only allowed to read health records and add new ones without the right to change or delete existing records. Below we give a general description of these policy constituents, as well as how they can be related to the view of external users, and how they will be represented at the programming level.

Principals A principal identifies a single principal or a set of principals, authorized to invoke the method and use the personal data. At the external user level, we let *principals* be described by either an individual or a group of individuals. At the programming level, an individual corresponds to an object representing that individual. A group of individuals is either represented by a set of objects, or by an *interface*, with the understanding that an interface represents the set of objects that *supports* the interface. (We say then an object *supports* an interface if the class of the object implements the interface.) As not all interfaces represent principals, we introduce an interface *Principal*, and require that an interface used to

specify principals must be a subinterface of *Principal*. Thus in compile time policies, principals are described by subinterfaces of *Principal*. The interface *Subject* corresponds to a "data subject", extending *Principal*.

An interface I is used to restrict the access to information, by requiring that the accessing object supports I. For policies reflecting external user settings (for instance policies on data types), I will be a subtype of *Principal*. For policies reflecting declared policies in a program (for instance policies on methods), I need not be a subtype of *Principal*. Interfaces are organized in an open-ended inheritance hierarchy, as in object-oriented program development, letting $I < J$ denote that I is a subinterface of J. For example, $Specialist < Doctor < HealthWorker$. We do not define a bottom element, since the hierarchy is open-ended. We let $\leq$ denote the transitive and reflexive extension of $<$.

Purposes A purpose R *identified by a purpose name, allows us to specify* that personal data must only be collected and processed for the given purpose. At the user level, purposes are described by purpose names. At the programming level, such purpose names are used in policy specifications. For instance, if a method is annotated with a purpose, the annotation specifies that the method may only be called when the caller has (at least) this purpose. Purpose names are defined by the keyword **purpose** and can be organized in a hierarchical structure, representing a *purpose hierarchy* [53]. We allow purpose names to be organized in an open-ended acyclic graph. Examples of purposes are *treatment*, *research*, or *marketing*. We let *all* be a predefined purpose, denoting the least specialized purpose. Consider the declaration

$$\texttt{purpose}\ a, b, c\ \texttt{where}\ a, b < c$$

This declaration makes a and b more specialized purposes than c. For example, $treatm, diagnosis, research < health$, and $monitoring < treatm$. If data is collected for *treatm* purposes, then it can be used for *treatm* as well as purposes subsumed under *treatm* purposes, but not for *research*. If data is collected for say *diagnosis*, then it can neither be used for *treatm* nor for *research*.

We allow purpose names to be organized in an open-ended directed acyclic graph. Consider an example, where *healthcare*, and *shopping*, have *billing* as a subpurpose; and *treatment* could be a subpurpose of both *healthcare* and *billing*. This example indicates that a strict tree-structure could be too limiting. This allows a single purpose name to reflect a specialization of a set of more general purposes. We let $\leq$ denote the transitive and reflexive extension of $<$.

Access rights The access right A restricts the access rights, restricting the kinds of operation that can be performed on the data, such as read access (*read*), incremental access (*incr*), write access (*write*), or a combination

of these. We define a complete lattice of these general access rights (in Figure 8.3) with *no* (no access) and *full* (full access) as the least and greatest access rights, respectively. The *read* access right gives read-only access to the principal, and similarly *write* allows for a write access. Incremental access, *incr*, gives the right to add new information without changing or reading old information. For instance, a lab assistant may be allowed to add test results to a patient's health records, but without reading existing information. The combination of read access and incremental access, $read \sqcup incr$ denoted *rincr*, allows a principal to read the information and to add more information, but not change existing information. This is quite useful in practice, for instance a nurse may be allowed to read and add test results to a patient's health records, but not overwrite or change old information. The combination of $write \sqcup incr$, denoted *wincr*, allows a principal to change and add more information, without reading, for instance she may overwrite, delete and add health records, but without the right to look inside these records. The partial ordering of access rights is denoted $\sqsubseteq_A$. We have that $incr \sqsubseteq_A (read \sqcup write)$ holds, reflecting that the incremental update $x : +e$ can be expressed as $x := x + e$.

Furthermore, $read \sqcap write$, $incr \sqcap write$, and $incr \sqcap read$ give *no* access. The combination of *read* and *write* gives *full* access (including incremental access), i.e., *full* is the same as $read \sqcup write$. This means that we have seven elements in the lattice for basic access rights as seen in Figure 8.3. In subsection 8.3.2, we extend this lattice with access rights for a data subject to access data about herself. At the program level, the specified access rights can be checked for a given program.

A policy is specified by a triple (I, R, A) restricting principals, purpose, and access rights, respectively. Such policy triples can be combined to form policy sets, which are used to represent restrictions due to multiple policies.

Privacy policies and consent are supposed to be decided and changed upon need by the subjects, i.e., the external users of a system that do not in general have insight into to program text. This means that external user defined restrictions on *principal*, *purpose*, and *access right* should be given in terms of a vocabulary or language meaningful to such external users. On the other hand, the external user defined restrictions must connect to concepts at the program level so that compliance can be defined and checked. In our approach, consent is expressed by *the presence of policies.*

In order to limit the amount of policy definitions, we consider user-defined policies for data types and methods, and define an effect system to deduce policy restrictions in each program state on the program variables. Policies on data type declarations give a higher degree of reusability than policies on for instance variable declarations. When personal information is limited to a relatively small number of methods and data types, this means that the privacy policy specifications needed are relatively few. We use a special policy symbol $\bullet$ to denote non-sensitive information. The default policy of a method is $\bullet$ if no policy is specified. Note that a method with no policy will not be able to access

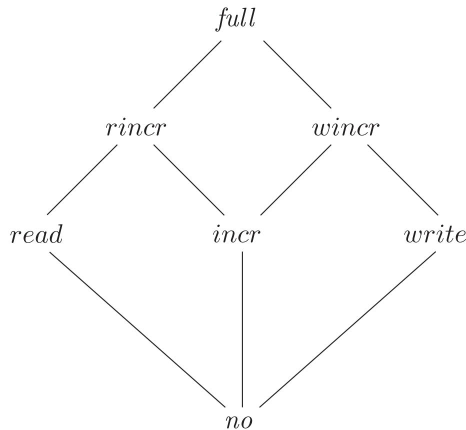

Figure 8.3: The lattice for general access rights (without *self*). Note that *rincr* is the same as $read \sqcup incr$, *wincr* is the same as $write \sqcup incr$, and *full* is the same as $read \sqcup write$.

or use any sensitive information, and variables of types with no policy cannot be assigned sensitive information. This will be checked statically.

For a method m that accesses sensitive information, the associated policy specifies which principals can invoke this method, for what purpose, and an upper bound on permissible access operations. Similarly a data type T with policy $\mathcal{P}_T$ expresses that all values of type T must respect the policy $\mathcal{P}_T$, which can be ensured by policy compliance checks during static analysis.

In the GDPR, *processing* of personal data is defined in terms of any operation or set of operations such as collection, storage, use, dissemination etc. (see Article 4 [41]). We focus on *use* and *disclosure* of personal information. At the programming level, *use* corresponds to the access rights given by the access rights lattice and *disclosure* is expressed by the first restriction on the policy (principal) i.e., a policy set on data describes to whom data is disclosed. Disclosure of information is also captured when information is exchanged through method parameters. However, towards the external users, the terms *use* and *disclose* may be meaningful.

8.3.2 Access rights for data subjects

Under Article 15 [41], the *Right of Access* by the data subject requires the data controller to give the data subject information about the personal data that the controller has about the subject (including the purposes for which this information is used). Based on this requirement, we introduce an interface *Subject* below interface Principal as the superinterface of all classes representing external users. Moreover, we introduce an additional access right, *self*. By means of *self*, one may specify access rights on information about self, i.e., the data

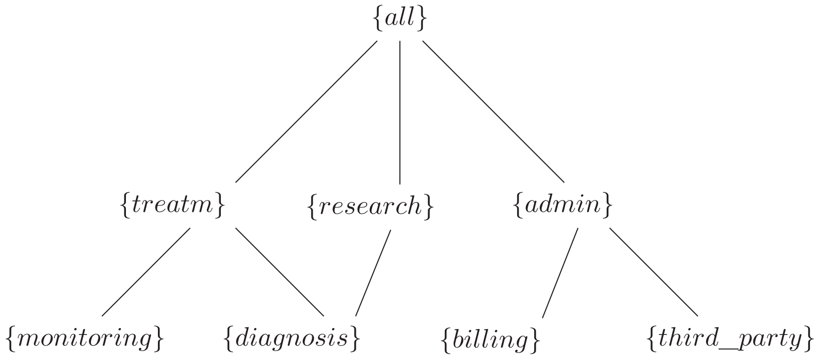

Figure 8.4: Sample purpose hierarchy

subject. One may then express general access rights in combination with access rights on self data. The policy triple

$$(Subject, all, \mathit{self} \sqcap \mathit{read})$$

supports Article 15 (1a) to (1c), by allowing each data subject read access to information about herself. It expresses the principle of giving a subject read access to data about herself. This triple could be added as a default policy for every sensitive data type. Note that the universal purpose *all* is needed to express to express this principle.

Mathematically, our lattice can be defined by a pair lattice as follows:

Definition 12 (Lattice of Access Rights)**.** *Access rights are organized in a lattice with carrier set* $\{(a, b) \mid a \sqsubseteq_A b\}$ *where both* a *and* b *range over the lattice of general access rights (given in Figure 8.3). We define*

$$\begin{aligned} (a, b) \sqsubseteq_A (a', b') &\equiv a \sqsubseteq_A a' \wedge b \sqsubseteq_A b' \\ (a, b) \sqcup (a', b') &\equiv (a \sqcup a', b \sqcup b') \\ (a, b) \sqcap (a', b') &\equiv (a \sqcap a', b \sqcap b') \end{aligned}$$

It follows that the redefined $\sqsubseteq_A$ is a partial order, and the carrier set has 22 elements. The element (a, b) is written $a \sqcup (\mathit{self} \sqcap b)$, and the access right $a \sqcup (\mathit{self} \sqcap a)$ is abbreviated a. The access right $a \sqcup (\mathit{self} \sqcap b)$ expresses that a is the access right on data in general (including self data) and b is the added access right on self data. Thus the access on self data is $a \sqcup b$.

We have that *no* is an identity element of $\sqcup$, and *full* an identity element of $\sqcap$. For instance, $\mathit{no} \sqcup (\mathit{self} \sqcap \mathit{full})$ is the same as *self*, meaning full access to self data, but no access to data about others. Furthermore, $\mathit{self} \sqcap \mathit{rincr}$ gives a principal the right to read self data and add new information about herself. In contrast, $\mathit{read} \sqcup \mathit{self}$ means full access to self data and read access to other data, and $\mathit{read} \sqcup (\mathit{self} \sqcap \mathit{rincr})$ means that a principal may read all data and also increment self data.

The identity of data subjects can be captured at runtime, but not in general at compile time, since these identities are in general not statically known. In order to check access rights about *self*, the static checking will try to detect if the data subject is the same as this or caller. This is discussed in Section 8.6.

8.3.3 Policy compliance

Methods and types are annotated with policies. Annotating these program constructs with policies is a prerequisite for assuring that processing is performed in accordance with the specified policies. The language syntax for policies is summarized in Figure 9.1 and some sample policies are found in Figure 8.5. Optional parts are written in brackets (as in [...]), while superscripts * and $^+$ denote repetition and non-empty repetition, respectively.

Definition 13 (Policy Compliance)**.** *The sub-policy relation less, expressing* policy compliance*, is defined by*

$$(I', R', A') \sqsubseteq (I, R, A) \equiv I \leq I' \wedge R' \leq R \wedge A' \sqsubseteq_A A$$

with $\bullet$ *as* bottom element*, representing non-sensitive information. It follows that* $\sqsubseteq$ *is a partial order. We let Any denote the most general interface, such that* $I \leq Any$ *for each* I*.*

A policy $\mathcal{P}'$ complies with the policy $\mathcal{P}$ if it has the same or larger principal, the same or more specialized purpose, and if the access rights of $\mathcal{P}'$ are the same or weaker than that of $\mathcal{P}$. We let *Any* denote the most general interface, such that $I \leq Any$ for any I.

We say that a method *respects a policy* $\mathcal{P}$ if the policy of the method complies with $\mathcal{P}$. The default policy of a method is $\bullet$ if no policy is specified. Intuitively, $\mathcal{P} \sqsubseteq \mathcal{P}'$ is used to express that the policy of a method implementation/respecification $\mathcal{P}$ complies with that of the method specification $\mathcal{P}'$. In particular, $\bullet \sqsubseteq \mathcal{P}$ expresses that an implementation without access to sensitive information complies with any policy.

Let $\mathcal{P}_{I,m}$ denote the policy of a method m given in an interface I, and $\mathcal{P}_{C,m}$ denote the policy of a method m given in a class C. It is required that the implementation of a method in a class C respects the policy stated in the interface I, i.e., $\mathcal{P}_{C,m} \sqsubseteq \mathcal{P}_{I,m}$. In addition, it is also required that a method redefined in an interface I respects the policy of that method in a superinterface J, i.e., $\mathcal{P}_{I,m} \sqsubseteq \mathcal{P}_{J,m}$. By transitivity of $\sqsubseteq$, a method implementation in a class that respects the policy given in an interface also respects the policy of the method given in a superinterface, i.e., $\mathcal{P}_{C,m} \sqsubseteq \mathcal{P}_{I,m}$ and $\mathcal{P}_{I,m} \sqsubseteq \mathcal{P}_{J,m}$ implies $\mathcal{P}_{C,m} \sqsubseteq \mathcal{P}_{J,m}$.

For instance, consider an interface *GetPresc* with a method *getPresc*() with a policy $(Nurse, treatm, read)$. An implementation of this method in a class must have a policy that complies with it, such as $(Any, treatm, read)$, $(Nurse, treatm, self \sqcap read)$, or $(Nurse, monitoring, read)$. In contrast, the implementation cannot have a policy $(Doctor, treatm, read)$, as this would not

allow a *Nurse* as the caller object, and also not $(Nurse, all, rincr)$, because this violates the purpose and the access restriction.

Moreover, the use of *self* in the access part allows us to distinguish between different kinds of self access for different purposes, for instance $(Patient, all, read \sqcap self)$ and $(Patient, privacy_settings, self)$. The latter gives full access to data about *self* for purposes of *privacy_settings*,[1] while the first gives read access to data about *self* for all purposes.

We define a lattice over sets of policies with meet and join operations, and generalize the definition of compliance to sets of policies:

Definition 14 (Compliance of Policy Sets)**.**

$$\{\mathcal{P}'_i\} \sqsubseteq \{\mathcal{P}_j\} \equiv \forall i \,.\, \exists j \,.\, \mathcal{P}'_i \sqsubseteq \mathcal{P}_j$$

This expresses that a policy set S' complies with a policy set S if each policy in S' complies with some policy in S. When no confusion occurs we simply write $\mathcal{P}$ instead of $\{\mathcal{P}\}$. For instance, $\mathcal{P} \sqsubseteq S$ denotes $\{\mathcal{P}\} \sqsubseteq S$, and $\mathcal{P} \sqcap S$ denotes $\{\mathcal{P}\} \sqcap S$. Furthermore we use the notation $A \sqsubseteq_A (I, R, A')$ when $A \sqsubseteq_A A'$, and use the notation $A \sqsubseteq_A \{(I, R, A')_i\}$ when $A \sqsubseteq_A (I, R, A')_i$ for some I (i.e., $A \sqsubseteq_A A'_i$).

We define meet and join operations over policy sets by set union and a kind of intersection, respectively, adding implicitly derivable policies:

Definition 15 (Join and Meet Over Policy Sets)**.**

$$S \sqcup S' \equiv closure(S \cup S')$$

$$S \sqcap S' \equiv closure(\{P \mid P \sqsubseteq S \wedge P \sqsubseteq S'\})$$

where the closure operation *is defined by*

$$closure(S) \equiv S \cup \{(I, R, A \sqcup A') \mid (I, R, A) \sqsubseteq S \wedge (I, R, A') \sqsubseteq S\}$$

We have a lattice with $\emptyset$ as the bottom element, $S \sqsubseteq S \sqcup S'$, and $S \sqcap S' \sqsubseteq S$. The closure operation adds implicitly derivable policies, and ensures that $\{(I, R, A \sqcup A')\} \sqsubseteq \{(I, R, A)\} \sqcup \{(I, R, A')\}$. For instance, we have that $\{(Doctor, treatm, read)\} \sqcup \{(Doctor, treatm, write)\}$ is the same as $\{(Doctor, treatm, full)\}$. [2]

The meet operation typically reflects worst-case analysis. For an actual parameter (or method result) we need to check that the policy set of the actual parameter allows all policies in the policy set of the corresponding formal parameter. For this we use the following notion:

Definition 16 (Implication of Policy Sets)**.** *We define the notation*

$$\mathcal{P}s' \Longrightarrow \mathcal{P}s$$

($\mathcal{P}s'$ guarantees $\mathcal{P}s$) by $\{\bullet\} \Longrightarrow \mathcal{P}s$ and $\mathcal{P}s \sqsubseteq \mathcal{P}s'$ for $\mathcal{P}s'$ other than $\{\bullet\}$.

[1] private_ settings purpose is not defined. Does it reflect consent update?

[2] CUT? This also means that join and meet operations on policy sets will not result in conflicting information since we have a lattice.

In particular, $\{\bullet\} \Longrightarrow \mathcal{P}s$, expresses that a non-sensitive actual parameter always is acceptable. If $\mathcal{P}s' \Longrightarrow \mathcal{P}s$ we also say that the former guarantees the latter. (Note that $\mathcal{P}s'$ and $\mathcal{P}s$ denote policy sets.)

8.3.4 Policies in an object-oriented setting

Here, we proceed to discuss how to use policies in combination with interfaces, methods, and types. An imperative programming language for defining classes is given in Section 8.4 by means of an imperative-style language for active objects. An example with policies and interfaces is given in Figure 8.5.

Definition 17 (Interface Syntax)**.** *An* interface *is declared with the BNF syntax*

$$\texttt{interface}\ I\ [\texttt{extends}\ J^{+}]\ \{D^{*}\}$$

where I and J range over interface names, and D denotes a method declaration (without body), with its own (optional) policy.

Here a new interface I is declared, extending a number of superinterfaces. A method redefined in I must have a policy that complies with that of the method in a superinterface J. Methods may be inherited (keeping the superinterface method policy) or redefined in I. For simplicity, we assume that a redefined version of the same method has the same parameter and return types as in the superinterface. (Alternatively we could use a version of co/contra-variance.)

We consider next single class inheritance given by an **implements** clause. A class C extending a superclass inherits all declarations of the superclass, apart from redefined methods and the **implements** clause (and class constructors are concatenated). We may allow a subclass to implement different interfaces than its superclass, and we may allow a redefined method to have a different policy than that of the superclass. In particular C does not need to support the interfaces of its superclass. The motivation for this is to achieve better flexibility. This requires that typing of object variables is done by interfaces, following the semantics of [89]. Thus, the policies of redefined methods need not comply with those in the superclass, as long as they comply with the policies of the interfaces implemented by C. We let the policies of inherited method be inherited as well, and must then comply with the requirements of the enclosing class (C).

Definition 18 (Method Declaration Syntax)**.**

$$T\ m([Y\ y]^{*})\ [::\mathcal{P}]$$

where T is the return type and Y is the type of parameter y.

An inherited method m inherits the policy of m in the superinterface, unless the interface declares its own policy for m. As mentioned, the redefined policy of m in an interface cannot be more restrictive than that of the superinterface (J), i.e., $\mathcal{P}_{I,m} \sqsubseteq \mathcal{P}_{J,m}$, ensuring that a class implementation of m satisfying $\mathcal{P}_{I,m}$ also satisfies any declarations of m in a superinterface.

Definition 19 (Type Declaration Syntax)**.**

$$\textbf{type}\ N\ [TypeParameters] =<\ type_definition >\ \ [:: \mathcal{P}s]$$

where the type parameters are optional. We let the policy of a type N, denoted $\mathcal{P}_N$, be a policy set. Types declared without a policy are non-sensitive.

The predefined basic types (*Nat*, *Int*, *String*, *Bool*, *Void*) are non-sensitive. Furthermore, object variables (references) are non-sensitive since a reference in itself does not carry any sensitive information. A user-defined type is sensitive if a policy set is specified, or if the definition contains a sensitive data type constructor (as explained below). If there for instance is a need for strings with sensitive information, restricted by a policy $\mathcal{P}s$, one would define a type for this by

$$\textbf{type}\ Info\ = String :: \mathcal{P}s$$

A list type $List[T]$ is sensitive if T is sensitive, and has the policy of T, i.e., $\mathcal{P}_{List[T]} \equiv \mathcal{P}_T$. The same principle applies to other container types, such as sets and multisets. When T' is declared as a subtype of T, we require that the policy of the subtype guarantees that of the supertype, i.e.,

$$T' \leq T \Rightarrow (\mathcal{P}_{T'} \Longrightarrow \mathcal{P}_T)$$

A sensitive data type can often be defined as a pair of (possibly non-sensitive) data, say

$$\textbf{type}\ Presc\ = Patient * String$$
$$:: \{(Doctor, treatm, full)), (Nurse, treatm, read)\}$$

In this case the *Presc* constructor function (i.e., the pair operator) is considered sensitive, since it associates data to a subject. An application of this constructor may create new sensitive information about a patient, and therefore we require that the enclosing method has a policy with write access to *treatm* data (such as the policy $\mathcal{P}_{Doc}$). In general, an application of a sensitive data type constructor requires write access, as will be formalized in the policy type rules for expressions (Section 8.5).

We consider next *sensitive* functions, which create new sensitive data, for instance a product of individually non-sensitive data may be sensitive. Generator functions (here called constructors) are considered *sensitive* if they i) combine information about a subject with non-sensitive or sensitive information or ii) use sensitive information. We assume that sensitive generators produce sensitive types (with some exceptions, such as constructors of encrypted data). Defined functions are *sensitive* if their type is sensitive and the definition directly or indirectly contains a sensitive application of a constructor. For instance we may (recursively) define a parameterized list type by $List[T] = empty() \mid append(List[T] * T)$ meaning that lists have the form $empty()$ or $append(l, x)$, where l is a list and x a value of type T. We let the notation $l + x$ abbreviate $append(l, x)$. The list is sensitive if T is sensitive, in which case the

append constructor function is also sensitive. The *Presc* type is sensitive (even though String is not), and the pair $(current_patient, "no\ health\ problems")$ is a sensitive application of the product constructor. These examples suffice for our purposes here. It can be detected statically if a function is sensitive (further details are omitted). Some predefined type constructors including encryption functions could be defined as non-sensitive.

Applications of sensitive functions may create new sensitive data, something which require write access. In this way the policy control of variables is driven by the declared types rather than variable declarations. The advantage is that policy specifications on the defined types are reusable in the same way that the defined types are reusable, while policy specifications on variable declarations would not in general be reusable. Furthermore, reusable policy specifications developed over time are likely to be more reliable than one-time adhoc specifications for program variables.

Example The example in Figure 8.5 shows a data type *Presc* with policy set $\{(Doctor, treatm, full), (Doctor, treatm, rincr), (Nurse, treatm, read)\}$. The policy $(Doctor, treatm, rincr)$ is redundant since $(Doctor, treatm, rincr) \sqsubseteq (Doctor, treatm, full)$, and is colored grey to indicate that. Method *makePresc* has policy $(Doctor, treatm, rincr)$, meaning that this method must be called by a Doctor object (or a more specialized object), for purposes of treatment and with read and incremental access (but not full write access). Thus a doctor can add new prescription but not change or remove old ones. Method *getPresc* has policy $(Nurse, treatm, read)$, meaning that this method must be called by a Nurse object (or a more specialized object such as a Doctor object), for purposes of treatment and with read-only access. These two methods, with associated policies, are inherited in interface *PatientData*. The method *getMyPresc* offered by the Nurse interface has policy $(Patient, treatm, self)$, meaning that this method must be called by a Patient object for treatment purposes and access is limited to data about the caller patient but not other patients. Alternatively, the policy could be $(Patient, treatm, self \sqcap read)$ if a patient is not allowed to change her treatment records.

8.3.5 Compliance checking of OODS languages

Consider an OODS language extended with policy specifications as above. Thus, methods that may access personal information are annotated with single policies, and data types that may involve personal information are annotated with policy sets reflecting the permitted usage by different principals. We assume pure expressions. In this setting, static checking of compliance consists of checking that interface extensions and implementations by classes respect the method policy specifications, and that method calls and all program variable accesses are done according to the relevant policies. Since the policy sets of the values of program variables may change from state to state, we use an effect system to keep track of the policy sets in a given program state. The rules use an environment Γ, which is a mapping from program variable names to policy sets,

such that the policy set of a variable in a given state gives an upper bound of the permitted operations. The environment is also used to determine the policy set of an expression. For each statement in a considered language there is one or more rules explaining how the environment Γ is modified by the statement. This is normally reflected in the conclusions of the rules. The premises of the rules incorporate policy checks, and in general this can be explained as follows:

- For a subinterface it is checked that the policy of a method complies with ($\sqsubseteq$) that of the same method in the superinterface.
- For a class it is checked that
  - the declared policy of each method complies with ($\sqsubseteq$) that of the same method in any interface implemented by the class (if any).
  - the actual policy of the implementation of a method complies with ($\sqsubseteq$) the declared policy in the class of that method.
- For a method call, it is checked that
  - the policy of the called method complies with ($\sqsubseteq$) the policy of the calling context (as given by the enclosing method body)
  - the policy set of each actual parameter guarantees ($\Longrightarrow$) that of the corresponding formal parameter.
- For a new statement, a similar check is done on the actual parameters.
- For read/write/incr access to a program variable, it is checked that
  - there is read/write/incr access in the policy set of the variable and also in the policy of the calling context (given by the enclosing method).
  - However, we may assume write access to local variables. This is harmless since they cannot be used to store information after termination of the enclosing method.
- For a return statement, it is checked that
  - the policy set of the returned value guarantees ($\Longrightarrow$) that of the method return type.
  - the policy set of each field according to Γ guarantees ($\Longrightarrow$) the declared policy of that field according to the policy on the type.
- For each application of a constructor function giving rise to sensitive information, it is checked that the enclosing method has write access.

Justification

The specification of policies could be a burden on the programmer. Reuse of policies is advantageous, and it would be desirable to keep the amount of policy specifications at a minimum. Therefore we let policies be specified for data types and methods only, and not for individual variables. Moreover, we imagine that only a limited amount of data types/methods deal with sensitive information. Thus it is advantageous to limit the policy specification to those. The policy inheritance of policies on methods increases policy reuse. We believe a single policy is appropriate for a method, and this means that data access for other purposes than the one in the method policy is not allowed. For instance, a method body with “treatm” as the method policy purpose cannot make calls to methods with “marketing” in the method policy.

If by mistake a data type/method is lacking a policy, the static detection would not be successful, since sensitive constructors are detected statically and require the corresponding data type to be sensitive. This implies that a subset of the data types and methods must be sensitive and have a non-empty policy in order for the static checking to be OK. However, there could be data types that should be sensitive but without a specified policy and without constructors detected as sensitive, for instance text with embedded personal information. This is left as the programmer’s responsibility.

Another issue could be that a programmer specifies full access for all purposes on all methods, for instance intending to shortcut the static checking of data manipulations in the body, and thereby hoping to allow everything. However, this will not work well since the principal part I of the policy of a method would need to be less than the principal parts of the policies on all the data types involved. In practice, that would mean that I must be less than a large number of interfaces, which is not possible in an open-ended hierarchy without a bottom element.

In our formalization a method has a single policy because each method should be made with a certain user group (principal) in mind. We have seen here that this decision has the benefit of making it harder to bypass the policy checks (regardless of whether it is intended or unintended). These considerations make it harder to get away with "wrong" policy specifications, but do not take away the programmers’ responsibility of making appropriate policy specifications. The static checking is based on the given specifications and will complain when there is something wrong with the policies.

In the next section, we will consider a small imperative language for active object systems and then define a type and effect system along the lines explained above. Figure 8.7 defines classes corresponding to the interfaces in Figure 8.5, using the imperative language.

8.4 An imperative programming language

In order to give a high-level view of distributed systems, we choose a small language based on the active object paradigm supporting high-level interaction

```
purpose monitoring, treatm, health
        where monitoring < treatm < health

policy P_Doc = (Doctor, treatm, full)
policy P_DocTask = (Any, treatm, full)
policy P_AddPresc = (Doctor, treatm, rincr)
policy P_GetPresc = (Nurse, treatm, read)
policy P_GetSelfPresc = (Nurse, health, read)
policy P_PatientPresc = (Patient, treatm, read)
policy P_Start = (Any, treatm, no)

policy P_Presc = {P_GetPresc, P_AddPresc, P_Doc}

type Presc == Patient * String :: P_Presc

interface Patient extends Subject {Void getSelfData() :: P_Start}
interface AddPresc {Void makePresc(Presc newp):: P_AddPresc}
interface GetPresc {Presc getPresc(Patient p) :: P_GetPresc}
interface PatientData extends AddPresc, GetPresc {}
interface Nurse extends Principal {
    Presc nurseTask() :: P_GetPresc
  with Patient
    Presc getMyPresc() :: P_PatientPresc }
interface Doctor extends Nurse{
  Void doctorTask(Patient p) :: P_DocTask }
```

Figure 8.5: Interface, type and policy definitions for the prescription example. Grey policy specifications are implicit while underlined ones need to be explicitly stated.

mechanisms [16]. The active object paradigm is based on concurrent autnomous objects and offers both synchronous and asynchronous communication, while avoiding shared variables and avoiding low level synchronization mechanisms such as explicit signaling and notification. This setting allows a simple, compositional semantics, as in [61, 93], which is beneficial to analysis. All code is organized in methods definitions inside classes, something which is helpful for static policy declarations and for class-wise static policy checking.

The language is imperative and strongly typed, using data types for defining data structure locally inside a class. The data type sublanguage is side-effect-free. A type system (for checking type-correctness w.r.t. ordinary types) can be made as in [62]. We formalize the analysis outlined in Section 8.3.5 for this language by incorporating policy specifications as defined in the previous section for method declarations and data types. An effect system calculates the policy set of a variable in a given program state and checks all variable accesses as well as

Pr	$::=$	$[\mathcal{T} \mid \mathcal{RD} \mid In \mid Cl]^*$	program
$\mathcal{T}$	$::=$	**type** $N\,[\overline{T}]$ =<type_expression> $[::\mathcal{P}s]$	type definition
T	$::=$	Int $\mid$ Any $\mid$ Bool $\mid$ String $\mid$ Void $\mid$ List[T] $\mid I \mid N$	interfaces and types
In	$::=$	**interface** I [**extends** I^+] $\{D^*\}$	interface declaration
Cl	$::=$	**class** C $([T\ z]^*)$	class definition
		[**implements** I^+] [**extends** C]	support, inheritance
		$\{[T\ w\ [= ini]]^*$	fields
		$[B\ [::\mathcal{P}]]$	class constructor
		[[**with** I] $M]^*\}$	methods
D	$::=$	$T\ m([T\ y]^*)\ [::\mathcal{P}]$	method signature
M	$::=$	$T\ m([T\ y]^*)\ [B]\ [::\mathcal{P}]$	method definition
B	$::=$	$\{[T\ x\ [= rhs];]^*\ [s]$ [; **return** rhs]$\}$	method blocks
v	$::=$	$w \mid x$	assignable variable
e	$::=$	$v \mid y \mid z \mid$ this $\mid$ caller $\mid$ void $\mid f(\overline{e}) \mid (\overline{e})$	pure expressions
ini	$::=$	$e \mid$ **new** $C(\overline{e})$	initial value of field
rhs	$::=$	$ini \mid e.m(\overline{e})$	right-hand sides
s	$::=$	skip $\mid s; s$	sequencing
		$\mid v := rhs \mid v :+ e \mid e!m(\overline{e}) \mid I!m(\overline{e})$	assignment and call
		$\mid$ **if** e **then** s [**else** s] **fi**	if statement
		$\mid$ **while** e **do** s **od**	while statement

Figure 8.6: BNF syntax of the core language, extended with policy specification. A field is denoted w, a local variable x, a method parameter y, a class parameter z, type names N, and list append is denoted $+$. The brackets in $[T]$ and $[\overline{T}]$ are ground symbols. Function symbols f range over pre-/user-defined functions/constructors with prefix/mixfix notation.

policy restrictions on called methods and generated sensitive data. We assume type-correct programs with respect to ordinary types.

The BNF syntax of the language is summarized in Figure 9.3. The notation $\overline{e}$ denotes a list of expressions e. As before, optional parts are written in brackets (except for type parameters, as in *List*$[T]$, where the brackets are ground symbols). The superscripts * and $^+$ denote repetition and non-empty repetition, respectively. The *cointerface* of a method is given by a **with** clause, and gives restrictions on the *callee* object: only objects supporting the cointerface may call methods in the interface. Thus for a call $o.m(\ldots)$, the caller (available through the caller variable) will be typed by the cointerface. For instance in the class implementation of *Nurse* in Figure 8.7, the *with* clause is needed for method *getMyPresc* in order to make the call to *getMyPresc* type correct since here it is required that caller is of interface *Patient*.

Class and method parameters, the implicit class parameter this, and the implicit method parameter caller are read-only. A class may implement a number of interfaces, and for each method of an interface of the class, it is required

```
class PATIENTDATA() implements PatientData {
  type PData = List[Presc] :: 𝒫_Presc
  PData pd = empty();

  Presc getPresc(Patient p){return last(pd/p)} :: 𝒫_GetPresc
  Void makePresc(Presc newp) {
     if newp ≠emptyString() then pd:+ newp fi } :: 𝒫_AddPresc
  }

class NURSE(PatientData pdb) implements Nurse{
  Presc nurseTask(Patient p){ return pdb.getPresc(p)} :: 𝒫_GetPresc
  with Patient
    Presc getMyPresc() {return pdb.getPresc(caller)} :: 𝒫_PatientPresc
  }

class DOCTOR() extends NURSE //inherits class parameter pdb
                    implements Doctor{
  Void doctorTask(Patient p){
    Presc oldp = pdb.getPresc(p);
    String text = ...; //new presc using symptoms info and oldp
    Presc newp = (p, text);
    pdb!makePresc(newp)}:: 𝒫_DocTask
  }

class PATIENT(String id, Doctor d, Nurse n) implements Patient{
  Void getSelfData(){ n!getMyPresc() } :: 𝒫_Start
  }

class MAIN(){
  PatientData pdbase = new PATIENTDATA();
  Nurse n = new NURSE(pdbase);
  Doctor d = new DOCTOR(pdbase);
  Patient p = new PATIENT("P001",d,n);

  { d!doctorTask(p); p!getSelfData() } :: 𝒫_Start // class constructor
  }
```

Figure 8.7: Doctor and Nurse classes accessing patient prescriptions

that the class defines the method such that the cointerface and types of each method parameter and return value are respected. Additional methods may be defined in a class as well, but these may not be called from outside the class. The language supports single class inheritance and multiple interface inheritance (using the keyword **extends**).

We assume that all inherited or implemented versions of a method m declared in an interface have the same input and output types. A method body $\overline{T\ x = e}; s$ with initialization of the local variables can be understood as $\overline{T\ x}; \overline{x := e}; s$ without initialization of the local variables. We assume type-correct programs, and when needed include type information in the programs subjected to static analysis: In the static analysis, we write e_T for an expression of type T, where T results from the underlying type checking. We write $o.m_I(\overline{e})$ when I is the interface of o as resulting from the underlying type checking.

As mentioned, we let $\leq$ denote the subtype relation. For instance, $Nat \leq Int$, and for a subinterface I' of I, we have $I' \leq I$. We also write $C \leq I$ if class C implements interface I, or a subinterface of I. The only variable typed by a class is this (allowing calls of form this!$m(...)$ where m is a method of the class, including privates ones).

The language could be extended in various ways, for instance with non-blocking forms of two-way method interaction. Local futures are supported by the runtime system and may trivially be included in the language. This would allow a future generated by one method to be picked up by another method executed by the same object. Furthermore, the language may be extended with cooperative scheduling (supporting suspended remote calls) as in [61]. This would be orthogonal to the treatment of privacy policies.

8.4.1 Data types and sensitive data types

A data type is defined by a type expression, possibly recursive. For our purposes we consider type expressions composed of disjoint unions and products, using names to distinguish the different cases (variants) of a disjoint union. These variants are called *constructor functions* since they define the values of the data type. For instance we may define a parameterized list type by $\mathit{List}[T] = \mathit{empty}() + \mathit{append}(\mathit{List}[T] * T)$, meaning that lists have the form $\mathit{empty}()$ or $\mathit{append}(l, x)$, where l is a list and x a value of type T. A pair product type can be defined by $\mathit{PatientInfo} = (\mathit{Patient} * \mathit{Nat})$ where *Patient* is a subinterface of *Subject*. Then the pair (p, d) is of type *PatientInfo* for p of interface *Patient* and d of type Nat. (Here "(_,_)" is the constructor.) Functions over a data type can then be defined by case expressions over the different variants of the type, or simply by a set of equations representing the different cases. Consider the type $\mathit{List}[\mathit{PatientInfo}]$. We may define a projection operator $(\mathit{proj} : \mathit{List}[\mathit{PatientInfo}] * \mathit{Patient} \rightarrow \mathit{List}[\mathit{PatientInfo}])$ by $\mathit{proj}(\mathit{empty}(), p) = \mathit{empty}()$ and $\mathit{proj}(\mathit{append}(l, (p, d)), p') =$ **if** $p = p'$ **then** $\mathit{append}(\mathit{proj}(l, p'), (p, d))$ **else** $\mathit{proj}(l, p')$. Using the infix notion / for *proj* and + for list append, this gives the definition $\mathit{empty}()/v = \mathit{empty}()$ and $(l + (v1, v2))/v =$ **if** $v = v1$ **then** $(l/v) + (v1, v2)$ **else** l/v. The projection

operator is extracting those pairs which have a given first element. The *last* function on lists of *PatientInfo* is defined such that $last(append(l, x)) = x$.

A data type is considered *sensitive* if its definition contains a variant with sensitive information or a product where one component is of interface *Subject*, or a subinterface of *Subject*, because a value of this type could be used to encode personal information about a data subject. (One could consider ways to override this, in cases where no personal information may occur.) Similarly, a constructor is considered *sensitive* if it contains a sensitive component or a component of interface *Subject* (or a subinterface). For instance the pair (p, d) is sensitive when p is of interface *Patient*. Moreover $empty()$ is not sensitive while $append(l, (p, d))$ is. A defined function is considered *sensitive* if the function type is sensitive and the definition contains a sensitive subexpression. (Again the language could have ways to overrule this when required, for instance in order to accomodate encryption functions.) An application of a constructor of a sensitive type and with an argument which is either sensitve or of interface Subject, may create new sensitive information. This requires write access (as checked by the type rules for expressions).

8.4.2 An example

An example program is given in Figure 8.7, showing class implementations of the interfaces given in Figure 8.5 as well as a main class. The *getPresc* call is a blocking call while the other calls are asynchronous. Note that the tuple $(p, text)$ in method *doctorTask* is sensitive and requires write access by the enclosing method, which is satisfied by the policy $\mathcal{P}_{DocTask} = (Any, treatm, full)$. If the policy had been $\mathcal{P}_{Doc}$, the call to *doctorTask* from the main program would fail the policy checking. The expression $last(pd/p)$ is sensitive since it gives a sensitive type, with policy $\mathcal{P}_{Presc}$. The enclosing method has policy $\mathcal{P}_{GetPresc}$, which is sufficient for read access of this kind of information since $\mathcal{P}_{GetPresc} \sqsubseteq \mathcal{P}_{Presc}$. The details of the policy checking for the example is shown in Section 8.5.1.

8.5 An effect system for privacy

In general, a static type or effect system consists of a set of rules that establish safety properties that hold in all states of an execution [84]. As we are interested in privacy policies, our rules ensure that a well-typed program enforces the specified privacy policies correctly. (This is done after ordinary type checking.) The rules use an environment Γ expressing statically derivable information about program variables in a given state, in our case privacy policies. As explained in Section 8.3.5, Γ is a mapping from program variable names to policy sets, such that the policy set of a variable in a given state gives an upper bound on the permitted usage. The environment may change from state to state, and therefore the rules will modify the environment, which means that we have an effect system. We use the notation $\Gamma[v]$ for map look-up and $\Gamma[v \mapsto \mathcal{P}]$ for

$$\frac{\mathcal{P}_{I,m} \sqsubseteq \mathcal{P}_{J,m} \quad \text{for each } J \in \overline{J} \text{ such that } m \in J}{\Gamma \vdash \ \textbf{interface}\ I\ \textbf{extends}\ \overline{J}\{\overline{D}\}\ \ \textbf{ok}}\ \text{(P-INTERFACE)}$$

$$\frac{\begin{array}{c}\mathcal{P}_{C,m} \sqsubseteq \mathcal{P}_{I,m} \quad \text{for each } I \in \overline{I} \text{ such that } m \in I \\ C \vdash\ M\ \textbf{ok} \quad \text{for each } M \in \overline{M}\end{array}}{\vdash\ \textbf{class}\ C(\overline{Z}\ \overline{z})\ \textbf{implements}\ \overline{I}\ \{\overline{W}\ \overline{w};\ \overline{M}\}\ \ \textbf{ok}}\ \text{(P-CLASS)}$$

$$\text{defining } \Gamma_C = [\overline{z} \mapsto \mathcal{P}_{\overline{Z}}, \overline{w} \mapsto \mathcal{P}_{\overline{W}}, \mathsf{this} \mapsto \{\bullet\}, pc \mapsto \{\bullet\}]$$

$$\frac{\begin{array}{c}C, m \vdash [\Gamma_C[\overline{y} \mapsto \mathcal{P}_{\overline{Y}}, \overline{x} \mapsto \{\bullet\}, \mathsf{caller} \mapsto \{\bullet\}]]\ s\ [\Gamma] \\ C, m \vdash [\Gamma]\ rhs :: \mathcal{P}' \quad \mathcal{P}' \Longrightarrow \mathcal{P}_T \\ \Gamma[w] \Longrightarrow \Gamma_C[w] \quad \text{for each field } w\end{array}}{C \vdash\ T\ m(\overline{Y}\ \overline{y})\{\overline{X}\ \overline{x}; s; \textbf{return}\ rhs\} :: \mathcal{P}\ \textbf{ok}}\ \text{(P-METHOD)}$$

Figure 8.8: Policy rules for classes and methods. (Note: read-only access for $\overline{z}$ and $\overline{y}$.)

extending Γ with a new binding (replacing any old binding for v). The policy set of a variable v in the context of Γ is simply given by $\Gamma[v]$.

Type enforcement requires that the policies are respected when the variables are accessed. This gives more fine-grained control letting the policies change with the program point. To reflect changes related to branching constructs we use an addition program variable *pc* (the "program counter") as common in type systems for security aspects [96]. For instance, in the branches of an if statement with a sensitive test, *pc* is adjusted by the policy set of the test. The statement $l := h$ where l and h are boolean variables, is semantically equivalent to **if** h **then** $l := \mathit{true}$ **else** $l := \mathit{false}$ **fi**, so both should result in a sensitive value for l when h has a sensitive value. The presence of *pc* makes this possible since the level of l is adjusted by the level of the test (recorded in *pc*) in the branches.

We give an effect system for ensuring privacy policy compliance, formalized by five kinds of judgments: For a statement s, the judgment

$$C, m \vdash [\Gamma]\ s\ [\Gamma']$$

expresses that inside a method body m and an enclosing class C, the statement(list) s when started in a state satisfying the environment Γ results in a state satisfying the environment Γ'. The rules are right-constructive in the sense that Γ' can be constructed from Γ and s. For an expression or right-hand side e, the judgment

$$C, m \vdash [\Gamma]\ e :: \mathcal{P}$$

(P-VAR)

$$\frac{read \sqsubseteq_A \Gamma[v] \sqcap (\mathcal{P}_{C,m}@(C,m))}{C, m \vdash [\Gamma]\ v ::\ \Gamma[v] \sqcap \Gamma[pc]}$$

(P-CONSTANT)

$$\frac{}{C, m \vdash [\Gamma]\ const() :: \Gamma[pc]}$$

(P-FUNC)

$$\frac{\begin{array}{ll} C, m \vdash [\Gamma]\ e_i ::\ \mathcal{P}_i & \text{for each argument } e_i \text{ of a sensitive type} \\ write \sqsubseteq_A \mathcal{P}_T \sqcap (\mathcal{P}_{C,m}@(C,m)) & \text{if } f_T \text{ is a sensitive constructor} \end{array}}{C, m \vdash [\Gamma]\ f_T(\overline{e}) ::\ \mathcal{P}_T \sqcap \Gamma[pc]}$$

(P-CALL)

$$\frac{\begin{array}{c} C \not\leq I \qquad \mathcal{P}_{I,n} \sqsubseteq_{Co,R}\ \mathcal{P}_{C,m}@(C,m) \\ C, m \vdash [\Gamma]\ e :: \mathcal{P}' \\ C, m \vdash [\Gamma]\ e_i :: \mathcal{P}_i \quad \mathcal{P}_i \Longrightarrow \mathcal{P}_{par(I,n)_i} \quad \text{for each } i \end{array}}{C, m \vdash [\Gamma] \quad e.n_I(\overline{e}) :: \mathcal{P}_{out(I,n)} \sqcap \Gamma[pc]}$$

(P-LOCALCALL)

$$\frac{\begin{array}{c} C \leq I \qquad \mathcal{P}_{I,n} \sqsubseteq\ \mathcal{P}_{C,m}@(C,m) \\ C, m \vdash [\Gamma]\ e :: \mathcal{P}' \\ C, m \vdash [\Gamma]\ e_i :: \mathcal{P}_i \quad \mathcal{P}_i \Longrightarrow \mathcal{P}_{par(I,n)_i} \quad \text{for each } i \end{array}}{C, m \vdash [\Gamma] \quad e.n_I(\overline{e}) :: \mathcal{P}_{out(I,n)} \sqcap \Gamma[pc]}$$

(P-NEW)

$$\frac{C, m \vdash [\Gamma]\ e_i :: \mathcal{P}_i \quad \mathcal{P}_i \Longrightarrow \Gamma_{C'}[z_i]}{C, m \vdash [\Gamma]\ \textbf{new}\ C'(\overline{e}) :: \Gamma[pc]}$$

Figure 8.9: Policy rules for expressions and right-hand sides.

expresses that the expression e when evaluated in a state satisfying Γ gives a value satisfying policy $\mathcal{P}$, where m is the enclosing method and C the enclosing class. For a method definition M in a class C, the judgment

$$C \vdash\ M\ \textbf{ok}$$

expresses that a method complies with its privacy policy. Similarly for a class definition Cl, the judgment

$$\vdash\ Cl\ \textbf{ok}$$

expresses that the method definitions comply with the behaviour described by the interfaces and that the method definitions in the class are OK. For an interface definition In, the judgment

$$\vdash\ In\ \textbf{ok}$$

expresses that any re-defined policy of the method in In must comply with that of the superinterface.

(P-SKIP)

$$\frac{}{C, m \vdash [\Gamma]\ skip\ [\Gamma]}$$

(P-COMPOSITION)

$$\frac{C, m \vdash [\Gamma]\ s_1\ [\Gamma_1] \quad C, m \vdash [\Gamma_1]\ s_2\ [\Gamma_2]}{C, m \vdash [\Gamma]\ s_1; s_2\ [\Gamma_2]}$$

(P-WRITE)

$$\frac{\begin{array}{c} C, m \vdash [\Gamma]\ rhs :: \mathcal{P} \\ write \sqsubseteq_A \Gamma_C[w] \sqcap (\mathcal{P}_{C,m}@(C, m)) \end{array}}{C, m \vdash [\Gamma]\ w := rhs\ [\Gamma[w \mapsto \mathcal{P}]]}$$

(P-LOCAL-WRITE)

$$\frac{C, m \vdash [\Gamma]\ rhs :: \mathcal{P}}{C, m \vdash [\Gamma]\ x := rhs\ [\Gamma[x \mapsto \mathcal{P}]]}$$

(P-INCR)

$$\frac{\begin{array}{c} C, m \vdash [\Gamma]\ rhs :: \mathcal{P} \\ incr \sqsubseteq_A \Gamma_C[w] \sqcap (\mathcal{P}_{C,m}@(C, m)) \end{array}}{C, m \vdash [\Gamma]\ w :+ rhs\ [\Gamma[w \mapsto \Gamma[w] \sqcap \mathcal{P}]]}$$

(P-ASYNCCALL)

$$\frac{C, m \vdash [\Gamma] \quad e.n_I(\overline{e}) :: \mathcal{P}}{C, m \vdash [\Gamma] \quad e!n_I(\overline{e})\ [\Gamma]}$$

(P-BROADCAST)

$$\frac{\begin{array}{c} \mathcal{P}_{I,n} \sqsubseteq_{Co,R}\ \mathcal{P}_{C,m}@(C, m) \\ C, m \vdash [\Gamma]\ e_i :: \mathcal{P}_i \quad \mathcal{P}_i \Longrightarrow \mathcal{P}_{par(I,n)_i} \quad \text{for each } i \end{array}}{C, m \vdash [\Gamma] \quad I!n(\overline{e})\ [\Gamma]}$$

(P-IF)

$$\frac{\begin{array}{c} C, m \vdash [\Gamma]\ e :: \mathcal{P} \\ C, m \vdash [\Gamma[pc \mapsto (\Gamma[pc] \sqcap \mathcal{P})]]\ s_1\ [\Gamma_1] \\ C, m \vdash [\Gamma[pc \mapsto (\Gamma[pc] \sqcap \mathcal{P})]]\ s_2\ [\Gamma_2] \end{array}}{C, m \vdash [\Gamma]\ \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{fi}\ [(\Gamma_1 \sqcap \Gamma_2)[pc \mapsto \Gamma[pc]]]}$$

(P-WHILE)

$$\frac{\begin{array}{ll} \multicolumn{2}{c}{C, m \vdash [\Gamma_i]\ e :: \mathcal{P}_i} \\ C, m \vdash [\Gamma_i[pc \mapsto (\Gamma_i[pc] \sqcap \mathcal{P}_i)]]\ s\ [\Gamma'_i] & i = 1 \dots n \\ \Gamma_{i+1} = \Gamma_i \sqcap \Gamma'_i & i = 1 \dots n \end{array}}{C, m \vdash [\Gamma_1]\ \textbf{while}\ e\ \textbf{do}\ s\ \textbf{od}\ [\Gamma_n[pc \mapsto \Gamma_1[pc]]]}$$

Figure 8.10: Policy rules for statements. In the last rule n is the least i such that $\Gamma_{i+1} = \Gamma_i$.

The typing rules for interfaces, classes, and methods are given in Figure 8.8, Figure 8.9 defines the typing rules for expressions and right-hand sides, and Figure 8.10 defines the typing rules for statements. We let $\mathcal{P}_{I,m}$ denote the policy of method m of interface I, $\mathcal{P}_{C,m}$ denote the policy of method m of class C, and $\mathcal{P}_T$ denote the policy associated with a type T. If no policy is specified for any declaration, we understand that there is no sensitive information, i.e., the policy is $\{\bullet\}$. Note that, if by mistake, no policy is specified on a method due to forgetfulness, the static compliance checking would detect any use of sensitive information and the method body would not pass the privacy checks. In particular data types with constructor functions associating data to subjects will be considered sensitive. A non-sensitive method would not be able to access or create sensitive data, and a non-sensitive type declaration would not allow assignment of sensitive information to variables of that type.

The rule P-INTERFACE checks that a redeclared method m in an interface I respects the policy of m in a superinterface J. The premise ensures that the policy declaration of m in I complies with the policy of m in J, i.e., $\mathcal{P}_{I,m} \sqsubseteq \mathcal{P}_{J,m}$. (The premise is redundant when the policy of m is inherited from J.)

In Rules P-CLASS and P-METHOD, W is the type of field w, Z is the type of class parameter z, X is the type of local variable x, and Y is the type of formal parameter y. Rule P-CLASS checks that a class definition is OK, requiring that the policy of each exported method complies with the policy of the method in the corresponding interface, and that each method definition respects its policy. A class constructor (if any) is treated like a method, with the name *init* (with an implicit *return void()* at the end). We therefore need not show the case of the class constructor explicitly.

Rule P-METHOD checks that a method definition respects the declared policy $\mathcal{P}$, requiring that the method body relates the starting enviroment to the resulting environment Γ, and that the policy on the return value evaluated in Γ must comply with the policy of the return type. The starting environment of a method is the environment of the class, denoted by Γ_C, defined by the declared policies of the types of the class parameters and fields, updated with the policies of the types of the formal parameters, and those of the initial values of the local variables. The latter are all $\{\bullet\}$, and so are the policies of this and caller, since they are object references. Rule P-METHOD also ensures that policies on the fields at method end according to Γ guarantee the policies on the types. (The guarantee operator, $\Longrightarrow$, is defined in Definition 5). The presence of a *with* clause gives no change in the premises, since the cointerface defines the interface of the caller, which has the default policy $\{\bullet\}$. Notice that the policies of declared types, methods, as well as Γ_C, are constant, while the policies of $\Gamma[v]$ change with the program point.

To check variable accesses and calls made in a method body, we define the policy of *method body*. This will allow the caller to act as a principal inside the method body (with the purpose and access right of the method), something which is needed when the current object does not in itself reflect a principal (i.e., when C does not implement a principal).

Definition 20 (Method Body Policy Set).
The policy set of the body of a method m *in class* C *is defined by*

$$(I, R, A)@(C, m) \equiv \begin{cases} \{(I, R, A)\} & \text{if } I \leq Principal \\ \cup_i \{(I_i, R, A)\} & \text{otherwise} \end{cases}$$

where (I, R, A) *is the policy of the method and where* I_i *ranges over all the interfaces of* C *that export* m.

For example, $\mathcal{P}_{DocTask}@(DOCTOR, doctorTask)$ will give $\{(Doctor, treatm, full)\}$. This allows the body of *doctorTask* to call *getPresc* since it can act as a *Doctor* (and since $\mathcal{P}_{DocTask}$ is the policy on method *doctorTask*). As another example, $\mathcal{P}_{AddPresc}@(PATIENTDATA, makePresc)$ will search for an interface which exports *makePresc*, which is the interface *AddPresc*. This means that the method body policy set of *makePresc* is $\{(Doctor, treatm, rincr)\}$, which suffices for the incremental update of *pd*.

The rules in Figure 8.9 define the policies resulting from expressions and right-hand sides: The Rule P-VAR says that the policy set of a variable v is the policy of v according to the environment ($\Gamma[v]$) and the policy set of the program counter *pc* according to Γ. The premise ensures that there is read access to v according to the policy set of the variable and according to the policy of the enclosing method body. Note that $read \sqsubseteq_A \{\bullet\}$, and the same holds for *write*, *incr*, and *self* as well.

Constant constructors represent non-sensitive information since they are not composed by sensitive information. The policy set of a constant is therefore given by the policy of *pc* in the current environment Γ, as stated in Rule P-CONST. This rule also applies to predefined constants such as *void*().

The Rule P-FUNC considers a function application $f_T(\bar{e})$ where T is the resulting type. The policy set of the function application is the meet of the policy set of T and the policy of *pc* in the environment Γ. The first premise ensures that each sensitive argument is OK. This implies that there is read access to each variable v, occurring in a sensitive argument. In case the function f is a sensitive constructor, it is required that there is write access according to the policy set of T and the policy set of the enclosing method body (premise 2). As constant constructor functions are considered non-sensitive and have no arguments, Rule P-CONST can be seen as a special case of Rule P-FUNC.

In addition to controlling the information extracted from an object, one also needs to control the information flowing into an object. This is checked by ensuring that the actual parameters respect the policies of the types of the formal parameters. This is checked as part of the P-CALL rule, and similarly, the actual class parameters are checked in the P-NEW rule. The Rule P-CALL ensures that the current object has sufficient access to call method n through interface I, that the arguments and callee expressions are OK. We use the notation $par(I, n)$ to denote these types, and $out(I,n)$ to denote the return type. The operation $\sqsubseteq_{Co,R}$ is a simplified policy compliance check, which only compares the I and R parts of the policies, i.e., $(I', R', A') \sqsubseteq_{Co,R} (I, R, A) \equiv I \leq I' \wedge R' \leq R$. When a method n is called through an interface I, we check that the purpose of the method body complies with that of n and that the calling object supports the

cointerface of n. The call itself then gets the policy given by the return type, as defined in the method n of interface I, and this is adjusted by the policy of pc.

Local calls are similar to remote calls, but as they may update the fields of the current object, it must be checked that the access rights of the enclosing method is respected by the called method. Therefore Rule P-LocalCall is like Rule P-Call, but the first premise is stronger than the case of remote calls, considering also the access right part. The first premise of checks that the interface of the called method is either the current class C (in which case the call is local) or is implemented by C (in which case the call is local if o is this). This overestimates the set of possible local calls in a sound manner (since the condition $o =$ this is beyond static control).

The Rule P-New ensures that the arguments are OK, and that the policy sets of each argument respects the policy of the type of the corresponding formal class parameter (as defined in class C' using *init* as the name of the class constructor). The value resulting from the object creation is a reference to the new object, and therefore has no sensitive information ($\{\bullet\}$). The value resulting from the object creation is then adjusted with the policy of pc.

The effect rules of Figure 8.10 explain the handling of statements. The rule P-Skip says that a *skip* statement does not change the environment. The rule P-Composition for sequential composition indicates that the environment resulting from one statement can be used as the starting environment for the following statement.

The Rule P-write considers the case that the left-hand side variable is a field w and checks that there is write access to this field, both with respect to the policy of the type of w and the policy of the enclosing method body. This check is done in the second premise. The first premise ensures that the right-hand side is OK and results in a policy set $\mathcal{P}$. This policy set is then used as the policy associated with w in the environment resulting from the assignment statement. Thus assigning non-sensitive values to fields is allowed if the enclosing method and the type have a policy with write access. The rule P-LocalWrite is similar except that we need not check write access (since full access is allowed for local variables). For simplicity, formal class and method parameters (as well as this and caller) are read-only in our language, and this is enforced by the BNF syntax of assignments because it's only allowed to write to the fields and local variables.

The rule P-incr for incremental assignment to a field w is similar to Rule P-write except that here *incr* access is required. The resulting policy for w is the meet of the policy on the former value and the policy on the right-hand side since the new value is $w + rhs$. Incremental assignment to a local variable, say $x :+ rhs$, is semantically the same as $x := x + rhs$ since there is full access to local variables, and we omit a rule for this.

For Rules P-write, P-local-write, and P-incr, the policy on *rhs* also captures the change in sensitive context due to *if* and *while* tests using the policy on *pc*, due to the rules for expressions. This ensures that the policy on *rhs* complies with that of the program counter context, i.e., *pc*.

The rule P-asyncCall for an asynchronous call is similar to Rule P-call, except that the return type is ignored (since no information is returned). The rule for

broadcast calls P-BROADCAST is similar, but without a check on the callee. The call is broadcast to all objects supporting interface I.

The rule P-IF is straight forward, apart from two considerations: In case the if-test is sensitive, the *pc* of the starting environment of each branch must be adjusted by the policy of the expression in the if-test. This is done by a meet operation on $\Gamma[pc]$, i.e., $\Gamma[pc \mapsto (\Gamma[pc] \sqcap \mathcal{P})]$. Secondly, the policy resulting from an if-statement is the *meet* of the policies at the end of each branch, corresponding to a worst case analysis, with the policy of *pc* in the final environment reset to its value before the if-statement. The rule P-WHILE is somewhat similar to P-IF, but the resulting policy is the least fix-point of the iterated effect on the starting policy, reflecting that the number of iterations is unknown at compile time. The fix-point will exist since the lattice hierarchy is finite, and since Γ_{i+1} is less than $\sqsubseteq \Gamma_i$ since $\Gamma_{i+1} = \Gamma_i \sqcap \Gamma'_i$. After the while statement *pc* is reset to its value before the while-statement.

8.5.1 Static compliance checking of the example

In Figure 8.7, which is a continuation of the example in Figure 8.5, we consider some classes implementing the interfaces, including a main class that is automatically instantiated when running the program.

Note that all policies on the visible methods (those exported by an interface) are inherited from the respective interfaces, and need not be repeated by the programmer. They are therefore marked as gray. Also the policy on the sensitive data type *PData* follows from that on *Presc* since the policy of *List*[T] is the policy of T. Only the local class constructor of *MAIN* needs an explicitly specified policy. The classes demonstrate most of the language features including blocking calls, asynchronous calls, and broadcasts, as well as write access, incremental access, and read access. And they demonstrate privacy policy specifications. A challenge here is that the construct $(p, text)$ requires write access since it constructs sensitive data. As discussed later this is acceptable in class *DOCTOR* since type *Presc* gives *full* treatment access to *Doctor* objects and class *DOCTOR* has interface *Doctor*. This expression would not be allowed in class *Nurse*.

We show below the static analysis of the program in Figure 8.7. The premises are handled one by one. The outline below demonstrates that the program satisfies the static analysis.

1. Rule P-ASYNCCALL. Consider the following snippet.

```
class MAIN(){ ...
{d!doctorTask(p)} :: P_Start }
```

   Here, $\underline{e!n_I(\overline{e})}$ is $d!doctorTask(p)$. The premises are shown below:

   1.1. $[\Gamma] \quad e.n_I(\overline{e}) :: \mathcal{P}$

1.1.1. $C \not\leq I$
$MAIN \not\leq Doctor$

$\mathcal{P}_{I,n} \sqsubseteq_{Co,R} \mathcal{P}_{C,m}@(C, m)$
$\mathcal{P}_{DocTask,doctorTask} \sqsubseteq_{Co,R} \mathcal{P}_{MAIN,init} \Leftrightarrow$
$(Any, treatm, full) \sqsubseteq_{Co,R} (Any, treatm, no)$

1.1.2. $[\Gamma]\ e :: \mathcal{P}'$
$[\Gamma]\ d :: \{\bullet\}$ $\quad //object\ references\ are\ not\ sensitive$

1.1.3. $[\Gamma]\ e_i :: \mathcal{P}_i$
$[\Gamma]\ p ::= \{\bullet\}$

$\mathcal{P}_i \Longrightarrow \mathcal{P}_{par(I,n)_i}$
$\{\bullet\} \Longrightarrow \mathcal{P}_{par(Doctor,doctorTask)}$
$\{\bullet\} \Longrightarrow \mathcal{P}_p$
$\{\bullet\} \Longrightarrow \{\bullet\}$ // trivially true

1.1.4. $[\Gamma] \quad e.n_I(\overline{e}) :: \mathcal{P}_{out(I,n)} \sqcap \Gamma[pc]$
$[\Gamma] \quad e.n_I(\overline{e}) :: \{\bullet\} \sqcap \{\bullet\}$
$[\Gamma] \quad e.n_I(\overline{e}) :: \{\bullet\}$

1.2. $[\Gamma] d!doctorTask(p)\ [\Gamma]$

2. Rules P-CALL, P-LOCAL-WRITE

```
class DOCTOR() extends NURSE implements Doctor{
  Void doctorTask(Patient p){
    Presc oldp = pdb.getPresc(p); ...}:: P_DocTask
  }
```

Here, $\underline{x := rhs} \implies Presc\ oldp = pdb.getPresc(p)$

2.1. $[\Gamma]\ rhs :: \mathcal{P}$ $\quad$ // P-LOCAL-WRITE *premise*
rhs is $pdb.getPresc(p)$

2.1.1. $C \not\leq I$
$DOCTOR \not\leq GetPresc$
$\mathcal{P}_{I,n} \sqsubseteq_{Co,R}\ \mathcal{P}_{C,m}@(C, m)$
$\mathcal{P}_{GetPresc,getPresc} \sqsubseteq_{Co,R} \mathcal{P}_{DocTask}@(DOCTOR, doctorTask) \Leftrightarrow$
$\mathcal{P}_{GetPresc,getPresc} \sqsubseteq_{Co,R} (Any, treatm, full)@(DOCTOR, doctorTask)$
$\Leftrightarrow$
$(Nurse,treatm,read) \sqsubseteq_{Co,R} (Doctor,treatm,full) \Leftrightarrow$
(Def: *Method Body Policy Set*)
$(Nurse, treatm, read) \sqsubseteq_{Co,R} (Doctor, treatm, full)$

(Def: *Policy Compliance*)

Here, Interface *Doctor* inherits $getPresc()$ from the *Nurse* interface, i.e., $Doctor \leq Nurse$, and policy of the inherited method complies with the policy in current context making this call valid.

2.1.2. $[\Gamma]\ e :: \mathcal{P}'$
$[\Gamma]\ pbd :: \{\bullet\}\ //object\ references\ are\ not\ sensitive$

2.1.3. $[\Gamma]\ e_i :: \mathcal{P}_i$
$[\Gamma]\ p ::= \{\bullet\}$

$\mathcal{P}_i \Longrightarrow \mathcal{P}_{par(I,n)_i}$
$\{\bullet\} \Longrightarrow \mathcal{P}_{par(GetPresc,getPresc)}$
$\{\bullet\} \Longrightarrow \mathcal{P}_p$
$\{\bullet\} \Longrightarrow \{\bullet\}$

2.1.4. $pdb.getPresc(p) :: \mathcal{P}_{out(I,n)} \sqcap \Gamma[pc]$
$pdb.getPresc(p) :: \mathcal{P}_{Presc} \sqcap \{\bullet\}$ // since pc is non-sensitive
i.e., $pdb.getPresc(p) :: \mathcal{P}_{Presc}$

2.2. $\Gamma[x \mapsto \mathcal{P}] \implies \Gamma[oldp \mapsto \mathcal{P}_{Presc}]$

3. Rules P-Func, P-Var, P-Local-Write, P-asyncCall

```
class DOCTOR() extends NURSE implements Doctor{
  Void doctorTask(Patient p){...
    String text = ...; //new presc using symptoms and oldp
    Presc newp = (p, text);
    pdb!makePresc(newp)}:: P_DocTask
}
```

3.1. $x := rhs$
$String\ text = rhs$ //P-LocalWrite
$rhs :: \{\bullet\}$

$\Gamma[x \mapsto \mathcal{P}] \implies \Gamma[text \mapsto \{\bullet\}]$

3.2. $Presc\ newp = (p, text);$ //P-Func, P-Var, P-LocalWrite

3.2.1. $[\Gamma]\ e_i ::\ \mathcal{P}_i$

3.2.1.1. $read \sqsubseteq_A \Gamma[v] \sqcap (\mathcal{P}_{C,m} @ (C, m))$ // P-Var
$\Gamma[p] \sqcap (\mathcal{P}_{DocTask}\ @\ (DOCTOR, doctorTask))$
$\{\bullet\} \sqcap ((Any, treatm, full) \cup (Doctor, treatm, full))$
$\{\bullet\} \sqcap (Doctor, treatm, full)$
i.e., $(Doctor, treatm, full)$.

$read \sqsubseteq_A (Doctor, treatm, full)$, which reduces to
$read \sqsubseteq_A full$

Likewise, for $text$ as it is also non-sensitive and same method body context applies.

3.2.1.2. $p :: \Gamma[p] \sqcap \Gamma[pc] \Leftrightarrow$
$p :: \{\bullet\}$, since pc is non-sensitive here.

These premises ensures that the variables in the constructor function has read access as well as that the current context complies with read access.

3.2.2. $write \sqsubseteq_A \mathcal{P}_T \sqcap (\mathcal{P}_{C,m}@(C,m))$, since the constructor $(_,_)$ is sensitive
$f_T(p, text)$ and T is $\mathcal{P}_{Presc}$
$write \sqsubseteq_A \mathcal{P}_T \sqcap (\mathcal{P}_{C,m}@(C,m))$
$write \sqsubseteq_A \mathcal{P}_{Presc} \sqcap (\mathcal{P}_{DocTask}@(DOCTOR, doctorTask))$
$write \sqsubseteq_A \{(Nurse, treatm, read), (Doctor, treatm, full)\} \sqcap (Doctor, treatm, full)$
$write \sqsubseteq_A (Doctor, treatm, full)$, which reduces to $write \sqsubseteq_A full$.

This premise checks if the sensitive information $(p, text)$ can be constructed in the current context, and here it can be constructed because the current context has $write$ access.

3.2.3. $[\Gamma]\ (p, text) ::\ \mathcal{P}_T \sqcap \Gamma[pc]$
$[\Gamma]\ (p, text) ::\ \mathcal{P}_{Presc}$, since pc is non-sensitive here.

3.3. $e!n_I(\overline{e}) = pdb!makePresc(newp)$

3.3.1. $C \not\leq I$
$Doctor \not\leq AddPresc$

$\mathcal{P}_{I,n} \sqsubseteq_{Co,R} \mathcal{P}_{C,m}@(C,m)$
$\mathcal{P}_{AddPresc,makePresc} \sqsubseteq \mathcal{P}_{DocTask}@(DOCTOR, doctorTask) \Leftrightarrow$
$(Doctor, treatm, rincr) \sqsubseteq_{Co,R} (Doctor, treatm, full) \Leftrightarrow$
$(Doctor, treatm, rincr) \sqsubseteq_{Co,R} (Doctor, treatm, full)$

3.3.2. $[\Gamma]\ e :: \mathcal{P}'$
$[\Gamma]\ pdb :: \{\bullet\}$

3.3.3. $[\Gamma]\ e_i :: \mathcal{P}_i$
$[\Gamma]\ newp :: \mathcal{P}_{Presc}$ // P-VAR

$\mathcal{P}_i \Longrightarrow \mathcal{P}_{par(I,n)_i}$
$\mathcal{P}_{newp} \Longrightarrow \mathcal{P}_{par(AddPresc,makePresc)}$

$\mathcal{P}_{Presc} \Longrightarrow \mathcal{P}_{Presc}$

4. Rules P-IF, P-INCR

```
class PATIENTDATA() implements PatientData { ...
  Void makePresc(Presc newp) {
     if newp ≠emptyString() then pd:+ newp fi } :: P_AddPresc
  }
```

if e **then** s_1 **else** s_2 **fi**

4.1. $e :: \mathcal{P}$ // P-VAR
$newp \neq emptyString() :: \mathcal{P}$

$read \sqsubseteq \Gamma[v] \sqcap (\mathcal{P}_{C,m}@(C, m))$ // P-VAR
$read \sqsubseteq_A \Gamma[newp] \sqcap (\mathcal{P}_{AddPresc}\ @\ (PATIENTDATA, makePresc))$
$read \sqsubseteq_A \mathcal{P}_{Presc} \sqcap \mathcal{P}_{AddPresc}$, since PatientData is not a principal.
$read \sqsubseteq_A \{(Nurse, treatm, read), (Doctor, treatm, full)\} \sqcap (Doctor, treatm, rincr)$
$read \sqsubseteq_A (Doctor, treatm, rincr)$

$newp :: [\Gamma[newp] \sqcap \Gamma[pc]]$
$newp :: [\mathcal{P}_{Presc} \sqcap \mathcal{P}_{Presc}]$
$newp :: \mathcal{P}_{Presc}$

$emptyString() :: \mathcal{P}$ // P-CONSTANT
$emptyString() :: \Gamma[pc]$
$emptyString() :: \{\bullet\}$, since pc is non-sensitive here.

$newp \neq emptyString() :: \mathcal{P}_{Presc}$

4.2. $[\Gamma[pc \mapsto (\Gamma[pc] \sqcap \mathcal{P})]]\ s_1\ [\Gamma_1]$
$\Gamma[pc \mapsto (\mathcal{P}_{Presc} \sqcap \mathcal{P}_{Presc})]\ pd : +newp\ [\Gamma_1]$
$\Gamma[pc \mapsto \mathcal{P}_{Presc}]\ pd : +newp\ [\Gamma_1]$

Now, rule P-INCR, on s_1

4.2.1. $rhs :: \mathcal{P}$
$newp :: \mathcal{P}_{Presc}$ // since $\Gamma[pc \mapsto \mathcal{P}_{Presc}]$

4.2.2. $incr \sqsubseteq_A \Gamma_C[w] \sqcap (\mathcal{P}_{C,m}@(C, m))$
$incr \sqsubseteq_A \Gamma_C[pd] \sqcap (\mathcal{P}_{AddPresc}@(PATIENTDATA, makePresc))$
$incr \sqsubseteq_A \mathcal{P}_{Presc} \sqcap (\mathcal{P}_{Doctor,treatm,rincr})$
$incr \sqsubseteq_A \mathcal{P}_{AddPresc}$
$incr \sqsubseteq_A (Doctor, treatm, rincr)$

which reduces to $incr \sqsubseteq_A rincr$

4.2.3. $[\Gamma[w \mapsto (\Gamma[w] \sqcap \Gamma[pc])]$
$[\Gamma[pd \mapsto (\Gamma[pd] \sqcap \Gamma[pc])]]$
$[\Gamma[pd \mapsto (\mathcal{P}_{Presc} \sqcap \mathcal{P}_{Presc})]]$
$[\Gamma[pd \mapsto \mathcal{P}_{Presc}]]$

4.3. $[\Gamma]$ **if** e **then** s_1 **else** s_2 **fi** $[\Gamma_1 \sqcap \Gamma_2[pc \mapsto \Gamma[pc]]]$
$\Gamma_1[pc \mapsto \Gamma[pc]]$
$\Gamma_1[pc \mapsto \mathcal{P}_{AddPresc}]$

Interface *PatientData* extends interfaces *GetPresc* and *AddPresc*, but does not redefine the policies on inherited methods. So the policies on inherited methods trivially complies with that of the superinterfaces. Thus interface *PatientData* is well-formed. Class *PATIENTDATA* is well-formed because

1. $\mathcal{P}_{PATIENTDATA,getPresc} \sqsubseteq \mathcal{P}_{GetPresc,getPresc}$ and $\mathcal{P}_{PATIENTDATA,makePresc} \sqsubseteq \mathcal{P}_{AddPresc,makePresc}$, i.e., the policies on the method definitions comply with those of the method declarations in the interfaces.

2. For method *getPresc*, the policy on the return value complies with the policy of the return type, i.e., $\mathcal{P}_{Presc} \Longrightarrow \mathcal{P}_{Presc}$.

   Moreover, for method *makePresc*,

   - Γ_C is defined by $[newp \mapsto \mathcal{P}_{Presc}, caller \mapsto \{\bullet\}]$ and the if-statement is well-formed (as described above in 4).
   - The policy on the field *pd* complies with that on the declared type, i.e., $\mathcal{P}_{Presc} \Longrightarrow \mathcal{P}_{Presc}$.

We may conclude that the static analysis is successful. However, with the current rules we cannot check if a *Patient* accessing her own information, through method *GetMyPresc*, is valid. In particular, we can not check the *self* access. We return to this in Section 8.6.

8.6 Awareness of subject

We discuss here how the above framework could be extended so that (static) awareness of the subject of sensitive information is handled. In particular, we would like the analysis to detect that expressions such as $last(pd/p)$ (with pd as in the example) result in data with p as data subject, and therefore can be communicated/returned to p by the principle of read access to data about self. With the formalism above it is required that the caller supports the *Nurse* interface.

Our framework uses interfaces to describe the visible aspects of the active objects and data types to define data structures, including personal data. We use subtyping to distinguish (potential) personal data from non-personal data.

The data type hierarchy is extended with a subtype *PersonalData*, and all sensitive data types must be of a subtype of *PersonalData*. We introduce the interface *Sensitive* as the superinterface of all classes holding personal information. Interface *Subject* is below *Principal*, and for instance interface *Patient* is below *Subject*. We let *PersonalData* support a function *subjects* returning the set of the subjects of the data, of type *Set*[*Subject*]. Let p be a subject. For a pair (p, d) where d is non-sensitive, we have that $subjects((p, d))$ is $\{p\}$, and for a sensitive constructor f we have that $subjects(f(p, \overline{d}))$ is $\{p\}$ when the list $\overline{d}$ is non-sensitive.

We now specify purpose by terms of the form $name(p)$ where *name* is a purpose name as before and p identifies the subject, either by an object (for instance given by this or caller), an interface name, or a set of object expressions. In a runtime tag, p will be a set of object references, while it may be over-approximated by an interface in the static setting.

In the example we would have that the policy for method *makePresc* could be $(Doctor, treatm(Patient), rincr)$. Furthermore, we could make a policy $(Doctor, treatm(p), rincr)$ where p is a Patient object. This way we may distinguish between the treatment of individual patients. With the added notions, we may extract the subject(s) of sensitive information inside a method. The data structure in the example with patient data *pd* is defined as a list of pairs as before, but now we can express that $subjects(pd/p) = \{p\}$ and $subjects(last(pd)) = subjects(pd)$ for a Patient p.

As mentioned in Section 8.3, we may include the general policy

$$(Subject, all, self \sqcap read)$$

to give each subject read access to personal data about herself. This allows a more liberal policy checking than in the previous section, by allowing the statement **return** e when $subjects(e)$ is caller, and allowing a parameter e in a method call to o when $subjects(e)$ is o.

The main achievement with the renewed example (see Figure 8.11) is that we detect statically that the *getMyPresc* method complies with the static policies, even if patients have no specified access rights on *PATIENTDATA* objects, because this method uses only self access. We will also be able to treat methods such as *getMyPresc* in class *NURSE* and *getSelfData* in class *PATIENT* in Figure 8.7.

In order to deal with dynamic changes in consent, we let interface *Sensitive* contain a method for updating the policies of sensitive data, *upd_consent*, with the new consent settings as a parameter *new_policy*. A class supporting *Sensitive* must then implement this method (preferably implemented directly in the runtime system) by changing the tag on any local data in the object where the caller is the subject (as given in the purpose part). If this is the case, the runtime tag l must be changed to $l \sqcap new_policy$. To initiate a change in consent settings with new policy *np*, a subject may make the broadcast

$$Sensitive!upd_consent(np)$$

```
policy P_Doc = (Doctor, treatm(Patient), full)
policy P_GetPresc = (Nurse, treatm(Patient), read)
policy P_Presc = {P_GetPresc, P_Doc}

class PATIENTDATA() implements PatientData {
  type PData = List[Presc] :: P_Presc
  PData pd = empty();
  Void makePresc(Presc newp) {
     if newp ≠emptyString() then pd:+ newp fi }
                    :: (Doctor, treatm(Patient), rincr)
  Presc getPresc(Patient p){return last(pd/p)}
                    :: (Nurse, treatm(p), read)
  with Patient
    Presc getMyPresc() {return getPresc(caller)}
                    :: (Patient, treatm(caller), read)
    // allowed since a subject has read access to self data
  ... }
```

Figure 8.11: Example with subject awareness. As before, gray parts are implicit.

v	$::=$	$\ldots \mid pcs \mid$ nextId	added variables
s	$::=$	$\ldots \mid v :=$ **get** u	added statement

Figure 8.12: BNF syntax of additional constructs used in the operation semantics.

which will go to all *Sensitive* objects and lead to adjustments of all sensitive data in the system where subject is caller.

8.7 Operational semantics

The operational semantics of the considered language is given in Fig. 9.9. Data values are tagged with policy sets. Compared to the static analysis, we could use more expressive policies, in particular, we may use sets of objects to define the principals, rather than interfaces. However, for simplicity we use interfaces as principals, letting each interface denote the set of object supporting the interface, making the correspondence with the type system easier. We could also let the operational semantics define the *subject* and *owner* (i.e., creator) of the data, as well as other GDPR-relevant aspects such as *expiration time*, but this is ignored here since we focus on the aspects of the static system.

We briefly explain the main elements of the runtime system used in the operational semantics. A runtime configuration of an active object system is captured by a multiset of objects and messages (using blank-space as the

ASSIGN :
$$o : \mathbf{ob}(\delta, v := e; \overline{s}) \xrightarrow{empty} o : \mathbf{ob}(\delta[v := e], \overline{s})$$

IF-TRUE :
$$o : \mathbf{ob}(\delta, \texttt{if } b \texttt{ then } \overline{s1} \texttt{ else } \overline{s2} \texttt{ fi}; \overline{s}) \xrightarrow{empty} o : \mathbf{ob}(\delta[pcs := push(pcs, l)], \overline{s1}; pcs := pop(pcs); \overline{s})$$
$$\texttt{if } \delta[b] = true_l$$

IF-FALSE :
$$o : \mathbf{ob}(\delta, \texttt{if } b \texttt{ then } \overline{s1} \texttt{ else } \overline{s2} \texttt{ fi}; \overline{s}) \xrightarrow{empty} o : \mathbf{ob}(\delta[pcs := push(pcs, l)], \overline{s2}; pcs := pop(pcs); \overline{s})$$
$$\texttt{if } \delta[b] = false_l$$

WHILE :
$$o : \mathbf{ob}(\delta, \texttt{while } b \texttt{ do } \overline{s1} \texttt{ od}; \overline{s}) \rightarrow o : \mathbf{ob}(\delta, \texttt{if } b \texttt{ then } \overline{s1};\ \texttt{while } b \texttt{ do } \overline{s1} \texttt{ od fi}; \overline{s})$$

NEW :
$$o : \mathbf{ob}(\delta, v := \texttt{new } C(\overline{e}); \overline{s}) \xrightarrow{o\ \leftrightarrow\ \delta[\mathsf{nextOb}].C(\delta[\overline{e}])} o : \mathbf{ob}(\delta[v := o'], \overline{s})$$
$$o' : \mathbf{ob}(\delta_C[\mathsf{this} \mapsto o', \overline{cp} \mapsto \delta[\overline{e}]], init_C)$$
$$\texttt{where } o' = (fresh, C),$$
with *fresh* a fresh reference relative to C

ASYNC. CALL :
$$o : \mathbf{ob}(\delta, a!m(\overline{e}); \overline{s}) \xrightarrow{o \rightarrow \delta[a].m(\delta[\mathsf{nextId}, \overline{e}])} o : \mathbf{ob}(\delta[\mathsf{nextId} := next(\mathsf{nextId})], \overline{s})$$
$$\texttt{msg } o \rightarrow \delta[a].m(\delta[\mathsf{nextId}, \overline{e}])$$

SYNC. CALL :
$$o : \mathbf{ob}(\delta, v := a.m(\overline{e}); \overline{s}) \xrightarrow{o \rightarrow \delta[a].m(\delta[\mathsf{nextId}, \overline{e}])} o : \mathbf{ob}(\delta, a!m(\overline{e}); v := \texttt{get } \delta[\mathsf{nextId}]; \overline{s})$$

START :
$$\texttt{msg } o' \rightarrow o.m(u, \overline{c})$$
$$o : \mathbf{ob}((\alpha|\beta'), \mathbf{idle}) \xrightarrow{o' \twoheadrightarrow o.m(u, \overline{c})} o : \mathbf{ob}((\alpha|(\beta[\mathsf{caller} \mapsto o', \mathsf{callId} \mapsto u, \overline{y} \mapsto \overline{c}, pcs \mapsto empty()])), \overline{s})$$
$$\texttt{where } (m, \overline{y}, \beta, \overline{s}) \text{ is the body of } m$$
in the class of this

RETURN :
$$o : \mathbf{ob}(\delta, \texttt{return } e) \xrightarrow{\delta[\mathsf{caller}] \leftarrow \delta[\mathsf{this}].(\delta[\mathsf{callId}], \delta[e])} o : \mathbf{ob}(\delta, \mathbf{idle})$$
$$\texttt{msg } \delta[\mathsf{caller}] \leftarrow \delta[\mathsf{this}].(\delta[\mathsf{callId}], \delta[e])$$

QUERY :
$$\texttt{msg } o \leftarrow o'.(u, c)$$
$$o : \mathbf{ob}(\delta, v := \texttt{get } u; \overline{s}) \xrightarrow{o \twoheadleftarrow o'.(u,c)} o : \mathbf{ob}(\delta, v := c; \overline{s})$$

NO-QUERY :
$$\texttt{msg } o \leftarrow o'.(u, c)$$
$$o : \mathbf{ob}(\delta, \overline{s}) \xrightarrow{o \twoheadleftarrow o'.(u,c)} o : \mathbf{ob}(\delta, \overline{s})$$
$$\texttt{if get } u \notin \overline{s}$$

Figure 8.13: Operational rules defining small-step semantics with policies.

binary multiset union constructor). Each rule in the operational semantics deals with only one object o, and possibly messages, reflecting the nature of concurrent distributed active objects, communicating asynchronously. Remote calls and replies are handled by message passing. When a subconfiguration $\mathcal{C}$ can be rewritten to a $\mathcal{C}'$, this means that the whole configuration $\ldots\mathcal{C}\ldots$ can be rewritten to $\ldots\mathcal{C}'\ldots$, reflecting interleaving semantics. Each object o is responsible for executing all method calls to o as well as self-calls. An object has at most one active process, reflecting the remaining part of a method execution. For our programming language we need not consider futures or suspended processes, but such mechanisms can be added in a straight forward manner since they do not pose additional privacy challenges. In order to handle method returns, our semantics creates an identity for each call (like a local *future*) passed as an implicit parameter, and inserts **get** statements referring to the call identity. (see Figure 8.12). By lifting these call labels and **get** statements to the language syntax, we would obtain support for object-local futures, as described in [65].

Objects have the form

$$o : \mathbf{ob}(\delta, \overline{s})$$

where o is the object identity, δ is the current object state, and $\overline{s}$ is a sequence of statements ending with a **return**, representing the remaining part of the active process, or **idle** when there is no active process. A message has the form

$$\texttt{msg}\ o \rightarrow o'.m(\overline{e})$$

representing a call to m with o as caller, o' callee, and $\overline{e}$ actual parameters, or

$$\texttt{msg}\ o \leftarrow o'.(u, d)$$

representing a completion event where d is the returned value and u the identity of the call. In addition, **msg** $o \rightarrow I.m(\overline{e})$ denotes a broadcast to all objects supporting interface I.

The operational rules reflect small-step semantics. For instance, the rule for *skip* is given by

$$o : \mathbf{ob}(\delta, skip; \overline{s}) \xrightarrow{empty} o : \mathbf{ob}(\delta, \overline{s})$$

saying that the execution of *skip* has no effect on the state δ of the object.

The semantics in Figure 9.9 formalizes the notion of idleness, and generation of objects and messages, including a rule (*no-query*) for disposal of unused reply messages. Generation of identities for objects and method calls is handled by underlying semantic functions and implicit attributes.

The operational semantics uses some additional variables, like *pcs* ("program counter stack") for remembering the stack of policies corresponding to the nesting of if/while statements, and nextId for generating unique identities for calls. These appear as fields in the operational semantics (nextId initialized with some value and with a *next* function to generate new unique values). Furthermore, this is handled as an implicit class parameter, while callId and caller appear as implicit method parameters, holding the identity of a call and its caller, respectively.

The operational semantics uses an additional *query* statement, **get** u, for dealing with the termination of call statements. A synchronous call is treated as an asynchronous call followed by a **get** query. The query **get** u is blocking while waiting for the method response with identity u. The added constructs are shown in Figure 8.12. We let a denote an object expression, b a Boolean expression, o an object identity, u a method call identity, d a value (a data value or an object identity), and c a value tagged with a policy.

The state of an object is given by a twin mapping, written $(\alpha|\beta)$, where α is the state of the field variables $\overline{w}$ (including nextId) and class parameters $\overline{cp}$ (including this), and β is the state of the local variables $\overline{x}$ and formal parameters $\overline{y}$ (including callId and caller) of the current process. Look-up in a twin mapping, $(\alpha|\beta)[z]$, is simply given by $(\alpha + \beta)[z]$. The notation $\alpha[z := e]$ abbreviates $\alpha[z \mapsto \alpha[e]]$, and the notation $(\alpha|\beta)[v := e]$ abbreviates **if** v *in* β **then** $(\alpha \,|\, \beta[v \mapsto (\alpha|\beta)[e]])$ **else** $(\alpha[v \mapsto (\alpha|\beta)[e]] \,|\, \beta)$, where *in* is used for testing domain membership.

Method invocation is captured by the rules ASYNC CALL/SYNC. CALL. The generated call identity is locally unique, and globally unique in combination with the parent object. The call identity generated by this rule is passed through an invocation message, which is to be consumed by the callee object by the Rule START. When an object has no active process, denoted ***idle***, a method call can be selected for execution by rule START. The invocation message is removed from the configuration by this rule, and the identity of the call is assigned to the implicit parameter callId. With Rule RETURN, a return value is generated upon method termination and passed in a completion message together with the call identity stored in callId. The return value is then fetched by Rule QUERY. Note that a query statement blocks until the corresponding return value is generated by Rule RETURN, whereas asynchronous calls do not block. The QUERY rule says that $v := get\ u$, in object o is replaced by the assignment $v := d$ when the completion **msg** $o \leftarrow o'.(u, d)$ appears, and the completion message is removed from the configuration. If object o does not contain **get** u then the completion message is removed without any effect on o. This happens when the corresponding call was an asynchronous call, which is similar to one-way message passing. In Rule START, we assume that m is bound to a method with local state β (including default values) and code $\overline{s}$. Note that bindings for the parameters $\overline{y}$ and the implicit parameter nextId are added to the local state.

Object creation is captured by the rule NEW. The generated object identity is based on a non-deterministically generated reference (reflecting factors outside the program). Note that an object reference encodes the class name, which makes the rules more compact. Here $init_C$ denotes the initialization statements (the class constructor) of class C, and δ_C denotes the initial state of class C with default or initial values for the fields. The binding of class parameters and this is added explicitly in the rule. We obtain an active object by letting $init_C$ initiate internal activity, using asynchronous self calls to allow the object to interleave continued internal activity with reaction to external calls. The initialization statements of a program (given by the class constructor of an instantiation of the last class in the program) will typically create the other initial objects.

The semantics of an if-statement without an else-part, **if** b **then** s **fi**, is usually equivalent to **if** b **then** s **else** $skip$ **fi**. However, this is not the case with respect to policy tags. For instance, the policy after $x := \mathit{false}$; **if** b **then** $x := \mathit{true}$ **fi** is not the same as after **if** b **then** $x := \mathit{true}$ **else** $x := \mathit{false}$ **fi** when b is sensitive, because in the latter case the policy of x is not changed when the else branch is taken, while it is changed in the former case. A solution to this is discussed at the end of the next section (on related work). A while loop is handled by expanding **while** b **do** s **od** to **if** b **then** s; **while** b **do** s **od fi** upon execution of the while-statement. Void methods return the value *void*. We assume all methods end in a return statement, including class constructors, which end in **return** *void* (although omitted in the examples). We assume that assignments of the form $w :+ e$ are represented by $w := w + e$ at runtime, and that initial values given to fields or local variables is expanded to assignments, as described earlier. A rule for broadcasting is omitted; however, the semantics is similar to that of asynchronous calls.

The given language fragment may be extended with constructs for local (stack-based) method calls, e.g., by using the approach of [**DSOS**] and it may be extended with cooperative scheduling and synchronization control as in [61].

8.7.1 Runtime policies

We explain here the privacy aspects of the operational semantics. We assume that the program has passed the policy typing, and therefore the operational semantics does not include a duplication of the static policy requirements during reduction. We then prove that any policy level obtained at runtime guarantees the one calculated by the static policy typing. This property, called *policy soundness*, is stated by Theorem 2. It guarantees that the policy checks will be satisfied at runtime when based on the runtime policy levels. We also prove a progress property.

As mentioned, the semantics uses an additional variable *pcs* in each method, reflecting the stack of context policy levels of enclosing if- and while-branches. The top of the *pcs* stack reflects the policy of the innermost branch. At an entry to an if/while statement, *pcs* is pushed with the policy set of the test expression, and *pcs* is popped upon exit. Note that *pcs* can be local since it must be empty upon method return. The relationship between *pcs* and *pc* as used in the static checking, is given as part of Theorem 2.

At runtime the evaluation of an expression e gives a policy tag l, in addition to a (normal) value d. We let the tagged value d_l denote this result, and let c denote tagged values, and let $d_l.tag$ be l. When such a value is assigned to a program variable v, the binding $v \mapsto d_l$ is added to the state. The state of an object is given by a twin-mapping as above, but the values of variables are now bound to tagged values. Thus the values appearing in the semantics are all tagged. Each object identity has the form of a pair (oid, C) where C is the class of the object and oid a unique identity relative to C.

The evaluation of an expression e in a state δ and with policy context *pcs* is denoted $\delta[e]$, where the data value d is evaluated ignoring tags, resulting in a

ground term, i.e., a term with only constructor functions, and where the tag is defined by

$$level(\delta[pcs]) \sqcap tag(d)$$

where $level(\delta[pcs])$ is the *meet* of all the policies in the stack pcs, and where the tag of d is evaluated according to the policies of the constructor functions in d, letting type constructors of non-sensitive types give a $\{\bullet\}$ policy.

The runtime policy level of a variable v in an execution state can differ from that of the static level in the corresponding program point. There are several reasons for this. For instance, there can be many calls to the same method with actual parameters of different policy levels. The runtime system uses the policies of actual parameters whereas the static analysis uses that of the formal parameters. At the start of a method, the static analysis will assume the declared policy levels for fields, whereas at actual runtime levels might differ. This is clarified below.

8.7.2 Theoretical results

In order to keep the operational rules simple, we have assumed that programs are well typed and have passed the static policy checks. Still it is not obvious that a statically correct program cannot go wrong, if for instance the statically derived policies are not respected at runtime. We therefore show results reflecting soundness and progress.

We observe that each state of an object of class C in an execution *corresponds* to a (static) program state in class C, and that each expression (other than future-related variables) evaluated at runtime corresponds to an occurrence of an expression in the program text. To formalize this correspondence we associate a statement number with each statement in the code, and when a statement s is executed we may obtain the statement number by the syntax $\#s$. When an object is about to execute a statement s appearing in the program code, the *corresponding program state* is given by the static environment just before that statement, denoted $\Gamma^{\#s}$. The number also identifies the enclosing method and class. Since the last statement of a then-branch has the same next statement as the last statement of the else-branch we cannot distinguish these in the correspondence. We need a way to solve this, for instance by letting the execution of a method record the trace of program statements executed as a list of statement numbers. This gives sufficient information to see which branch is taken, but not to determine if the corresponding program state is before or after the end of the if-statement. However, this can be determined by the presence of the pop-statement: After pop the corresponding state is the one after the if-statement, and before pop the corresponding state is the last state of the branch as given by the trace. For a given state $o : \mathbf{ob}(\delta, \overline{s})$, we may therefore talk about the unique *corresponding program state* and its environment Γ. (Even runtime states starting with a get-statement correspond to program states, since the values of all program variables are the same as before the start of the call.)

Furthermore, we observe that the policies at runtime may differ from those at compile time, for instance in connection with parameter passing since the runtime policies are driven by the actual parameters while the static ones are driven by the declaration of the formal parameters. In general, the static rules use meet operations corresponding to worst-case analysis, while the runtime rules give the actual policy.

We first prove a soundness result saying that the runtime value of a variable or expression will have a policy that guarantees the one calculated statically according to Γ in the corresponding state: The run time value of a variable will have a policy that guarantees the one in the corresponding Γ. This also holds for expressions. For the special variable *pc*, there is a similar correspondence with *pcs*.

Theorem 2 (Soundness)**.** *Consider a given state $o : \boldsymbol{ob}(\delta, \overline{s})$ of an object o, and let Γ be the policy environment of the corresponding program state (as defined above) in method m of a class C. We have*

$$(C, m \vdash [\Gamma]\ v :: \mathcal{P}) \Rightarrow (\delta[v].tag \Longrightarrow \mathcal{P}) \tag{8.1}$$

$$(C, m \vdash [\Gamma]\ e :: \mathcal{P}) \Rightarrow (\delta[e].tag \Longrightarrow \mathcal{P}) \tag{8.2}$$

$$level(\delta[pcs]) \Longrightarrow \Gamma[pc] \tag{8.3}$$

where v is a program variable and e is an expression over program variables.

Proof. We first prove that property (2) follows from (1) and (3), and then prove property (1) and (3) by course-of-values induction on the derivation of executions as given by the operational rules and by induction on the derivation of the static compliance. We consider an arbitrary object o, which have state $o : \mathbf{ob}(\delta, \overline{s})$ with m of class C as the enclosing method and with Γ as the environment of the corresponding program state. Note that the derivation of static policies is terminating and deterministic. Each program state is assigned a unique environment Γ defining the static policies in the state. We let $\Gamma[e]$ denote the unique policy set $\mathcal{P}$ such that $C, m \vdash [\Gamma]\ e :: \mathcal{P}$.

Consider expressions e (other than variables), and assume (1) and (3) in a (runtime) state δ corresponding to a static program state with environment Γ. We prove that $\delta[e].tag \Longrightarrow \Gamma[e]$. This is trivial when $\delta[e].tag$ is $\{\bullet\}$ since $\{\bullet\} \Longrightarrow \mathcal{P}$ for any $\mathcal{P}$. It remains to prove that $\Gamma[e] \sqsubseteq \delta[e].tag$ holds when $\delta[e].tag$ is not $\{\bullet\}$. The static policy of a functional expression $f(\overline{e})$ of type T is given by $\mathcal{P}_T \sqcap \Gamma[pc]$. The runtime policy is based on the result of the evaluation. A functional expression $f(\overline{e})$ when evaluated gives a value d of type T' for $T' \leq T$ with policy $level(\delta[pcs]) \sqcap tag(d)$. It suffices that $tag(d) \Longrightarrow \mathcal{P}_T$. For $T' \leq T$ we have $\mathcal{P}_{T'} \Longrightarrow \mathcal{P}_T$.

Consider next property (3), $level(\delta[pcs]) \Longrightarrow \Gamma[pc]$. The *pc* and *pcs* variables are only changed at the entry and exit of if- and while-statements. The condition trivially holds over other statements. To simplify the connection between *pc* and *pcs*, we could add a local variable *pcs* in the static policy type rules. We then let the starting environment of each branch in an if- or while-statement

modify *pcs* by $\Gamma[pcs := push(pcs, \mathcal{P})]$ where $\mathcal{P}$ is the policy of the test, and let the final environment update *pcs* by $\Gamma[pcs := pop(pcs)]$. We may prove that $\Gamma[pc] = level(\Gamma[pcs])$ by induction over the policy rules. Instead of property (3), it then suffices to prove the property

$$level(\delta[pcs]) \Longrightarrow level(\Gamma[pcs]) \tag{8.4}$$

The induction hypothesis *IH* is now the conjunction of (1), (2), and (4). Below we will look at proof cases of the form $IH \Rightarrow IH'$ where IH' is *IH* with δ and Γ replaced by the state and environment after executing an arbitrary program statement.

Before entry to an if-statement with test b, we assume *IH* and must prove

$$level(\delta[pcs \mapsto push(\delta[pcs], l)][pcs]) \Longrightarrow level(\Gamma[pcs \mapsto push(\Gamma[pcs], \mathcal{P})][pcs])$$

where l is $\delta[b].tag$ and $\mathcal{P}$ is given by $C, m \vdash [\Gamma]\ b :: \mathcal{P}$. This reduces to

$$level(\delta[pcs]) \sqcap l \Longrightarrow level(\Gamma[pcs]) \sqcap \mathcal{P}$$

which follows from (2) and (4) of *IH* and monotonicity of $\sqcap$ with respect to $\Longrightarrow$, i.e., $(X \Longrightarrow X' \wedge Y \Longrightarrow Y') \Rightarrow (X \sqcap Y \Longrightarrow X' \sqcap Y')$, which is obvious.

Before exit of an if-statement we have the induction hypothesis at the end of the chosen branch, and we need to prove

$$level(\delta[pcs \mapsto pop(\delta[pcs])][pcs]) \Longrightarrow level((\Gamma_1 \sqcap \Gamma_2)[pcs \mapsto pop(\Gamma[pcs])][pcs])$$

where Γ_1 and Γ_2 are the respective environments at the end of the two branches. This reduces to

$$level(pop(\delta[pcs])) \Longrightarrow level(pop((\Gamma_1 \sqcap \Gamma_2)[pcs]))$$

which is trivial if $pop(\Gamma_1[pcs])$ is the same as $pop(\Gamma_2[pcs])$, which can easily be proved by induction over the derivations of the type and effect system. The situation for while-loops is similar.

Finally we consider variables: We observe that Γ is only modified by assignment-like statements (to fields and local variables), if- and while-statements, and method start, and program variables in δ are only updated by assignment-like statements and method start. We consider below assignments, if-statements, and method start. Assume *IH*. For an assignment $x := e$ we need to prove that

$$C, m \vdash [\Gamma[x \mapsto \mathcal{P}_e]]\ v :: \mathcal{P} \Rightarrow \delta[x \mapsto \delta[e]][v].tag \Longrightarrow \mathcal{P}$$

where $C, m \vdash [\Gamma]\ e :: \mathcal{P}_e$. For variables other than x this reduces to *IH*. For x we need to prove:

$$\delta[e].tag \Longrightarrow \mathcal{P}_e$$

which holds by the second conjunct of *IH*. Consider next the end of an if-statement. Let *IH* hold at the end of the chosen branch. We need to prove

$$C, m \vdash [\Gamma \sqcap \Gamma']\ v :: \mathcal{P} \Rightarrow \delta[v].tag \Longrightarrow \mathcal{P}$$

where Γ' is the environment at the end of the branch not chosen. This reduces to

$$\delta[v].tag \Longrightarrow (\Gamma \sqcap \Gamma')[v] \sqcap (\Gamma \sqcap \Gamma')[pc]$$

which is trivial since $\Gamma[v] \Longrightarrow (\Gamma \sqcap \Gamma')[v]$. At method start, i.e., when o has the form $o : \mathbf{ob}(\delta, \mathbf{idle})$, we need to prove for the case of a field w

$$\delta[w].tag \Longrightarrow \mathcal{P}_W$$

where W is the type of field w ($\mathcal{P}_W$ is the same as $\Gamma_C[w]$) and δ is the state resulting from the previous method execution, or the initial value of w. (The operational semantics ensures that a method start must follow a method end, or start from initial values, because the former creates an idle state and the latter represents the only way to continue from an idle state.) From IH we know that $\delta[w].tag \Longrightarrow \Gamma[w]$ where Γ is the environment of the previous method execution. From the premise of the P-METHOD we have that $\Gamma[w] \Longrightarrow \mathcal{P}_W$. By transitivity of $\Longrightarrow$ the rest follows. (The case of initial values is straight forward since the operation semantics and static analysis use the same expressions for the initial values.)

The situation for method parameters y is similar. At the start of a method execution (Rule START), the runtime policies of the method parameters $\overline{y}$ is given by the tags of the actual parameters, which by IH must guarantee the static policies of the parameters, which by Rule P-CALL must guarantee the policies of the formal parameter types $\overline{Y}$, which are fixed fora method m of class C, Altogether we have that runtime polices guarantee that static ones at method start.

Consider a query statement where c is the value received by the caller. In the runtime system this value is the same as the one returned by the callee, and the policy of the returned value at runtime must guarantee the static one by IH, and the static policy of the returned value guarantees the policy of the method's return type (say T) by Rule P-METHOD. Thus the runtime policy of c guarantees $\mathcal{P}_T$. On the caller side, the policy of the received result is that of c and in the static system it is the type of the method result, i.e., $\mathcal{P}_T$. We have therefore proved (1) for queries. New statements are similar. Asynchronous calls are simpler since no program variable is changed. ■

The above result does not have so much value if the runtime system allows programs that do not progress when they are supposed to do so, i.e., if no rule applies in a state where execution should continue. We therefore prove a progress property of the operational semantics saying that the execution of each object in a program will continue, unless the object is **idle** and there are no incoming messages reflecting method calls, or the object is blocked, i.e., trying to perform a **get** statement when the corresponding reply message has not appeared. Moreover, no errors are generated apart from undefined expressions.

Theorem 3 (Progress)**.** *Assume that the evaluation of program expressions is terminating normally with a defined value. If a configuration $\mathcal{C}$ rewrites to $\mathcal{C}'$ by the operation rules and $\mathcal{C}'$ cannot be reduced further, then each object*

*o : $\boldsymbol{ob}(\delta, \overline{s}) \in \mathcal{C}'$ is **idle** ($\overline{s}$ is **idle**) and there is no invocation message **msg** $o' \rightarrow o.m(...) \in \mathcal{C}'$, or o is blocked ($\overline{s}$ starts with **get** u) and there is no message **msg** $o \leftarrow o'.(u, c) \in \mathcal{C}'$.*

Proof. We show that for each statement one rule will apply as long as the conditions stated in the theorem do not hold. There is one unconditional rule for each statement, except **if**, which has two complementary rules, one for each case of the value of the if-test. No rules depend on a context condition, except the rules *start* and *query*, which require the presence of an appropriate message, but these are exactly the acceptable conditions stated in the theorem. Consider next the well-definedness of expressions over variables added in the operation semantics (*pcs*, *nextFut*, *myfuture*). The pop operations on *pcs* will terminate normally since each pop is preceded by a push. Each return statement must be preceded by a start statement, therefore *myfuture* will have a value. The special variable *nextFut* always has a value. ■

In particular, there will be no errors or exceptions, apart from undefined values resulting from evaluation of expressions. A call to a null object is possible, and this could lead to blocking of the caller object, if there is a get-statement for the call. Method-not-understood errors is captured by the underlying type checking [62].

8.8 Related work

The focus of this paper is the intersection of the GDPR, privacy policy formalization, and programming languages. This intersection is relatively recent and features several threads of active research such as policy specification, policy enforcement, monitoring, privacy by design, language based privacy, privacy enhancing technology.

Several attempts have been made to express privacy polices, through a language with formal syntax and semantics such as XACML [105], EPAL [6], APPEL [75], and XPref [2]. An analysis of these policy languages can also be found in [10, 68]. Privacy restrictions are also expressed formally using ontologies [15] or dedicated logics such as [9, 12, 18]. However, a direct comparison of these policy languages and logics with our policy language is not straight forward, mainly because we focus on policy aspects that can be verified statically and can only express limited aspect of a policy, while these policy languages can express a wider range of privacy restriction. In contrast to the mentioned policy languages, we focus on static checking and in particular check compliance of program, by class-wise analysis.

Access control models, such as discretionary access control (DAC) [69] and role based access control (RBAC) [98], have been historically utilized in order to support security requirements [32]. In RBAC, permissions (to perform operations) are associated with a role or set of roles [98]. Thus there are common features in our work and RBAC. In addition to the hierarchies of roles and access rights supported by RBAC, our framework introduces hierarchies of purposes to control

role access. However, our work uses static analysis while RBAC uses runtime analysis. In a literature review [42], by Fernández-Alemán et al. identifies the access control models deployed by electronic health records (EHR), where 35 of 45 reviewed articles used access control methods. Interestingly, 27 of those 35 specifically used RBAC. However, these conventional access models are not designed to enforce privacy policies [45], due to lack of several privacy protection requirements (e.g. purpose). In order to express purpose (and other privacy-related aspects), the RBAC model is extended, as in [76, 82, 121].

Privacy by Design (PbD) has been discussed and promoted from several viewpoints such as formal approaches [71, 100, 112], privacy engineering [30, 49, 86], privacy-enhancing technologies (PET) [44, 52], and privacy design patterns [26, 59]. Tschantz and Wing, and Daniel Métayer, discuss the significance of formal methods for foundational formalizations of privacy related aspects [71, 112]. In [112], Tschantz and Wing point out the usefulness of mathematical formulations of privacy notions for the purpose of guiding the development of privacy preserving technologies and making it easier to spot privacy violations. In [100], Schneider discusses the main ideas of *Privacy by Design* and summarizes key challenges (purpose, right to be forgotten, consent, and compliance) in achieving privacy-by-construction and probable means to handle these challenges. Our work addresses the challenges concerning purpose and (at least partially) compliance "by construction". Hoepman, in [59] derives and defines eight privacy design strategies, from existing privacy principles and data protection laws. The engineering aspects of privacy by design is addressed, but there is a lack on how to apply them in practice. In our work, we adhere to several privacy design strategies such as separating and hiding the data and encapsulation in an object-oriented context.

Hayati and Abadi [53] describe a language-based approach based on information-flow control, to model and verify aspects of privacy policies in the Jif (Java Information Flow) programming language. In this approach data collected for a specific purpose is annotated with Jif principals and then the methods needed for a specific purpose are also annotated with Jif principals. Explicitly declaring purposes for data and methods ensures that the labeled data will be used only by the methods with connected purposes. Purposes are organized in a purpose hierarchy, where sub-purposes can be declared using the (Jif specific) *acts-for* relation. However, this representation of purpose is not sufficient to guarantee that principals will perform actions compliant with the declared purpose. In contrast, this can be checked statically in our approach, because principals are restricted by purposes.

Basin et al. [11] propose an approach that relates a purpose with a business process and use formal models of inter-process communication to demonstrate GDPR compliance. Process collection is modeled as data-flow graphs which depict the data collected and the data used by the processes. Then these processes are associated with a data purpose and are used to algorithmically generate data purpose statements (i.e., specifying which data is used for which purpose) and detect violation of data minimization. Since in GDPR, end-users should know the necessary purpose of data collection, some works such as [11] propose

to audit logs and detect if a system supports a purpose. In a continuation of this work, in [4] Arfelt et al., show how such an audit can be automated by monitoring. Automatic audits and monitoring can be applied to a system like ours as a complementary step to verify how it complies with the GDPR. Besides, our work is more focused on integrating such legal instruments during the design phase, using formal language semantics.

Anthonysamy et al. [3] demonstrate a *semantic-mapping* approach to infer function specifications from semantics of natural language. This technique is useful in compliance verification as it aids in identification of program constructs that implements certain policies. The authors implement this technique in a tool, CASTOR, which takes policy statements (in natural language) and source code as input and outputs a set of semantic mappings between policies and function specifications (function name, associated class, parameters etc.). In [102], Sen et al. develop and demonstrate techniques for policy compliance checking in big data systems. Privacy policies, are specified using a policy language *LEGALEASE*, and restrict the information-flow based on store (data store), purpose, role, and other considerations. And then *GROK* (data inventory tool) maps code-level schema elements to data types in *LEGALEASE*. Compliance checking then reduces to information flow analysis.

In [1], Adams and Schupp consider black-box objects that communicate through messages. The approach is centered around algorithms that take as input an architecture and a set of privacy constraints, and output an extension of the original architecture that satisfies the privacy constraints. This work is complementary to ours in that it puts restrictions on the run-time message handling. In contrast to our work, the approach does not concern analysis of program code. In [43], Ferrara and Spoto discuss the role of static analysis for GDPR compliance. The authors suggest combining taint analyses and backward slicing algorithms to generate reports relevant for the various actors (i.e., data protection officers, chief information officers, project managers, and developers) involved at various stages of GDPR compliance. In particular, taint analysis is performed on each program statement and then the data-flow of sensitive information is reconstructed using backward-slicing. These flows are then abstracted into the information needed by the compliance actors.

Dynamic flow sensitivity [54] also applies to privacy, as pointed out by Schneider in [100]. A branching statement with sensitive information in the test, may indirectly leak privacy information if a variable changed in one branch is not changed in the other branch. This is not a problem in the static analysis, since after a branching construct the information of all branches are combined. But it is a problem in the operational semantics, since there you only see the chosen branch. To avoid this problem in our operational semantics, we take the following approach: For an if-statement with sensitive information in the test, we add trivial assignments $v := v$ to ensure that the variables changed in one branch also are changed in the other branch. Such an assignment will upgrade the privacy policy of v with $level(pcs)$, which prevents branching-related privacy leakage. (While-statements can be handled similarly.)

8.9 Conclusion

In this paper we started by investigating challenges and opportunities with the GDPR from a language-based perspective. Specifically we focused on the *data protection by design* principle, embedding privacy requirements into a programming language, and discussed the relevance for the OODS setting where all interaction between objects is made through interfaces, so-called interface abstraction.

We defined a specification language for formulating privacy policies, and discussed static and runtime privacy polices, and formalized a concept of static privacy policies as well as the notion of policy compliance. We chose three primary constituents of a privacy policy, namely *principal, purpose*, and *access right*. Such policies are meaningful at compile time, but cover only a subset of the GDPR aspects. We show how privacy policies can be declared for methods and data types, restricting the usage of sensitive data. The policy specification language can be added to any programming object-oriented programming language supporting interface abstraction.

We have formulated rules for privacy policy compliance, and these rules are given by an extended type and effect system for a high-level imperative language supporting active objects, extended with privacy policy specifications. The problem of checking a program's compliance reduces to efficient type-checking. If the program satisfies the checks, then there is no violation of the stated privacy policies. Implication in the other direction is not guaranteed, due to over-approximation in the static analysis. However, the case study demonstrates that the static analysis covers realistic scenarios.

We distinguish between *read*, *write*, and *incr* access rights. For a given principal and purpose, *incr* allows addition of personal information but without read access to existing personal information, whereas the combination of *read* and *incr* (*rincr*) allows both. These different access rights proved practically valuable in the case study related to healthcare, allowing us to differentiate the roles of a nurse (*read*), doctor (*rincr*), and lab assistant (*incr*). We have briefly discussed how to improve the analysis so that a data subject has access to personal data about herself, adding *self* as an additional access right.

The combination of method and data type policies allows class-wise static checking. It also encourages reuse of policy specifications and makes it possible to detect too strong or too weak policies by means of the static analysis (as discussed at the end of Section 8.3). A challenge of object-oriented programming is that not all classes represent principal actors, and will therefore not be a natural part of policies on data types. We compensate this by a notion of transfer of principle rights from caller to callee.

Furthermore, we have defined an operational semantics with policy tags on sensitive data, and proved soundness of the static compliance analysis with respect to the operational semantics. Finally, we have shown a progress property. In the future we would like to work out a larger case study, and in particular focus on the dynamic policy management and consent as outlined here and in [106].

Acknowledgements. We thank the anonymous reviewers for their comments and feedback, which helped us to improve the manuscript.

This work was partially supported by the project IoTSec - Security in IoT for Smart Grids, with number 248113/O70 part of the IKTPLUSS program funded by the Norwegian Research Council, and by the project SCOTT (www.scott- project.eu) funded by the Electronic Component Systems for European Leadership Joint Undertaking under grant agreement No 737422.

Authors' addresses

Shukun Tokas University of Oslo, Postboks 1337 Blindern, 0316 Oslo, Norway, shukunt@uio.no

Olaf Owe University of Oslo, Postboks 1337 Blindern, 0316 Oslo, Norway, olaf@uio.no

Toktam Ramezanifarkhani University of Oslo, Postboks 1337 Blindern, 0316 Oslo, Norway, toktamr@uio.no

Chapter 9

A formal framework for consent management

Shukun Tokas, Olaf Owe

Published in *Proceedings of the 40th International Conference on Formal Techniques for Distributed Objects, Components, and Systems, FORTE 2020*, June 2020, pp. 169–186. DOI: 10.1007/978-3-030-50086-3$_{1}$**0**.

Abstract

The aim of this work is to design a formal framework for consent management in line with EU's General Data Protection Regulation (GDPR). To make a general solution, we consider a high-level modeling language for distributed service-oriented systems, building on the paradigm of active objects. Our framework provides a general solution for data subjects to observe and change their privacy settings and to be informed about all personal data stored about them. The solution consists of a set of predefined types for privacy related concepts, a formalization of policy compliance, a set of interfaces that forms the basis of interaction with external users for consent management, a set of classes that is used in interaction with the runtime system, and a runtime system enforcing the consented policies.

9.1 Introduction

In response to the emerging privacy concerns, the European Union (EU) has approved the General Data Protection Regulation (GDPR) [41] to strengthen and impose data protection rules across the EU. This regulation requires controllers that process personal data of individuals within EU and EEA, to process personal information in a "lawful, fair, and transparent manner". Article 6 and Article 9 of the regulation [41] provide the criteria for lawful processing, such as consent, fulfillment of contractual obligation, compliance with a legal obligation etc. The regulation (including several other data protection laws) recognises consent as one of the lawful principles for legitimate processing, and Article 7 sets out the conditions for the processing personal data when relying on consent.

The authors were partially supported by IoT-Sec (NRC) (https://its-wiki.no/wiki/IoTSec:Home) and SCOTT (EU)(www.scott-project.eu).

A data subject's consent reflects his/her agreements in terms of the processing of personal data. The regulation indicates that the consent must specifically be given for the particular *purpose* of processing. It is also indicated in Recital 43 that the data subject should be given a free choice to accept or deny consent for specific purposes, rather than having one consent for several purposes. In particular our focus is on processing of personal data when *consent* is the legal ground, i.e., processing is valid only if a data subject has given consent for the specific purpose, otherwise processing of the personal data should cease. Moreover, this can be extended to incorporate other applicable legal grounds, such as vital interest, legitimate interest etc, but a discussion on this will be out of scope of this work.

Furthermore, Article 15 of the regulation prescribes that the data subject has *Right of Access*, which requires the data controllers to provide the data subject with his/her personal data, the purposes of processing, the legal basis for doing so, and other relevant information (see Article 15[41]). WP29 recommends controllers to introduce tools, such as a privacy dashboards through which the data subject can be informed and engaged regarding the processing of their personal data [5]. The regulation also introduces an obligation for data controllers to demonstrate compliance, i.e., accountability (see Article 5(2) [41]). These requirements are likely to pose substantial administrative burden. This work is an attempt to design a pragmatic solution to address these requirements, using a formal approach. In particular, our framework covers certain aspects of *privacy principles* (Article 5), *lawfulness of processing* (Article 6), *privacy by design* (Article 25) and *data subject access request* (Article 15). Due to the nature of these requirements and space constraints, we cover these requirements partially.

The privacy requirements in the data protection regulations are defined informally, therefore, to avoid ambiguity the policy language equipped with a formal semantics is essential [71]. It is essential that the policy terminology establishes a clear link between the law and the program artifacts. For this, we let privacy policies and consent specifications be expressed in terms of several predefined names, reflecting standard terminology (allowing names to be added as needed). It is necessary that the policy terminology used towards the data subject is simple but with a formal connection to the underlying programming elements. We have previously studied static aspects of privacy policies and static checking of policy compliance from a formal point of view, a brief overview is given in [109].

The aim of this work is to design a formal framework for consent management where a data subject can change his/her privacy settings through predefined interfaces, which could be part of a library system. The data subjects are seen as external system users without knowledge of the underlying program. Data subjects may interact with the system at runtime through a user-friendly interface (e.g. a privacy dashboard), to view current privacy settings and update these settings. To make a general solution, we consider a high-level modeling language for distributed service-oriented systems, building on the paradigm of *active objects* [61, 85]. The method for protecting access to personal data in this setting comprises of: tagging the data with $(subject, purpose)$ pairs; associating

A	$::=$	$no \mid read \mid incr \mid write \mid rincr \mid wincr \mid full$	access rights
$\mathcal{RD}$	$::=$	$\texttt{purpose}\ R^+\ [\texttt{where}\ Rel\ [\texttt{and}\ Rel]^*]$	purpose declaration
Rel	$::=$	$R^+ < R^+$	sub-purpose declaration
P	$::=$	$I \mid o$	principals
$\mathcal{P}$	$::=$	(P, R, A)	policies
C	$::=$	$pos(P, R, A) \mid neg(P, R, A)$	consented policies
Q	$::=$	C^*	policy list

Figure 9.1: BNF syntax definition of the policy language. I ranges over interface names, R over purpose names, and P over principal names. A principal is given by an object or an interface (representing all objects of that interface). Superscripts * and $^+$ denote general and non-empty repetition, respectively.

a *purpose* to each method accessing personal data; storing consented policies of a subject in a subject object; deriving an *effective* policy for the access from the executing method and data tags; and comparing the effective policy with the current consented policies to determine if it is a valid operation.

The main contribution of this research is a framework that consists of: *(i)* a policy specification language; *(ii)* a formalization of runtime policy compliance; *(iii)* predefined interfaces and classes for consent management; *(iv)* a run-time system for dynamic checking of privacy compliance, with built-in generation of runtime privacy tags when new personal data is created. We prove a notion of runtime compliance with respect to the consented policies.

Paper outline. The rest of the paper is structured as follows. Section 9.2 presents the policy and consent specification language, a formalization of policy compliance, and the core language. Section 9.3 introduces the functionality for consent management. Section 9.4 presents tag generation, dynamic checking and an operational semantics. Section 9.5 discusses related work, and Section 9.6 concludes the paper and discusses future work.

9.2 Language setting

In order to formalize the management and processing of personal information, we introduce basic notions for privacy policies and consent in Section 9.2.1, and introduce a small language for interface and class definitions in Section 9.2.2.

9.2.1 Policy and consent specification

Privacy policies are often described in natural language statements. To verify formally that the program satisfies the privacy specification, the desired notions of privacy need to be expressed explicitly. To formalize such policies, we define a policy specification language. In our setting, a privacy policy is a statement that expresses permitted use of the personal information by the declared program

```
purpose treatm, health_care where treatm < health_care

policy P_Doc = (Doctor, treatm, rincr) // general policy
consent pos(P_Doc) = (Doctor, health_care, write) // general positive consent
consent neg(P_MyDoc) = (Dr.Hansen, treatm, full) // specific negative consent
```

Figure 9.2: Sample purpose and policy definitions. Here $Dr.\,Hansen$ is a principal object.

entities. In particular, a policy is given by triples that put restrictions on what *principals* can access the personal data for specific *purposes* and *access-rights*. That being the case, a policy $\mathcal{P}$ is given by a triple (P, R, A), where:

i) P describes a principle that can access personal information and is given by an object representing a principal, or by an interface (representing all objects supporting that interface). Interfaces are organized in an open-ended inheritance hierarchy, letting $I < J$ denote that principal I is a subinterface of J and letting $o < I$ if object o supports I. We let $\leq$ denote the transitive and reflexive extension of $<$. As an example,

$$Specialist < Doctor < HealthWorker$$

ii) The purpose name R describe the specific purpose for which personal data can be used. Such purpose names are organized in an open-ended directed acyclic graph, reflecting specialization. For instance, the declaration

$$\texttt{purpose}\ spl_treatm, treatm\ \texttt{where}\ spl_treatm < treatm$$

makes *spl_treatm* more specialized purpose than *treatm*. If data is collected for the purpose of *spl_treatm* then it cannot be used for *treatm*. However, if it is collected for the purpose of *treatm* then it can be used for *spl_treatm*. We let $\leq$ denote the transitive and reflexive extension of $<$, and let the predefined purpose *all* be the least specialized purpose.

iii) Access rights A describe the permitted operations on personal data, and are given by a lattice, with *full* and *no* as top and bottom and with a partial ordering $\sqsubseteq_A$: *read* gives read access, *write* gives write access (without including read access), *incr* allows addition of new information but neither read nor write is included. The join of *read* and *incr* is abbreviated *rincr*, the join of *write* and *incr* is abbreviated *wincr*, while the join of *read* and *write* is *full*.

The language syntax for policies is summarized in Fig. 9.1, where [ ] is used as meta-parenthesis, and superscripts * and $^+$ denote general and non-empty repetition, respectively. Sample policies are given in Fig. 9.2.

Definition 21 (Policy Compliance). *Policy compliance, $\sqsubseteq$, is defined by*

$$(P', R', A') \sqsubseteq (P, R, A) \equiv P' \leq P \wedge R' \leq R \wedge A' \sqsubseteq_A A$$

Thus, a policy $\mathcal{P}'$ complies with $\mathcal{P}$ if it has the same or smaller interface, the same or more specialized purpose, and the same or weaker access rights.

In order to deal with both addition and removal of policies, we organize the policies in a list of negative and positive policies, such that the newest and most significant policy is last in the list. A positive consent has the form $pos(\mathcal{P})$, where $\mathcal{P}$ is a policy triple, meaning that access to personal data requiring p is allowed. A negative consent has the form $neg(\mathcal{P})$, meaning that access to personal data requiring p is forbidden. The disjoint union of these two forms is captured by the type *Consent*. Consented policies are organized in a *Consent* list. We define compliance of policies with respect to such a list L by:

$$\begin{array}{lcl} \mathcal{P} \sqsubseteq \epsilon & = & \mathit{false} \\ \mathcal{P} \sqsubseteq (L; pos(\mathcal{P}')) & = & \textbf{if } \mathcal{P} \sqsubseteq \mathcal{P}' \textbf{ then } \mathit{true} \textbf{ else } \mathcal{P} \sqsubseteq L \\ \mathcal{P} \sqsubseteq (L; neg(\mathcal{P}')) & = & \textbf{if } \mathcal{P} \sqsubseteq \mathcal{P}' \textbf{ then } \mathit{false} \textbf{ else } \mathcal{P} \sqsubseteq L \end{array}$$

where _; _ denotes list append. Thus positive or negative policies later in the list (capturing newer ones) override policies earlier in the list (capturing older ones) with smaller policy triples. This gives a simple way to upgrade and downgrade consent, and with a uniform treatment of negative as well as positive consent.

9.2.2 A high-level language for active objects

In the setting of active objects, the objects are autonomous and execute in parallel, communicating by so-called asynchronous method invocations. Object-local data structure is defined by data types. Classes are defined by an imperative language while data types and associated functions are defined by a functional language. We assume interface abstraction, i.e., remote field access is illegal and an object can only be accessed though an interface. This allows us to focus on major challenges of modern architectures, without the complications of low-level language constructs related to the shared-variable concurrency model.

A strongly typed language for active objects based on [61] is given in Fig. 9.3. The programs we consider are defined by a sequence of declarations of interfaces (containing method declarations), classes (containing class parameters, fields, methods and class constructors), and data type definitions. Class parameters are like fields, but with read-only access. A subclass inherits class parameters, fields, and methods (and the class constructor) unless redefined. A method m may have a cointerface Co given by the with clause, **with** Co, restricting callers to objects supporting interface Co (this is checked statically and allows type-correct call backs). Each method dealing with personal data must have an associated purpose, given at the end of the method definition (:: R), if any, otherwise the one declared for the method in the interface, if any, or otherwise the one declared for the interface. Methods may declare local variables and end with a return statement. We include standard statements such as skip, assignment (:=), object creation (**new**), if- and while-statements, and we allow blocking calls ($v := o.m(\overline{e})$) where o is the callee and $\overline{e}$ is the list of actual parameters, and

Pr	$::=$	$[\mathcal{T} \mid \mathcal{RD} \mid In \mid Cl]^*$	program
$\mathcal{T}$	$::=$	**type** $N\ [\overline{T}] = \langle\text{type_expression}\rangle$	type definition
T	$::=$	Int | Any | Bool | String | Void | List$[T] \mid I \mid N$	interfaces and types
In	$::=$	**interface** I [**extends** I^+] $\{D^*\}[::R]$	interface declaration
Cl	$::=$	**class** C $([T\ z]^*)$	class definition
		[**implements** I^+] [**extends** C]	support, inheritance
		$\{[T\ w\ [= ini]]^*\ [B\ [::R]]$	fields and constructor
		[[**with** I] $M]^*\}$	methods
D	$::=$	**op** $T\ m([T\ y]^*)\ [::R]$	method signature
M	$::=$	**op** $T\ m([T\ y]^*)\ [B]\ [::R]$	method definition
B	$::=$	$\{[T\ x\ [= rhs];]^*\ [s]$ [; **return** rhs]$\}$	method blocks
v	$::=$	$w \mid x$	assignable variable
e	$::=$	$v \mid y \mid z \mid$ this | caller | void | $f(\overline{e}) \mid (\overline{e})$	pure expressions
ini	$::=$	$e \mid$ **new** $C(\overline{e})$	initial value of field
rhs	$::=$	$ini \mid e.m(\overline{e})$	right-hand sides
s	$::=$	skip | $s;s \mid v := rhs \mid v :+ e \mid e!m(\overline{e}) \mid I!m(\overline{e})$	assignment and call
		| **if** e **then** s [**else** s] **fi** | **while** e **do** s **od**	if and while

Figure 9.3: BNF syntax of the core language, extended with purpose specifications $(::R)$. A field is denoted w, a local variable x, a method parameter y, a class parameter z, type names N, expressions e, and expression lists $\overline{e}$. The brackets in $[T]$ and $[\overline{T}]$ are ground symbols. Function symbols f range over pre-/programmer-defined functions/constructors with prefix/mixfix notation.

```
interface PrivacySettings {
  with User
    op PolicyList seeMyPolicies() // the current policy list is sent to the user
    op Bool addPolicy(Policy p) // add a new policy (return false if redundant)
    op Bool remPolicy(Policy p) } // remove a policy (return false if redundant)
```

Figure 9.4: Interface declarations for subject's privacy settings

asynchronous calls $o!m(\overline{e})$ and broadcasts $I!m(\overline{e})$ to all objects of interface I. The incremental update $v :+ e$ extends a list v with one or more elements e.

We consider pure expressions, including products $(e_1, e_2, \ldots)$, lists, and function applications $f(\overline{e})$ where f may be a defined function or a constructor function (including ";" for lists and constants such as nil, $void$, 0, 1, 2, etc.). A value is a variable-free expression with only constructor functions, such as the list $nil; 1; 2; 3$.

```
interface Sensitive{
  with Subject
    op Void requestMySensitiveData() //  a subject (the caller) requests to see
  }                              //  all personal data in the sensitive object about her

interface Subject extends PrivacySettings, Sensitive, Principal {
  with Sensitive
    op Void receiveMySensitiveData(List[TaggedData] tl)
  with User
    op Void collectMyData() //initiate collection of personal data about the subject
    op List[TaggedData] seeMyData() //  the received info is sent to subject user
  } :: all
```

Figure 9.5: Interface declarations for sensitive objects and data subjects.

9.3 Consent management

The policy settings of each data subject may change dynamically during runtime in interaction with the external users. In order to handle this, we define a runtime system where personal data values are tagged with specification of data subjects and processing purposes The runtime system will check that every access to personal data complies with the consented policies. Since there could be a huge amount of personal information in a distributed system, it is essential that the information in the tags is minimized. *Our framework includes a general solution for subjects to observe and change their privacy settings.* We chose to let the information about the consented policies be stored separately from the tags. The tags are generated by the runtime system as explained in detail in Section 9.4.1. The consented policy may change dynamically, in contrast to the information in the tags, which do not change. By combining the core information in the tags with the dynamically changing consent information, we are able to keep the information in the tags relatively small.

We let interface *Principal* correspond to a system user, be it a person, an organization, or other identifiable actor. Interface *PrivacySettings* (Fig. 9.4) defines methods for accessing and resetting consented policies by the data subject, while the subinterface *Subject* (Fig. 9.5) defines methods for consent management including functionality for requesting and updating policy settings. For each data subject there is an associated object (i.e., the subject object) supporting *PrivacySettings* and *Subject*, and this object is used to manage the privacy settings and policies in interaction with an external user (for instance through an app on a mobile phone). The subject object will contain the consented policies and is used when personal data about the data subject is accessed, in order to check compliance with the consented policies as explained in the operational semantics. In addition, it is used to manage the collection of personal data from sensitive objects. Thus the class *SUBJECT*, supporting interface *Subject*, deals

with handling of consented policies and collection of personal data.

The interface *PrivacySettings* specifies the interface for updating consent (Fig. 9.4). It includes methods for adding and removing consent by the user such that after successful addition/removal of a policy $\mathcal{P}$, that policy (or a smaller) allows/denies access to personal data. There is also functionality for an user to check her current policy settings, through method *seeMyPolicies*, which returns all policies of that user.

Class *PRIVACYSETTINGS* in Fig. 9.6 implements *PrivacySettings* by storing the consented policies in a field variable *consented*, which is a *Consent* list (i.e., list of consented policies). The add operation $addPolicy(\mathcal{P})$ adds $pos(\mathcal{P})$ at the end of the consented list (unless $\mathcal{P} \sqsubseteq consented$ holds already), and the remove operation $remPolicy(\mathcal{P})$ adds $neg(\mathcal{P})$ at the end of the list (unless $\mathcal{P} \sqsubseteq consented$ gives false, in which case it is redundant). The consent list of a subject S can be initialized with some initial policies, including $(S, all, rincr)$ for self access.

One may also remove redundant consented policies in the list when new ones are added, using the following strategy: A positive policy $pos(\mathcal{P})$ occurring in a list L is *redundant* in the list if $\mathcal{P} \sqsubseteq L'$ holds where L' is the list with this occurrence removed. Similarly, a negative policy $neg(\mathcal{P})$ occurring in L is *redundant* if $\mathcal{P} \sqsubseteq L'$ gives false. In these cases L can replaced by L' in order to simplify future compliance tests by limiting the size of the *consented* list.

9.3.1 Data collection from sensitive objects to data subjects

In order to restrict processing of personal information, we define an interface *Sensitive*, which will be the superinterface of all objects handling personal data. The interfaces *Subject* and *Sensitive* in Fig. 9.5 define the functionality for collection of personal information for subjects and define consent management. Class *SUBJECT* in Fig. 9.7 gives an implementation. A call to the method *collectMyData* on a subject object from the corresponding user will start a process to collect all personal information about the subject. The broadcast $Sensitive!requestMySensitiveData()$ sends a *requestMySensitiveData* message to all objects implementing *Sensitive*. A sensitive object may receive a *requestMySensitiveData* request from a subject object (*caller*) and will then react by collecting the personal data tagged with the subject and send it back to the subject object through the method *receiveMySensitiveData*. This data is then collected incrementally and stored in a (tagged) list, *mydata*, which can be accessed by the corresponding user using *seeMyData* or *seeData*. This class may be used as superclass of objects supporting *Subject*. The method *requestMySensitiveData* is provided by, and implemented in, the runtime system as explained in Section 9.4.2.

Interface *Subject* has *all* as declared purpose, and all methods in the interface and class inherit this purpose. The access to personal information in *SUBJECT* complies with the general policy $(S, all, rincr)$ for a subject object S.

```
class PRIVACYSETTINGS (User user) implements PrivacySettings {
 List[Consent] consented; //  the current privacy policies
with User
   op PolicyList seeMyPolicies()
     {return (if caller = user then consented else nil fi)}
   op Bool addPolicy(Policy p) { //  add a new policy (return false if redundant)
     Bool ok := (caller = user); if ok then
      if p ⊑ consented then ok := false
      else consented :+ pos(p) fi fi; return ok } //  incremental update
   op Bool remPolicy(Policy p){ //  add a negative policy (return false if redundant)
     Bool ok := (caller=user); if ok then
      if p ⊑ consented then consented :+ neg(p)//  incremental update
      else ok := false fi fi; return ok }
...}
```

Figure 9.6: The implementation of privacy settings and policy changes.

```
class SUBJECT implements Subject // thereby also PrivacySettings
    extends PRIVACYSETTINGS {
  List[TaggedData] mydata; //  personal data collected about subject
  op Void receiveMySensitiveData(List[TaggedData] tl){
    mydata :+ tl } //  could also include info of caller, i.e., mydata :+ (caller,d)
with User
  op Void collectMyData(){
    if caller = user then
      mydata := nil; //  clear list
      Sensitive!requestMySensitiveData() fi } //broadcast to all sensitive objects
  op List[TaggedData] seeMyData(){
    return (if caller = user then mydata else nil fi)}
}
```

Figure 9.7: The implementation of Subject.

9.4 Runtime system

The operational semantics of the considered language is given in Fig. 9.9. Data values are tagged with set of pairs of subject and purpose. A runtime configuration of an active object system is captured by a multiset of objects and messages (using blank-space as the binary multiset union constructor). Each object o is responsible for executing all method calls to o as well as self-calls. An object has at most one active process, reflecting the remaining part of a method execution. Objects have the form

$$o : \mathbf{ob}(\delta, \overline{s})$$

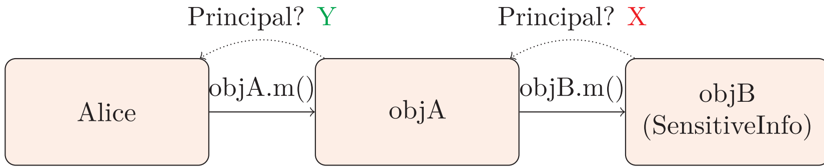

Figure 9.8: Call chain. Here Alice is the principal of the method execution on objB.

where o is the object identity, δ is the current object state, and $\overline{s}$ is a sequence of statements ending with a **return**, representing the remaining part of the active process, or **idle** when there is no active process. The state of an object δ is given by a twin mapping from variable names to tagged values, written $(\alpha|\beta)$, where α is the state of the field variables $\overline{w}$ and class parameters $\overline{cp}$ (including this), and β is the state of the local variables $\overline{x}$ and formal parameters $\overline{y}$ of the current process. Look-up in a twin mapping, $(\alpha|\beta)[v]$, is simply given by **if** v *in* β **then** $\beta[v]$ **else** $\alpha[v]$, where *in* is used for testing domain membership. The notation $\alpha[v \mapsto e]$ denotes map update, and the notation $(\alpha|\beta)[v := e]$ abbreviates **if** v *in* β **then** $(\alpha \mid \beta[v \mapsto (\alpha|\beta)[e]])$ **else** $(\alpha[v \mapsto (\alpha|\beta)[e]] \mid \beta)$.

In addition, the operational semantics defines the system variables pcs and nextId, which appear in the state of each object (in α). The "program counter stack" pcs is used for storing the stack of tags on the conditions corresponding to the nesting of enclosing if/while statements, and nextId is used for generating unique identities for calls.

Furthermore, the self reference this is handled as an implicit class parameter, while callId and caller appear as implicit method parameters, holding the identity of a call and its caller, respectively.

Example. Consider some personal health data with the tag $\{(Alice, treatm)\}$, and assume the consented policies $(\ldots; pos(Doctor, treatm, full))$ in object *Alice*. A Doctor can then read the data since there is a matching positive policy with at least read access where Doctor is the principal and the purpose of the current method is *treatm* or less. However, for the consented list $(\ldots; pos(Doctor, treatm, full); neg(Bob, treatm, read))$, where *Bob* is a doctor object, read access will be denied due to the presence of negative policy.

9.4.1 Runtime tagging of values

The runtime checking uses two special notions: The *current purpose*, denoted $R_{current}$, is the purpose of the enclosing method, which we assume is statically specified, as in [109]. (Alternatively one could take the purpose defined in some other way, for instance by data-flow graphs as in [11].) Secondly, we define the *current principal*, denoted $P_{current}$, as the first principal object found by following the dynamic call chain from a method execution as illustrated in Fig. 9.8 (ignoring non-principals such as $objA$).

The runtime evaluation of an expression e gives a tagged value c of form d_l with a tag l. In a method execution the evaluation of an expression e in a state δ and with policy context *pcs* is denoted $\Delta[e]$, where the data value is evaluated (as explained in the next subsection) ignoring tags, resulting in a ground term, i.e., a term d with only constructor functions (g), and where the tag is given by the *tag* function defined below: For tagged data values, the tag function is given by $tag(d_l) = l$, and for untagged values it is given by:

$$\begin{array}{lllllll} tag() & = & flatten(\delta[pcs]) & tag(d_l, \overline{c}) & = & l \cup tag(\overline{c}) \\ tag(g(S)) & = & \{(S, R_{current})\} \cup tag() & tag(S, \overline{c}) & = & \{(S, R_{current})\} \cup tag(\overline{c}) \\ tag(g(\overline{c})) & = & tag(\overline{c}), \textbf{otherwise} & tag(d, \overline{c}) & = & tag(d) \cup tag(\overline{c}), \textbf{otherwise} \end{array}$$

Note that the tag includes *flatten*(δ[*pcs*]), defined as the *union* of all tags in the stack *pcs*. An untagged product $(..., S, ...)$ will also include the tag $(S, R_{current})$. A pair (S, S') will be tagged with $\{(S, R_{current}), (S', R_{current}), flatten(\delta[pcs])\}$. An untagged constructor value $g(S, \overline{c})$ is tagged like the product $(S, \overline{c})$. When a subject S occurs as an argument to a constructor term or product, the pair $(S, R_{current})$ is added to the tag set. Note that $g(S)$, (S, S'), (S, c), and (c, S) include $(S, R_{current})$ in the tag set, but S and (S) do not, as a product must have at least two arguments. A tag (S, R) is redundant in a tag set l, and may be removed, if there is another tag (S, R') in l such that $R < R'$. Non-personal data will have an empty tag set. Policies are considered non-personal.

9.4.2 Runtime checking of privacy compliance

The runtime system keeps track of the current consented policy list for each subject, specifying the policies for accessing personal data concerning the subject. In the runtime system there is a mapping from subjects to policy lists

$$\mathcal{M} : Subject \rightarrow PolicyList$$

given by $\mathcal{M}[S] == S.consented$ where each *consented* is maintained by the runtime system. Note that even though remote field access is not possible within the program syntax, this restriction does not apply to the runtime system.

The evaluation of expressions, $\Delta[e]$, is done depth-first, left-to-right. Thus $\Delta[f(\overline{e})]$ is $[f(\Delta\overline{e}])$, $\Delta[\textbf{if } b \textbf{ then } e \textbf{ else } e']$ is $\Delta[e]$ if $\Delta[b]$ is true and $\Delta[e']$ if $\Delta[b]$ is false, and for a value c, $\Delta[c]$ is c. (Here b is a boolean expression.) For a defined function f, $\Delta[f(\overline{c})]$ is obtained by the definition of f replacing the formal parameters by the actual values $\overline{c}$. We let the evaluation of a variable v have a built-in compliance check of read access:

$$\begin{array}{llll} [v] & = & \delta[v], & \textbf{if } \forall (S, R) \in tag(\delta[v]) \,.\, (P_{current}, R, read) \sqsubseteq \mathcal{M}[S] \\ [v] & = & error, & \textbf{otherwise} \end{array}$$

In the first line, the tag is defined by the *tag* function in Section 9.4.1. A policy $(S, all, rincr)$ is initially added to the consented list of each subject object S, to allow the data subject to read and increment his/her own data.

For write access, we define a modified state update function $\Delta[v := c]$ so that it includes the appropriate checks for assignments, and similarly for incremental assignments. Note that there is no check on local variables since they form the local work space, i.e., a method has always write access to the local variables.

$$\begin{array}{lcll} [x := c] & = & \delta[x := c], & \\ [w := c] & = & \delta[w := c], & \textbf{if } \forall (S,R) \in tag(c) \,.\, (P_{current}, R, write) \sqsubseteq \mathcal{M}[S] \\ [w := c] & = & \delta[w := error], & \textbf{otherwise} \end{array}$$

This definition is lifted to expressions e, letting $[x := e]$ denote $[x := \Delta[e]]$. Similarly, $\Delta[v :+ c]$ requires $(P_{current}, R, incr) \sqsubseteq \mathcal{M}[S]$ for $(S, R) \in tag(c)$. Non-personal data can be accessed without restrictions since the tag is empty.

Implementation of method *requestMySensitiveData* is provided by the runtime system by making the call $caller!receiveMySensitiveData(tl)$ where tl is given by $\Delta[\overline{w}]/caller$, i.e., the tagged values of fields with *caller* in the tag.

Runtime overhead. We have given a solution for compliance checking by a runtime system formulated at a high-level of abstraction. We here discuss the overhead in tagging and checking with this solution, and how it can be reduced. By combining the core information in the tags with the dynamically changing consent information, we are able to keep the information in the tags relatively small, and moreover the tags are not changed when the consent is changed, which is a crucial property. Thus the main overhead is in accessing the consented list for the subjects in the tag. Note that the updates on each consent list is atomic, so there is no need for critical regions nor object synchronization at the runtime level. Thus a compliance check made by one object will not slow down the other objects. This processing can easily be made more efficient by letting each principal pull a copy of a subject's consent setting when needed. However, as this could lead to outdated consent information, one could use a version number for each subject's consent list, and let a principal check that it has the latest version before applying its local copy of a consent list. A further method of reducing overhead, would be to re-represent each consent list by means of a mapping (from principal and purpose of a given subject to access right) thereby the list traversal is reduced to direct look-up. This method has a cost whenever a consent list is updated. A further discussion is beyond the scope of this paper.

9.4.3 Operational rules

Each rule in the operational semantics deals with only one object o, and possibly messages, reflecting the nature of concurrent distributed active objects, communicating asynchronously. Remote calls and replies are handled by message passing. When a subconfiguration $\mathcal{C}$ can be rewritten to a $\mathcal{C}'$, this means that the whole configuration $\ldots\mathcal{C}\ldots$ can be rewritten to $\ldots\mathcal{C}'\ldots$, reflecting interleaving semantics. The operational rules reflect small-step semantics. For instance, the rule for *skip* is given by

$$o : \mathbf{ob}(\delta, skip; \overline{s}) \xrightarrow{empty} o : \mathbf{ob}(\delta, \overline{s})$$

ASSIGN :
$$o : \mathbf{ob}(\delta, v := e; \overline{s}) \xrightarrow{empty} o : \mathbf{ob}([v := e], \overline{s})$$

IF-TRUE :
$$o : \mathbf{ob}(\delta, \texttt{if}\ b\ \texttt{then}\ \overline{s1}\ \texttt{else}\ \overline{s2}\ \texttt{fi}; \overline{s}) \xrightarrow{empty} o : \mathbf{ob}(\delta[pcs := push(pcs, l)], \overline{s1}; pcs := pop(pcs); \overline{s})$$
$\texttt{if}\ [b] = true_l$

IF-FALSE :
$$o : \mathbf{ob}(\delta, \texttt{if}\ b\ \texttt{then}\ \overline{s1}\ \texttt{else}\ \overline{s2}\ \texttt{fi}; \overline{s}) \xrightarrow{empty} o : \mathbf{ob}(\delta[pcs := push(pcs, l)], \overline{s2}; pcs := pop(pcs); \overline{s})$$
$\texttt{if}\ [b] = false_l$

WHILE :
$$o : \mathbf{ob}(\delta, \texttt{while}\ b\ \texttt{do}\ \overline{s1}\ \texttt{od}; \overline{s}) \rightarrow o : \mathbf{ob}(\delta, \texttt{if}\ b\ \texttt{then}\ \overline{s1};\ \texttt{while}\ b\ \texttt{do}\ \overline{s1}\ \texttt{od}\ \texttt{fi}; \overline{s})$$

NEW :
$$o : \mathbf{ob}(\delta, v := \texttt{new}\ C(\overline{e}); \overline{s}) \xrightarrow{o\ \leftrightarrow\ \delta[\mathsf{nextOb}].C(\delta[\overline{e}])} o : \mathbf{ob}(\delta[v := o'], \overline{s})$$
$o' : \mathbf{ob}(\delta_C[\mathsf{this} \mapsto o', \mathsf{nextId} \mapsto initialFut(o'), \overline{cp} \mapsto [\overline{e}]], init_C)$
$\texttt{where}\ o' = (fresh, C),$ with $fresh$ a fresh reference relative to C

ASYNC. CALL:
$$o : \mathbf{ob}(\delta, a!m(\overline{e}); \overline{s}) \xrightarrow{o \rightarrow [a].m([\mathsf{nextId}, \overline{e}])} o : \mathbf{ob}(\delta[\mathsf{nextId} := next(\mathsf{nextId})], \overline{s})$$
$\texttt{msg}\ o \rightarrow [a].m([\mathsf{nextId}, \overline{e}])$

SYNC. CALL :
$$o : \mathbf{ob}(\delta, v := a.m(\overline{e}); \overline{s}) \xrightarrow{o \rightarrow [a].m([\mathsf{nextId}, \overline{e}])} o : \mathbf{ob}(\delta, a!m(\overline{e}); v := \texttt{get}\ \delta[\mathsf{nextId}]; \overline{s})$$

START :
$$\texttt{msg}\ o' \rightarrow o.m(u, \overline{c}) \quad o : \mathbf{ob}((\alpha|\beta'), \mathbf{idle}) \xrightarrow{o' \twoheadrightarrow o.m(u, \overline{c})} o : \mathbf{ob}((\alpha|(\beta[\mathsf{caller} \mapsto o', \mathsf{callId} \mapsto u, \overline{y} \mapsto [\overline{c}], pcs \mapsto nil])), \overline{s})$$
$\texttt{where}\ (m, \overline{y}, \beta, \overline{s})$ is the body of m in the class of this

RETURN :
$$o : \mathbf{ob}(\delta, \texttt{return}\ e) \xrightarrow{\delta[\mathsf{caller}] \leftarrow \delta[\mathsf{this}].(\delta[\mathsf{callId}], [e])} o : \mathbf{ob}(\delta, \mathbf{idle})$$
$\texttt{msg}\ \delta[\mathsf{caller}] \leftarrow \delta[\mathsf{this}].(\delta[\mathsf{callId}], [e])$

QUERY :
$$\texttt{msg}\ o \leftarrow o'.(u, c) \quad o : \mathbf{ob}(\delta, v := \texttt{get}\ u; \overline{s}) \xrightarrow{o \twoheadleftarrow o'.(u, c)} o : \mathbf{ob}(\delta, v := c; \overline{s})$$

NO-QUERY :
$$\texttt{msg}\ o \leftarrow o'.(u, c) \quad o : \mathbf{ob}(\delta, \overline{s}) \xrightarrow{o \twoheadleftarrow o'.(u, c)} o : \mathbf{ob}(\delta, \overline{s})$$
$\texttt{if get}\ u \notin \overline{s}$

Figure 9.9: Operational rules defining small-step semantics with privacy tags. Unique future identities are ensured by $initialFut$, parameterized with the parent and $next$.

saying that the execution of *skip* has no effect on the state δ of the object.

Each method call will have a unique identity u. A message has the form

$$\textbf{msg}\ o \rightarrow o'.m(u, \overline{c})$$

representing a call to m with o as caller, o' callee, and $\overline{c}$ actual parameters, or

$$\textbf{msg}\ o \leftarrow o'.(u, c)$$

representing a completion event where c is the returned value and u the identity of the call. In addition, **msg** $o \rightarrow I.m(u, \overline{c})$ denotes a broadcast to all I objects.

The semantics in Fig. 9.9 formalizes the notion of idleness, and generation of objects and messages, including a rule (*no-query*) for garbage collection of unused reply messages. Generation of identities for objects and method calls is handled by underlying semantic functions and implicit attributes.

The operational semantics uses an additional *query* statement, **get** u, for dealing with the termination of call statements. A synchronous call is treated as an asynchronous call followed by a **get** query. The query **get** u is blocking while waiting for the method response with identity u.

Assignment is handled by updating the state, requiring that there is read access to any personal data (using Δ). An if-statement requires read access to personal data in the condition and the resulting tag set l is pushed on the policy stack *pcs*, ensuring that all evaluations in the taken branch implicitly includes l in the tag set. A while loop is handled by expanding **while** b **do** s **od** to **if** b **then** s; **while** b **do** s **od** **fi** upon execution of the while-statement. Void methods return the value *void*. We assume all methods end in a return statement, including class constructors, which end in **return** *void* (although omitted in the examples). An assignment of the form $v :+ e$ is treated as an atomic operation at runtime. (When lists are implemented by linked lists, this operation can be executed by a single pointer assignment, since the value of e is not affected by other objects.) Semantically, $v :+ e$ is the same as $v := v + e$, and we do not show a special rule for it. This means that a consent update can also be considered atomic. Furthermore, we assume that initial values given to fields or local variables are expanded to assignments, as described earlier.

For simplicity, rules for broadcasting (similar to that for asynchronous calls) and local synchronous calls (i.e., queries on local calls) are omitted, since such calls do not pose additional privacy challenges. In the current semantics, a query on a local call will lead to deadlock. The handling of local queries would require addition of a stack in the object state in order to be able to push and pop unfinished local method frames, for instance as in [61].

The theorem below ensures that every access to a data subject's personal information will comply with the consented policy.

Theorem 4 (Runtime compliance)**.** *After a policy is successfully removed, all further variable accesses that need this policy will fail by giving a runtime error until the policy, or a stronger one, is added again.*

Proof. Consider an object state δ where $(S, R) \in tag(\delta[v])$. Let policy p denote $(P_{current}, R, read)$. We must prove that a runtime look-up of v in such a state gives error after a policy p' such that $p \sqsubseteq p'$ is removed from the consented list of S and before a policy p'' such that $p \sqsubseteq p''$ is added to the consented list of S.

Every variable look-up is made through one of the operational rules, by means of δ or Δ. By inspection of these rules, we observe that all program variables are evaluated by Δ apart from caller and this in rule Return, but here the *pcs* stack is empty (since a return statement occurs last in a body), so evaluation by δ in this case is the same as by Δ. It remains to show the theorem for variable access through Δ, and for an access to v we must show that $p \not\sqsubseteq \mathcal{M}[S]$.

By induction on the length of the execution we show that $\Delta[v]$ gives error between the successful removal of p' and addition of p'' to $\mathcal{M}[\mathcal{S}]$. A successful removal must perform the atomic operation $consented :+ neg(p')$ in S. Right afterwards, $neg(p')$ is the last element in $\mathcal{M}[\mathcal{S}]$ and therefore $\Delta[v]$ gives error. If a consent $neg(p''')$ is added, $p \sqsubseteq \mathcal{M}[\mathcal{S}]$ remains false. If a consent $pos(p''')$ is added, we may assume that $p \not\sqsubseteq p'''$ (otherwise p''' can be used as p'' and there is nothing to prove) and by the induction hypothesis $p \sqsubseteq \mathcal{M}[\mathcal{S}]$ remains false. ■

9.5 Related work

This paper focuses on the intersection between compliance formalization and programming languages. This line of work is relatively recent, featuring several threads of active research such as policy specification, policy enforcement, monitoring, privacy by design, language-based privacy, and role-based access control.

The work presented in [70] provides a privacy management framework for the definition of privacy agents (such as subject, controller) acting as representatives of individuals. These privacy agents play a specific role as “representative” or “proxy” of the user in order to manage personal data and ensure privacy-compliant interactions among agents. We share with [70] the objective of privacy compliant interactions, but we use an integrated style, i.e., including compliance checks within objects and actors accessing personal data. In addition, we use the same policy language for different actors and consented policies are maintained in subject objects. Cunche et al. [79] present a generic information and consent framework for IoT that allows the data subject to express privacy requirements as well as receive the information and associated privacy policy. The privacy policies for subjects and controllers are based on the PILOT semantics [91]. Privacy policies in [91] are more expressive than ours as they also encapsulate contextual information, but the semantics of policy compliance is not discussed in particular. We define fewer privacy requirements and focus on compliance formalization. The approach followed in [79] makes use of dedicated privacy agents, while we integrate the compliance checks in actor objects.

Sen et al. [102] demonstrate techniques for compliance checking in big data systems. Privacy policies are specified using a policy specification language, *LEGALEASE*, where policies can be expressed using *allow* and *deny* clauses to

permit and prohibit access. Policies can be expressed using nested allow-deny rules. Policy clauses use data store, purpose, role, and data type attributes to specify information flow restrictions. Then, a data inventory tool *GROK* maps data types in code to high-level policy concepts, and the compliance checking then reduces to a form of information flow analysis. This is similar to our approach in [109] where we associate policy with the *types* carrying sensitive information, but the difference is that the type-policy mapping is integrated in the language. The policy specification language in [102] has some similarities with our work: the semantics of policies is compositional and policies are expressed as lists of positive and negative policies. However, for the sake of simplicity, we do not consider nested-policies. All information flow restrictions (policy attributes) are encoded as a lattice in [102], while in our setting that is not the case. However, in [102] the concept lattice does not seem to distinguish with information about other subjects, which we do and in addition we can generate tags at runtime when new information (involving a subject or personal information) is created.

Yang et al. [119] propose a *policy-agnostic programming* model. Sensitive data values are associated with policies and then the programmer may implement the rest of the program in a policy agnostic manner. The language's [120] runtime system enforces these policies to ensure that only policy compliant values are used in computations. In contrast, we use generalized polices for each subject (including purpose) and minimize the information in the tags.

Other examples of language-based approaches relying on information-flow control include the role-based approach in [81] and the purpose-based approach in [53]. Myers and Liskov present a model of decentralized information flow labels, where *principals* and *labels* are the essentials of the model [81]. Principals are the entities that own, update and release (to other principals) information. A label is a set of *owner: reader* policy pairs, where *owner* is the data owner (i.e., subject in our approach), and *reader* is the principal that has read access to this data. Programs and data are annotated with such labels, and information flow restrictions are enforced by type checking. For an access to be valid, all the policy requirements of the label should be enforced, which holds in our approach as all the tags must comply with the consented policies. There are no generalized policies, and the tags will take more space than in our case. In [53], Hayati and Abadi describe an approach to model and verify aspects of privacy policies in the Jif (Java Information Flow) programming language. Data collected for a specific purpose is annotated with Jif principals and then the methods needed for a specific purpose are also annotated with Jif principals. Explicitly declaring purposes for data and methods ensures that the labeled data will be used only by the methods with connected purposes. However, this representation of purpose is not sufficient to guarantee that principals will perform actions compliant with the declared purpose. In contrast, this can be checked at runtime in our approach.

Basin et al. [11] propose an approach that relates a purpose with a business process and use formal models of inter-process communication to demonstrate GDPR compliance. Process collection is modeled as data-flow graphs which depict the data collected and the data used by the processes. Then these

processes are associated with a data purpose and are used to algorithmically generate data purpose statements, i.e., specifying which data is used for which purpose and detect violation of data minimization. A main challenge tackled by this work is to automatically generate compliant privacy policies from the model. We share with this work an explicit specification of *purpose*. In [11], a purpose is associated with a process, while in our approach a method accessing personal information is tagged with a purpose and personal data is tagged with sets of $(subject, purpose)$ pairs. This tagging is useful in generating privacy policies to check compliance.

9.6 Conclusion

We propose a consent management framework that allows a data subject to communicate and update consent policies to the controller and to view all personal data about her in the system along with the purposes for which they are used. We have considered a core language for distributed active object systems and formalized the notion of policy compliance and given an operational semantics for the considered programming language. The runtime system ensures that every access to personal data complies with the currently consented polices.

We have illustrated the feasibility of formalizing GDPR specific privacy requirements, including *privacy by design* by providing explicit specifications of *purpose* and policy constructs; *lawfulness and transparency* of processing based on consented purposes; *data subject access request* by providing predefined interfaces and classes to assist in providing the data subject with the personal data and purposes for which it is being processed.

Our framework includes a general solution for subjects to observe and change their privacy settings and for subjects to be informed about all personal data stored about them. The solution consists of a set of predefined types for privacy related concepts and a set of interfaces that forms the basis for interaction with external users, a set of classes that is used in interaction with the runtime system, and runtime checking of all access to personal data to ensure that it complies with the current privacy settings. The same framework can be reused for another language, as long as the assumption of interface abstraction is respected and as long as the purpose of any method handling personal data is identified.

Future Work: The framework can be extended to accommodate for other legal bases by having separate policy lists for each legal basis, and a logic to chose from these bases as required. More information can be included in the tags for a richer compliance check, for instance, the *data creator* can be recorded as the current principal of the method instance creating the data. More information can be included in the policy specification, for example restrictions on temporal validity, data collectors, and data creators. Furthermore, we could add cases of non-personal tag information as exceptions to the generated tags, for instance to deal with encryption. We can easily add more fine-grained methods for selection of policies/personal data in the interfaces/classes for privacy settings and data collection (from sensitive objects), for instance using purpose and principal to

limit the selection.

Acknowledgements. The authors were partially supported by IoT-Sec (NRC) and SCOTT (EU).

Authors' addresses

Shukun Tokas University of Oslo, Postboks 1337 Blindern, 0316 Oslo, Norway, shukunt@uio.no

Olaf Owe University of Oslo, Postboks 1337 Blindern, 0316 Oslo, Norway, olaf@uio.no

Bibliography

[1] Adams, R. and Schupp, S. "Constructing Independently Verifiable Privacy-Compliant Type Systems for Message Passing Between Black-Box Components". In: *Verified Software. Theories, Tools, and Experiments.* Springer, 2018, pp. 196–214.

[2] Agrawal, R. et al. "An XPath-based preference language for P3P". In: *Proceedings of the 12th international conference on World Wide Web.* ACM. 2003, pp. 629–639.

[3] Anthonysamy, P. et al. "Inferring semantic mapping between policies and code: the clue is in the language". In: *Intern. Symposium on Engineering Secure Software and Systems.* Springer. 2016, pp. 233–250.

[4] Arfelt, E., Basin, D., and Debois, S. "Monitoring the GDPR". In: *European Symposium on Research in Computer Security.* Springer. 2019, pp. 681–699.

[5] Article 29 Working Party. *Guidelines on Consent under Regulation 2016/679.* https://ec.europa.eu/newsroom/article29/item-detail.cfm?item_id=623051. Accessed: 2020-02-05.

[6] Ashley, P. et al. "Enterprise privacy authorization language (EPAL)". In: *IBM Research* vol. 30 (2003), p. 31.

[7] Askarov, A. et al. "Termination-insensitive noninterference leaks more than just a bit". In: *European symposium on research in computer security.* Springer. 2008, pp. 333–348.

[8] Austin, T. H. and Flanagan, C. "Efficient purely-dynamic information flow analysis". In: *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security.* 2009, pp. 113–124.

[9] Barth, A. et al. "Privacy and contextual integrity: framework and applications". In: *2006 IEEE Symposium on Security and Privacy (S&P'06).* May 2006, 15 pp.–198.

[10] Barth, A. "Design and analysis of privacy policies". PhD thesis. Stanford University, 2008.

[11] Basin, D., Debois, S., and Hildebrandt, T. "On purpose and by necessity: compliance under the GDPR". In: *Proceedings of Financial Cryptography and Data Security* vol. 18 (2018), pp. 20–37.

[12] Basin, D. et al. "Runtime monitoring of metric first-order temporal properties". In: *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science.* Schloss Dagstuhl-Leibniz-Zentrum für Informatik. 2008.

[13] Bennett, C. J. and Raab, C. D. *The governance of privacy: Policy instruments in global perspective*. Routledge, 2017.

[14] Beringer, L. “End-to-end multilevel hybrid information flow control”. In: *Asian Symposium on Programming Languages and Systems*. Springer. 2012, pp. 50–65.

[15] Besik, S. I. and Freytag, J.-C. “Ontology-Based Privacy Compliance Checking for Clinical Workflows”. In: (2019).

[16] Boer, F. D. et al. “A survey of active object languages”. In: *ACM Computing Surveys* vol. 50, no. 5 (2017), 76:1–76:39. URL: http://doi.acm.org/10.1145/3122848.

[17] Breaux, T. et al. *An Introduction to privacy for technology professionals*. IAPP Publication, 2020.

[18] Breaux, T. D., Hibshi, H., and Rao, A. “Eddy, a formal language for specifying and analyzing data flow specifications for conflicting privacy requirements”. In: *Requirements Engineering* vol. 19, no. 3 (2014), pp. 281–307.

[19] Broy, M. and Stølen, K. *Specification and Development of Interactive Systems*. Monographs in Computer Science. 2001.

[20] Buiras, P., Vytiniotis, D., and Russo, A. “HLIO: Mixing static and dynamic typing for information-flow control in Haskell”. In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. 2015, pp. 289–301.

[21] Cardelli, L. “Type systems”. In: *ACM Computing Surveys (CSUR)* vol. 28, no. 1 (1996), pp. 263–264.

[22] Cavoukian, A. “Privacy by design: origins, meaning, and prospects for assuring privacy and trust in the information era”. In: *Privacy protection measures and technologies in business organizations: aspects and standards*. IGI Global, 2012, pp. 170–208.

[23] Christey, S., Kenderdine, J. E., and et.al. *Common weakness enumeration (CWE version 2.9)*. ACM SIGAda Ada Letters, 2015.

[24] Clarke, R. *Introduction to dataveillance and information privacy, and definitions of terms (1999)*. http://www.rogerclarke.com/DV/Intro.html. Accessed: 2020-08-29.

[25] Clarkson, M. R. and Schneider, F. B. “Hyperproperties”. In: *Journal of Computer Security* vol. 18, no. 6 (2010), pp. 1157–1210.

[26] Colesky, M., Hoepman, J.-H., and Hillen, C. “A critical analysis of privacy design strategies”. In: *2016 IEEE Security and Privacy Workshops (SPW)*. IEEE. 2016, pp. 33–40.

[27] Cronk, J. “Strategic privacy by design”. In: *Portsmouth, NH: International Association of Privacy Professionals* (2018).

[28] Dahl, O.-J. *Object-oriented specifications, Research directions in object-oriented programming.* 1987.

[29] Dahl, O.-J. *Verifiable Programming.* 1992.

[30] Danezis, G. et al. "Privacy and Data Protection by Design-from policy to engineering". In: *arXiv preprint arXiv:1501.03726* (2015).

[31] Darvas, Á., Hähnle, R., and Sands, D. "A theorem proving approach to analysis of secure information flow". In: *International Conference on Security in Pervasive Computing.* Springer. 2005, pp. 193–209.

[32] Demurjian, S. A. et al. "Multi-Level Security in Healthcare Using a Lattice-Based Access Control Model". In: *International Journal of Privacy and Health Information Management (IJPHIM)* vol. 7, no. 1 (2019), pp. 80–102.

[33] Denning, D. E. and Denning, P. J. "Certification of programs for secure information flow". In: vol. 20, no. 7 (July 1977), pp. 504–513.

[34] Devriese, D. and Piessens, F. "Noninterference through secure multi-execution". In: *IEEE Symposium on Security and Privacy.* IEEE. 2010, pp. 109–124.

[35] Din, C. C. and Owe, O. "A sound and complete reasoning system for asynchronous communication with shared futures". In: *Journal of Logical and Algebraic Methods in Programming* vol. 83, no. 5-6 (2014), pp. 360–383.

[36] Din, C. C. et al. "Observable behavior of distributed systems: Component reasoning for concurrent objects". In: *The Journal of Logic and Algebraic Programming* vol. 81, no. 3 (2012), pp. 227–256.

[37] Dovland, J. et al. "Lazy behavioral subtyping". In: *International Symposium on Formal Methods.* Springer. 2008, pp. 52–67.

[38] Erlingsson, Ú. *The inlined reference monitor approach to security policy enforcement.* Tech. rep. Cornell University, 2003.

[39] European Data Protection Supervisor. *Data Protection.* https://edps.europa.eu/data-protection/data-protection_en. Accessed: 2020-08-24.

[40] European Parliament and Council of the European Union. *Charter of Fundamental Rights of the European Union.* https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:12012P/TXT&from=EN. Accessed: 2020-04-29. Official Journal of the European Union.

[41] European Parliament and Council of the European Union. *The General Data Protection Regulation (GDPR).* https://eur-lex.europa.eu/eli/reg/2016/679/oj. Accessed: 2019-11-24. Official Journal of the European Union.

[42] Fernández-Alemán, J. L. et al. "Security and privacy in electronic health records: A systematic literature review". In: *Journal of biomedical informatics* vol. 46, no. 3 (2013), pp. 541–562.

[43] Ferrara, P. and Spoto, F. "Static analysis for GDPR compliance". In: *Proceedings of the Second Italian Conference on Cyber Security, Milan, Italy.* (Milan, Italy, Feb. 6–9, 2018). CEUR Workshop Proceedings 2058. Proceedings available online at http://ceur-ws.org/Vol-2058/paper-10.pdf. Aachen, 2018. URL: http://ceur-ws.org/Vol-2058/.

[44] Fischer-Hbner, S. and Berthold, S. "Privacy-enhancing technologies". In: *Computer and Information Security Handbook*. Elsevier, 2017, pp. 759–778.

[45] Fischer-Hübner, S. *IT-security and privacy: design and use of privacy-enhancing security mechanisms*. Springer-Verlag, 2001.

[46] Fisher, K., Launchbury, J., and Richards, R. "The HACMS program: using formal methods to eliminate exploitable bugs". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* vol. 375, no. 2104 (2017), p. 20150401.

[47] Goguen, J. A. and Meseguer, J. "Unwinding and inference control". In: *IEEE Symposium on Security and Privacy*. 1984, pp. 75–75.

[48] Guernic, G. L. "Automata-Based Confidentiality Monitoring of Concurrent Programs". In: *Proceedings of*. 2007, pp. 218–232.

[49] Gürses, S., Troncoso, C., and Diaz, C. "Engineering privacy by design reloaded". In: *Amsterdam Privacy Conference*. 2015, pp. 1–21.

[50] Hammer, C. "Experiences with PDG-based IFC". In: *International Symposium on Engineering Secure Software and Systems*. Springer. 2010, pp. 44–60.

[51] Hammer, C. and Snelting, G. "Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs". In: *International Journal of Information Security* vol. 8, no. 6 (2009), pp. 399–422.

[52] Hansen, M. et al. *Readiness Analysis for the Adoption and Evolution of Privacy Enhancing Technologies: Methodology, Pilot Assessment, and Continuity Plan*. Tech. rep. Tech. rep., ENISA, 2015.

[53] Hayati, K. and Abadi, M. "Language-based enforcement of privacy policies". In: *International Workshop on Privacy Enhancing Technologies*. Springer. 2004, pp. 302–313.

[54] Hedin, D., Bello, L., and Sabelfeld, A. "Information-flow security for JavaScript and its APIs". In: *Journal of Computer Security* vol. 24, no. 2 (2016), pp. 181–234. URL: https://doi.org/10.3233/JCS-160544.

[55] Hedin, D. and Sabelfeld, A. "A Perspective on Information-Flow Control." In: *Software Safety and Security - Tools for Analysis and Verification*. Vol. 33. NATO Science for Peace and Security Series - D: Information and Communication Security. IOS Press, 2012, pp. 319–347.

[56] Heintze, N. and Riecke, J. G. "The SLam Calculus: Programming with Secrecy and Integrity". In: *POPL'98*. POPL'98. ACM, 1998, pp. 365–377.

[57] Hewitt, C., Bishop, P., and Steiger, R. “A universal modular ACTOR formalism for artificial intelligence”. In: *Proceedings of the Third International Joint Conference on Artificial Intelligence*. IJCAI’73. Stanford, USA: Morgan Kaufmann Publishers Inc., 1973, pp. 235–245. URL: http://dl.acm.org/citation.cfm?id=1624775.1624804.

[58] Hoare, C. A. R. *Communicating Sequential Processes*. International Series in Computer Science. 1985.

[59] Hoepman, J.-H. “Privacy design strategies”. In: *IFIP International Information Security Conference*. Springer. 2014, pp. 446–459.

[60] Johnsen, E. B. and Owe, O. “An Asynchronous Communication Model for Distributed Concurrent Objects”. In: *Software and Systems Modeling* vol. 6, no. 1 (2007), pp. 35–58.

[61] Johnsen, E. B. and Owe, O. “An asynchronous communication model for distributed concurrent objects”. In: *Software & Systems Modeling* vol. 6, no. 1 (2007), pp. 39–58.

[62] Johnsen, E. B., Owe, O., and Yu, I. C. “Creol: A type-safe object-oriented model for distributed concurrent systems”. In: *Theoretical Computer Science* vol. 365, no. 1-2 (2006), pp. 23–66.

[63] Johnsen, E. B. et al. “Intra-Object Versus Inter-Object: Concurrency and Reasoning in Creol”. In: vol. 243 (July 2009), pp. 89–103.

[64] Joshi, R. and Leino, K. R. M. “A semantic approach to secure information flow”. In: *Science of Computer Programming* vol. 37, no. 1-3 (2000), pp. 113–138.

[65] Karami, F., Owe, O., and Ramezanifarkhani, T. “An evaluation of interaction paradigms for active objects”. In: *Journal of Logical and Algebraic Methods in Programming* vol. 103 (2019), pp. 154–183. URL: https://doi.org/10.1016/j.jlamp.2018.11.008.

[66] Kremenek, T. and Engler, D. “Z-ranking: Using statistical analysis to counter the impact of static analysis approximations”. In: *International Static Analysis Symposium*. Springer. 2003, pp. 295–315.

[67] Krishnamurthi, S. “Programming Languages Application and Interpretation”. In: (2016). Accessed: 2020-10-24.

[68] Kumaraguru, P. et al. “A survey of privacy policy languages”. In: *Workshop on Usable IT Security Management (USM 07): Proceedings of the 3rd Symposium on Usable Privacy and Security, ACM*. 2007.

[69] Lampson, B. W. “Protection”. In: *ACM SIGOPS Operating Systems Review* vol. 8, no. 1 (1974), pp. 18–24.

[70] Le Métayer, D. “A formal privacy management framework”. In: *International Workshop on Formal Aspects in Security and Trust*. Springer. 2008, pp. 162–176.

[71] Le Métayer, D. "Formal methods as a link between software code and legal rules". In: *International Conference on Software Engineering and Formal Methods*. Springer. 2011, pp. 3–18.

[72] Le Métayer, D. and Schmidt, D. "Structural operational semantics as a basis for static program analysis". In: *ACM Computing Surveys (CSUR)* vol. 28, no. 2 (1996), pp. 340–343.

[73] Liu, J. et al. "Fabric: A platform for secure distributed computation and storage". In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 2009, pp. 321–334.

[74] Mantel, H. and Sudbrock, H. "Types vs. pdgs in information flow analysis". In: *International Symposium on Logic-Based Program Synthesis and Transformation*. Springer. 2012, pp. 106–121.

[75] Marchiori, M. et al. "The platform for privacy preferences 1.0 (P3P1.0) specification". In: *World Wide Web Consortium Recommendation REC-P3P-20020416* (2002).

[76] Masoumzadeh, A. and Joshi, J. B. "PuRBAC: Purpose-aware role-based access control". In: *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer. 2008, pp. 1104–1121.

[77] medium.com. *The single most important change in data privacy regulation in 20 years: GDPR*. https://medium.com/datadriveninvestor/ the-single-most-important-change-in-data-privacy-regulat ion-in-20-years-gdpr-b9026b9acfa9. Accessed: 2019-12-20.

[78] Miller, M. *Robust Composition: Towards a Uni ed Approach to Access Control and Concurrency Control*. Johns Hopkins University, 2006.

[79] Morel, V., Cunche, M., and Le Métayer, D. "A Generic Information and Consent Framework for the IoT". In: *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*. IEEE. 2019, pp. 366–373.

[80] Myers, A. C. "JFlow: Practical Mostly-Static Information Flow Control". In: pp. 228–241.

[81] Myers, A. C. and Liskov, B. "Protecting privacy using the decentralized label model". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* vol. 9, no. 4 (2000), pp. 410–442.

[82] Ni, Q. et al. "Privacy-aware role-based access control". In: *ACM Transactions on Information and System Security (TISSEC)* vol. 13, no. 3 (2010), p. 24.

[83] Nielson, F., Nielson, H. R., and Hankin, C. *Principles of program analysis*. Springer, 2015.

[84] Nielson, F. and Nielson, H. R. "Type and Effect Systems". In: *Correct System Design: Recent Insights and Advances*. Springer, 1999, pp. 114–136. URL: https://doi.org/10.1007/3-540-48092-7_6.

[85] Nierstrasz, O. “A tour of Hybrid – A Language for Programming with Active Objects”. In: *Advances in Object-Oriented Software Engin.* Prentice-Hall, 1992, pp. 67–182.

[86] Notario, N. et al. “PRIPARE: integrating privacy best practices into a privacy engineering methodology”. In: *2015 IEEE Security and Privacy Workshops*. IEEE. 2015, pp. 151–158.

[87] *Open Web Application Security Project (OWASP) Top 10 2010 and 2013.* http://www.owasp.org/index.php. 2017.

[88] Owe, O. “Reasoning about inheritance and unrestricted reuse in object-oriented concurrent systems”. In: *International Conference on Integrated Formal Methods*. Springer. 2016, pp. 210–225.

[89] Owe, O. “Verifiable Programming of Object-Oriented and Distributed Systems”. In: *From Action Systems to Distributed Systems - The Refinement Approach.* Ed. by Petre, L. and Sekerinski, E. Chapman and Hall/CRC, 2016, pp. 61–79. URL: https://doi.org/10.1201/b20053-8.

[90] Owe, O. and Ramezanifarkhani, T. “Confidentiality of interactions in concurrent object-oriented systems”. In: *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Springer, 2017, pp. 19–34.

[91] Pardo, R. and Le Métayer, D. “Analysis of privacy policies to enhance informed consent”. In: *IFIP Annual Conference on Data and Applications Security and Privacy*. Springer. 2019, pp. 177–198.

[92] Plotkin, G. D. “Structural operational semantics”. In: *Aarhus University, Denmark* (1981).

[93] Ramezanifarkhani, T., Owe, O., and Tokas, S. “A secrecy-preserving language for distributed and object-oriented systems”. In: *Journal of Logical and Algebraic Methods in Programming* vol. 99 (2018), pp. 1–25. URL: https://doi.org/10.1016/j.jlamp.2018.04.001.

[94] Ramezanifarkhani, T. and Razzazi, M. “Principles of Data Flow Integrity: Specification and Enforcement.” In: *Journal of Information Science and Engineering* vol. 31, no. 2 (2015), pp. 529–546.

[95] Russo, A. and Sabelfeld, A. “Dynamic vs. Static Flow-Sensitive Security Analysis”. In: IEEE. 2010, pp. 186–199.

[96] Sabelfeld, A. and Myers, A. C. “Language-based information-flow security”. In: *IEEE Journal on Selected Areas in Communications* vol. 21, no. 1 (2003), pp. 5–19.

[97] Sabelfeld, A. and Russo, A. “From Dynamic to Static and Back: Riding the Roller Coaster of Information-Flow Control Research”. In: *Perspectives of Systems Informatics*. Ed. by Pnueli, A., Virbitskaite, I., and Voronkov, A. Vol. 5947. Springer, 2010, pp. 352–365.

[98] Sandhu, R. S. et al. “Role-based access control models”. In: *Computer* vol. 29, no. 2 (1996), pp. 38–47.

[99] Sayed, B. *Protection against malicious JavaScript using hybrid flow-sensitive information flow monitoring.* University of Victoria, 2015.

[100] Schneider, G. “Is privacy by construction possible?” In: *International Symposium on Leveraging Applications of Formal Methods.* Springer. 2018, pp. 471–485.

[101] Sebesta, R. W. *Concepts of programming languages.* Boston: Pearson, 2012.

[102] Sen, S. et al. “Bootstrapping privacy compliance in big data systems”. In: *2014 IEEE Symposium on Security and Privacy.* IEEE. 2014, pp. 327–342.

[103] Sieghart, P. *Privacy and computers.* Latimer New Dimensions, 1976.

[104] Simonet, V. *The Flow Caml System. Software release.* July 2003.

[105] Standard, O. *Extensible access control markup language (XACML) version 2.0.* 2005.

[106] Tokas, S. and Owe, O. *A Formal Framework for Consent Management.* Proceedings available online at https://doi.org/10.23658/taltech.nwpt/2019. Tallinn, Estonia, Nov. 2019.

[107] Tokas, S. and Owe, O. “A Formal Framework for Consent Management”. In: *International Conference on Formal Techniques for Distributed Objects, Components, and Systems.* Springer. 2020, pp. 169–186.

[108] Tokas, S., Owe, O., and Johansen, C. *Code Diversification Mechanisms for Internet of Things.* Tech. rep. 35645. available online at https://www.duo.uio.no/bitstream/handle/10852/75932/1/testmain.pdf. University of Oslo, Department of Informatics, 2020.

[109] Tokas, S., Owe, O., and Ramezanifarkhani, T. “Language-Based Mechanisms for Privacy by Design”. In: *Privacy and Identity Management 2019.* IFIP Advances in Information and Communication Technology 576. Springer. Brugg/Windisch, Switzerland, Aug., 2019, 2020, pp. 142–158. URL: https://doi.org/10.1007/978-3-030-42504-3_10.

[110] Tokas, S., Owe, O., and Ramezanifarkhani, T. “Static Checking of GDPR-Related Privacy Compliance for Object-Oriented Distributed Systems”. In: *Journal of Logical and Algebraic Methods in Programming* (2019). under review.

[111] Tokas, S. and Ramezanifarkhani, T. *Language-Based Support for GDPR-Related Privacy Requirements.* Proceedings available online at https://nwpt2018.ifi.uio.no/proceedings.pdf. Oslo, Norway, 2018.

[112] Tschantz, M. C. and Wing, J. M. “Formal Methods for privacy”. In: *International Symposium on Formal Methods.* Springer. 2009, pp. 1–15.

[113] Tse, S. and Zdancewic, S. “Run-time principals in information-flow type systems”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* vol. 30, no. 1 (2007), 6–es.

[114] Ustaran, E. *European Data Protection: Law and Practice*. an IAPP Publication, International Association of Privacy Professionals, 2018.

[115] Venkatakrishnan, V. N. et al. "Provably correct runtime enforcement of non-interference properties". In: *International Conference on Information and Communications Security*. Springer. 2006, pp. 332–351.

[116] Volpano, D., Irvine, C., and Smith, G. "A sound type system for secure flow analysis". In: *Journal of computer security* vol. 4, no. 2-3 (1996), pp. 167–187.

[117] Wardell, D. C. et al. "A method for revealing and addressing security vulnerabilities in cyber-physical systems by modeling malicious agent interactions with formal verification". In: *Procedia computer science* vol. 95 (2016), pp. 24–31.

[118] Wing, J. M. "A symbiotic relationship between formal methods and security". In: *Proceedings Computer Security, Dependability, and Assurance: From Needs to Solutions (Cat. No. 98EX358)*. IEEE. 1998, pp. 26–38.

[119] Yang, J. et al. "Preventing information leaks with policy-agnostic programming". PhD thesis. Massachusetts Institute of Technology, 2015.

[120] Yang, J., Yessenov, K., and Solar-Lezama, A. "A language for automatically enforcing privacy policies". In: *ACM SIGPLAN Notices* vol. 47, no. 1 (2012), pp. 85–96.

[121] Yang, N., Barringer, H., and Zhang, N. "A purpose-based access control model". In: *Third International Symposium on Information Assurance and Security*. IEEE. 2007, pp. 143–148.

[122] Zheng, L. and Myers, A. C. "Dynamic security labels and noninterference". In: *IFIP World Computer Congress, TC 1*. Springer. 2004, pp. 27–40.

[123] Zheng, L. and Myers, A. C. "Dynamic security labels and static information flow control". In: *International Journal of Information Security* vol. 6, no. 2 (2007), pp. 67–84. URL: http://dx.doi.org/10.1007/s10207-007-0019-9.